The Problem

In data science, machine learning, and statistical tasks we are often asked to solve what is called a supervised machine learning problem. Such a problem usually involves at least:

- Learning an approximation function `f()` such that `f(x_i) ~ y_i`, where `x_i` is a vector of values and `y_i` a scalar number or a categorical value and `i` is the "row index" or the name of which example `(x_i, y_i)` we are working with. The data used to estimate or fit this function is called "the training data."
- Estimating the quality of `f()` on new data: that is, how close `f(x)` will be to future `y`-values.
- Later applying the model to unseen future data (that is assumed be distributed similarly to our training and test data). This is the application that all other steps (building model, estimating the future out of sample quality of the model) are serving. If we "go crazy" and treat one of the sub-goals as the actual overall goal we endanger this step. For instance: good performance on the training is a nice to have, but *not* an actual end goal.

What is meant by "overfitting" is: the estimated `f()` will tend to show off or over perform on the data used to fit, train, or construct it. I have some notes on this sort of selection bias here: https://win-vector.com/2020/12/10/overfit-and-reversion-to-mediocrity-the-bane-of-data-science/.

Selecting a model that "looks good" is enough to bias the model's evaluation with respect to the data set we said it "looked good" on. So even when using unbiased methods, the data scientist can introduce bias by choosing to use one model (say the one fit by logistic regression) over another (say using using an observed prevalence everywhere as a probability prediction).

Let's slow that down. Suppose you have two equivalent (or statistically exchangeable data sets) and that you train your model on one data set. You then evaluate the quality of the model on both the training data set and the left-out (let's call this hold-out or test) data set.

> **If they return different quality estimates, which one do you trust?**

Put off convincing yourself that the performance will often be different on the set used to train the model and an unused set. Concentrate only on: if they disagree, which one, if any, is more likely to be right?

It is a standard observation that:

- Models will tend to score differently on training data and held out data.
- Data you have not used looks more like future application data, as you will also not have seen future application data (else you wouldn't need a model!).
- We trust out of sample or test estimates more than estimates made on training data.

There are in fact adjustments that allow one to estimate the out of sample performance as a function of the observed in-sample or training performance. However, for complicated models (such as random forest) such adjustments become unfeasible, and one must rely on held-out data (or simulated held-out data by cross frame methods). We are trying to exhibit a general point, so we will not use the particular details of this model to make adjustments on training performance.

Also, held-out data has a great justification from a software engineering perspective.

If you have never tried your model on data not seen during model construction and selection, then the first time it sees new data will be in production.

The point is: we want to see possible problems *before* going to production.

Let's use R to demonstrate we get different behavior on training and test data.

The Activity / Demonstration

First we attach our packages.

```
library(ggplot2)
library(wrapr)
library(sigr)
library(cdata)
# requires development version 1.3.2
# remotes::install_github('WinVector/WVPlots')
library(WVPlots)
library(parallel)
```

Some Functions

Now let's define a few functions we will use later.

```
# estimate the deviance of a probability prediction
# with respect to a TRUE/FALSE target
# For a discussion of deviance, please see:
#   https://youtu.be/d6Yr7tTzNh8
deviance <- function(prediction, truth) {
  eps <- 1.0e-6
  mean(-2 * ifelse(
    truth,
    log(pmax(prediction, eps)),
    log(pmax(1 - prediction, eps))))
}

# For a model score, pick a threshold that maximizes accuracy.
# Accuracy tends not to always be a good measure, but we
# will use it here for an example.
# For a discussion against accuracy, please see:
#   https://youtu.be/R1HBtUEbU7o
pick_threshold <- function(predictions, truth) {
  # may want to pick this threshold on additional held-out
  # data to avoid picking one that only works on training data
  # this doesn't cause a data leak, just possibly a low
  # quality choice.
  roc_curve <- build_ROC_curve(
    modelPredictions = predictions,
    yValues = truth)
  roc_curve$accuracy <- (roc_curve$Sensitivity * sum(truth) +
                           roc_curve$Specificity * sum(!truth)) /
                         length(truth)
  idx <- which.max(roc_curve$accuracy)[[1]]
  if(idx>=length(truth)) {
    return(roc_curve$Score[[idx]] - 1)
  }
  return((roc_curve$Score[[idx]] + roc_curve$Score[[idx+1]])/2)
}
```

Our Example Data

We define our synthetic data. We are using synthetic data because for synthetic data:

- We know the right answer.
- We can generate as much of it as we want.

```
set.seed(2021)

x_seq <- c(0, 0.25, 0.5, 0.75, 1)
y_fn <- function(x) {
  (x >= 0.375) & (x <= 0.875)
}
mk_data <- function(m) {
  d <- data.frame(
    x1 = sample(x_seq, size = m, replace = TRUE),
    x2 = sample(x_seq, size = m, replace = TRUE),
```

```
    x3 = runif(n = m),
    stringsAsFactors = FALSE)
  d$y <- y_fn(d$x1)
  return(d)
}
```

Our problem is simple.

- The dependent variable, or quantity we are later trying to predict is `y`.
- `y` is `TRUE` or `FALSE` and the probabilities depend on the explanatory variable `x1`.
- `x2` and `x3` are irrelevant, and it is part of the modeling process to work that out.

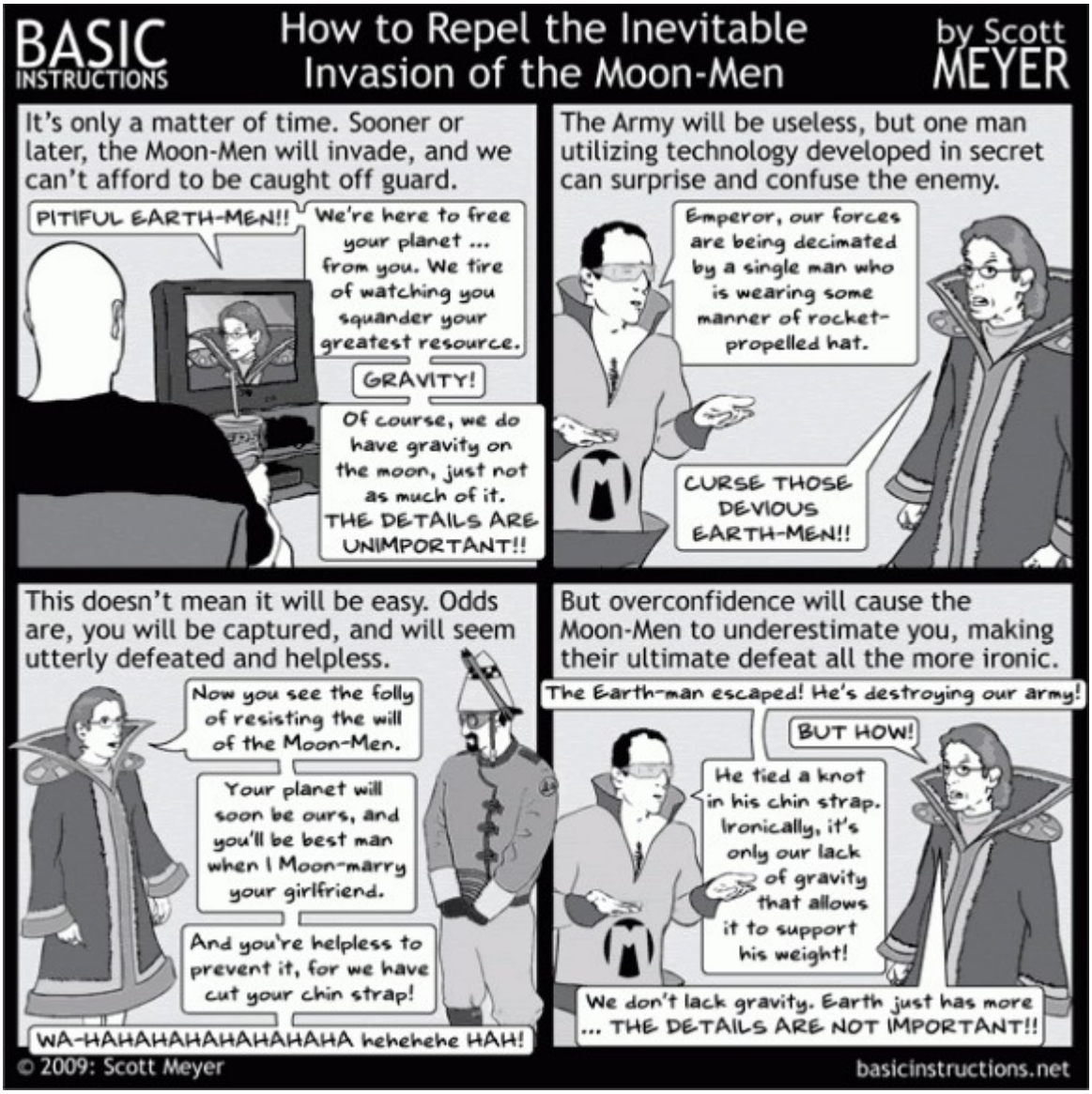
The Task

As with all supervised machine learning problems, we assume during training we have both the inputs (the explanatory variables) and the desired output (the dependent variable). The whole reason we are building a model is we assume during application we will have the values of the explanatory variables available, but not yet know the value of the dependent variable (so predicting it would be of value). An easy example is: predicting if somebody will click on an online advertisement in the next 5 minutes. If you show an ad and wait 5 minutes you have the answer. But you would like predictions to help decide which ad to show (presumably one with an estimated high chance of being clicked on).

We assume all data: training, hold-out test, and future application data is drawn from an infinite possible population of examples. This is why we think a model built on one sample may help us on another: they are different samples related by being from the same population. Under this frequentist-style formulation probabilities are well defined: they are the frequencies of seeing different events over re-draws of the training and test data sets.

For our synthetic data we can simulate the ideal population by drawing a very large sample. We can also reason about the ideal model that would be fit on such data by building an ideal data frame that has exactly the limiting distribution of `x1` and `y` and ignores `x2` and `x3`. We collect some statistics about the ideal generalized *linear* model below. Note the actual relation isn't a generalized linear one (it is in fact non-monotonic) so we are only characterizing the best model from a given generalized linear family (in this case logistic regression), not the best possible model.

We are staying away form the details of actual modeling procedure, so we can learn evaluation procedures that work on most any modeling procedure.



"THE DETAILS ARE NOT IMPORTANT", "Basic Instructions" Copyright 2009 Scott Meyer

Characterizing The Problem

To continue, we build up some summaries on how our modeling procedure will work on "perfect data."

```
ideal_data <- data.frame(
  x1 = x_seq,
  x2 = 0,
  x3 = 0)
ideal_data$y <- y_fn(ideal_data$x1)

ideal_prevalence <- mean(ideal_data$y)
ideal_const_accuracy <- max(
  ideal_prevalence,
  1 - ideal_prevalence)
ideal_const_accuracy
```

```
## [1] 0.6
```

```
ideal_const_deviance <- 2*(
```

```
  sum(ideal_data$y)*log(ideal_prevalence) +
    sum(!ideal_data$y)*log(1-ideal_prevalence)) /
  nrow(ideal_data)
ideal_const_deviance
```

```
## [1] -1.346023
```

```
ideal_model <- glm(
  y ~ x1 ,
  family = binomial(),
  data = ideal_data)
ideal_data$predict <- predict(
  ideal_model, newdata = ideal_data,
  type = 'response')

knitr::kable(ideal_data)
```

x1	x2	x3	y	predict
0.00	0	0	FALSE	0.2103950
0.25	0	0	FALSE	0.2925468
0.50	0	0	TRUE	0.3908959
0.75	0	0	TRUE	0.4989878
1.00	0	0	FALSE	0.6071745

From the above we can work out the best achievable deviance for this model family. Deviance is a "loss" or criticism, so lower is better. We are flipping the sign to make larger better so it behaves more like accuracy. We are also normalizing deviance by dividing by the number of rows to allow us to compare data sets of varying sizes.

```
(ideal_linear_deviance <- -deviance(
  prediction = ideal_data$predict,
  truth = ideal_data$y))
```

```
## [1] -1.260473
```

We can also establish the best accuracy our model prediction gives. We emphasize a good model returns a continuous score, not a decision rule. Some argument as to why to do this are given here: "Don't Use Classification Rules for Classification Problems", and "Against Accuracy".

```
# 0.5 is NOT always a good threshold choice!
# However, most "return predicted class" methods
# quietly hard-code this threshold.
(ideal_linear_threshold <- pick_threshold(
  predictions = ideal_data$predict, truth = ideal_data$y))
```

```
## [1] 0.3417213
```

```
(ideal_linear_accuracy <- mean(
  ideal_data$y == (ideal_data$predict >= ideal_linear_threshold)))
```

```
## [1] 0.8
```

Modeling

Now let's try and fit models to `100` rows of training data. We will use `1000000` of test data to simulate being able to estimate the expected performance of the model on the infinite population the samples are being drawn from.

```
d_train <- mk_data(100)
# Use a very large test set to simulate
# access to unobserved population we are
# drawing from.
# Obviously, in practice test sets are not
# usually so much larger than training.
d_test <- mk_data(1000000)
```

A Constant or Null-Model

Our first model is null-model, or a constant we hope to do better than. It predicts the same probability for every example.

```
null_prediction <- ifelse(
  sum(d_train$y) >= nrow(d_train)/2,
  TRUE,
  FALSE)
null_prediction
```

```
## [1] FALSE
```

I like to set up some small data frames for plotting. In this case we have only the outcome we are trying to predict and our simple constant prediction.

```
plot_train <- data.frame(
  y = d_train$y)

plot_train$pred_class_null <- null_prediction
head(plot_train)
```

```
##       y pred_class_null
## 1 FALSE           FALSE
## 2  TRUE           FALSE
## 3  TRUE           FALSE
## 4  TRUE           FALSE
## 5 FALSE           FALSE
## 6 FALSE           FALSE
```

This model has the following accuracy when used with a threshold of `1/2` (which is optimal for well-calibrated models that do not vary or are constants such as this, but not for models in general).

```
# accuracy is usually not a good choice of metric
# https://win-vector.com/2020/12/27/against-accuracy/
accuracy_train_null <- sum(plot_train$y == plot_train$pred_class_null)
/
                       length(plot_train$y)
accuracy_train_null
```

```
## [1] 0.57
```

And we can set up the same plotting frame for test data.

```
plot_test <- data.frame(
  y = d_test$y)

plot_test$pred_class_all <- null_prediction
head(plot_test)
```

```
##       y pred_class_all
## 1 FALSE          FALSE
## 2  TRUE          FALSE
## 3 FALSE          FALSE
## 4  TRUE          FALSE
## 5  TRUE          FALSE
## 6 FALSE          FALSE
```

```
accuracy_test_null <- sum(plot_test$y == plot_test$pred_class_all) /
  length(plot_test$y)
accuracy_test_null
```

```
## [1] 0.600176
```

We see constant models are simple, so they show similar performance on training data (where we estimate the outcome probability or prevalence) and test data.

A Real Model

Let's now fit our generalized linear model.

```
model_all <- glm(
  y ~ x1 + x2 + x3,
  family = binomial(),
  data = d_train)

summary(model_all)
```

```
##
## Call:
## glm(formula = y ~ x1 + x2 + x3, family = binomial(), data = d_train)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.6313  -0.9648  -0.6999   1.1816   1.6260
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  -1.3290     0.6386  -2.081   0.0374 *
## x1            1.5825     0.6171   2.564   0.0103 *
## x2           -0.5086     0.5938  -0.857   0.3917
## x3            0.9767     0.7469   1.308   0.1910
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
```

```
##
##     Null deviance: 136.66  on 99  degrees of freedom
## Residual deviance: 127.43  on 96  degrees of freedom
## AIC: 135.43
##
## Number of Fisher Scoring iterations: 4
```

For every proposed decision threshold we can estimate utilities. The utilities we are estimating (in this case on training data, so this is a biased estimate!) are:

- Sensitivity: what fraction of the positive outcome cases have scores above the given threshold. We would like this to be `1`.
- Specificity: what fraction of the negative outcome cases have scores below the given threshold. We would like this to be `1`.
- Accuracy: what fraction of cases do we "get right" in the sense positive outcomes are selected and negative outcomes are rejected. We would like this to be `1`.

In all cases: the proposed thresholds are on the `x`-axis, and the quantities are in different panels or facets of the graph.

We are deliberately treating the linear model as a black-box and not using linear model theory or summary items to evaluate the model. This gets us familiar with procedures that can be used on any model type and also can be used to compare different modeling techniques.

```
plot_train$prediction_all <- predict(
  model_all,
  newdata = d_train,
  type = 'response')

ThresholdPlot(
  plot_train,
  xvar = 'prediction_all',
  metrics = c("sensitivity", "specificity", "accuracy"),
  truthVar = 'y',
  title = 'All variable model on training') +
  geom_vline(
    xintercept = 0.5,
    color = "Brown",
    linetype = 3)
```

Dr. Nina Zumel motivates the `ThresholdPlot` here, and discusses the limits of accuracy as an evaluation measure here.

In the above the vertical dotted brown line depicts where the `0.5` decision threshold that is hard-coded into so many decision rules lies. Where it crosses the curves we see what sensitivity, specificity, and accuracy would be induced by using comparison to this threshold to change our model score into a decision rule.

We can also pick an optimal accuracy score threshold. There is a bias in using the training data to pick it: we may end up with a threshold that works well on training data not well on future application data. For simple liner models such as our example the effect is not large or important. For models such as random forest the effect is critical, and we would need a group of held-out calibration data to pick the threshold or need to use cross-frame techniques to simulate having such data.

```
# 0.5 is NOT always good threshold choice!
# However, most "return predicted class" methods
# quietly hard-code this threshold.
threshold_all <- pick_threshold(
  predictions = plot_train$prediction_all,
  truth = plot_train$y)
threshold_all
```

```
## [1] 0.4085472
```

Notice the ideal threshold is not `0.5`.

In the following plot frame we see how the predicted outcome (`TRUE`/`FALSE`) relates to the prediction score being above our below our decision threshold.

```
plot_train$pred_class_all <- plot_train$prediction_all >= threshold_all
head(plot_train)
```

```
##       y pred_class_null prediction_all pred_class_all
## 1 FALSE           FALSE      0.2415958          FALSE
## 2  TRUE           FALSE      0.5433782           TRUE
## 3  TRUE           FALSE      0.6920609           TRUE
## 4  TRUE           FALSE      0.3229556          FALSE
## 5 FALSE           FALSE      0.6995049           TRUE
## 6 FALSE           FALSE      0.4041549          FALSE
```

Now that we have a hard decision rule, we can estimate the accuracy of this rule.

```
accuracy_train_all <- sum(
  plot_train$y == plot_train$pred_class_all)/length(plot_train$y)
accuracy_train_all
```

```
## [1] 0.71
```

Working With Held-Out Data

We can repeat the same analysis on held out test data. Notice we are not re-fitting the threshold on test data. When we don't re-fit we are testing how a threshold chosen prior to seeing the test data will behave on test data. We want test data performance to be good, but taking steps to optimize our decisions on test data ruins the validity of comparing test data to future application data. Such adjustments effectively reduce the test data to a sub-species of training data or calibration data; meaning they are possibly no longer sources of unbiased estimates of future out of sample perfomance.

```
plot_test$prediction_all <- predict(
  model_all,
  newdata = d_test,
  type = 'response')

plot_test$pred_class_all <- plot_test$prediction_all >= threshold_all
head(plot_test)
```

```
##       y pred_class_all prediction_all
## 1 FALSE          FALSE      0.3256321
## 2  TRUE           TRUE      0.4695278
## 3 FALSE          FALSE      0.2285854
## 4  TRUE          FALSE      0.3905929
## 5  TRUE           TRUE      0.6014191
## 6 FALSE          FALSE      0.2804365
```

```
accuracy_test_all <- sum(plot_test$y == plot_test$pred_class_all) /
                      length(plot_test$y)
accuracy_test_all
```

```
## [1] 0.657773
```

Notice this test accuracy (0.657773) is less than the observed training accuracy (0.71), but greater than the estimated fit constant model accuracy (0.600176, which itself is very close to the best possible constant model accuracy for this problem: 0.6).

Trying Many Different Training Set Sizes

Now we collect all the above steps into a re-usable function. We will run this function repeatedly on varying training set sizes to see what performance we get. This allows us to characterize model performance for this family of models as a function of how much training data we have.

The code is only included here for completeness, I suggest skimming to where we resume discussion.

```
est_perf <- function(m) {
  d_train <- mk_data(m)
  model_all <- suppressWarnings(glm(
    y ~ x1 + x2 + x3,
    family = binomial(),
    data = d_train))
  train_prediction_all <- suppressWarnings(predict(
    model_all,
    newdata = d_train,
    type = 'response'))
  threshold_all <- pick_threshold(
```

```
    predictions = train_prediction_all,
    truth = d_train$y)
  accuracy_train_all <- mean(
    d_train$y == (train_prediction_all >= threshold_all))
  deviance_train_all <- deviance(
    prediction = train_prediction_all,
    truth = d_train$y)
  test_prediction_all <- suppressWarnings(predict(
    model_all,
    newdata = d_test,
    type = 'response'))
  accuracy_test_all <- mean(
    d_test$y == (test_prediction_all >= threshold_all))
  deviance_test_all <- deviance(
    prediction = test_prediction_all,
    truth = d_test$y)
  return(data.frame(
    m = m,
    train_accuracy = accuracy_train_all,
    train_deviance = deviance_train_all,
    test_accuracy = accuracy_test_all,
    test_deviance = deviance_test_all))
}

if(!file.exists('evals.csv')) {
  m_seq <- c(3:1000)
  m_reps <- pmin(50, pmax(3, floor(5000/m_seq)))
  r_seq <- lapply(
    seq_len(length(m_seq)),
    function(i) rep(m_seq[[i]], m_reps[[i]]))
  r_seq <- unlist(r_seq)
  ncore <- detectCores()
  evals <- mclapply(
    r_seq,
    est_perf,
    mc.cores =  ncore)
  evals <- do.call(rbind, evals)
  write.csv(
    evals,
    file = 'evals.csv',
    row.names = FALSE)
} else {
  evals <- read.csv(
    'evals.csv',
    stringsAsFactors = FALSE,
    strip.white = TRUE)
}
```

Analyzing The Results

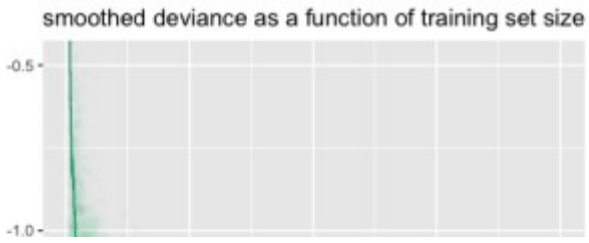
We now re-arrange the data into a more ready to plot scheme. Some notes on the theory of data arrangment can be found here.

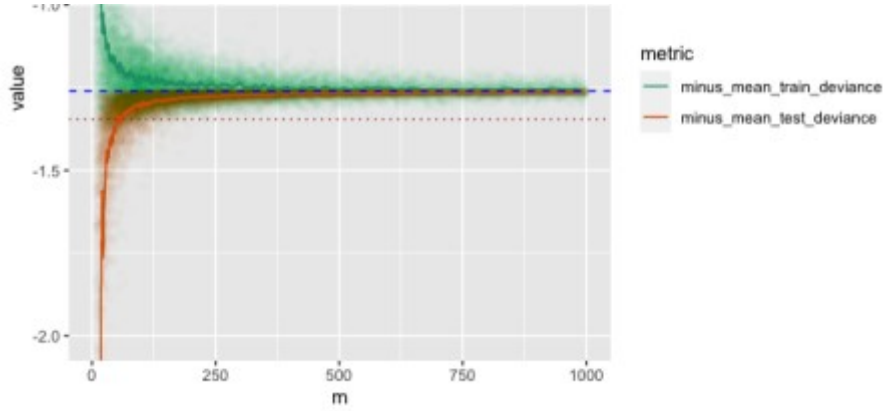
```
evals <- evals[complete.cases(evals), , drop = FALSE]
evals$minus_mean_train_deviance <- -evals$train_deviance
evals$minus_mean_test_deviance <- -evals$test_deviance
evals$train_deviance <- NULL
evals$test_deviance <- NULL

evals_plot <- pivot_to_blocks(
  evals,
  nameForNewKeyColumn = 'metric',
  nameForNewValueColumn = 'value',
  columnsToTakeFrom = c(
    'train_accuracy', 'minus_mean_train_deviance',
    'test_accuracy', 'minus_mean_test_deviance'))

# get levels in order so key matches plot heights
evals_plot$metric <- factor(
  evals_plot$metric,
  levels = c(
    'train_accuracy', 'test_accuracy',
    'minus_mean_train_deviance', 'minus_mean_test_deviance'))
```

Deviance Results

This graph is the one to memorize from this demonstration. In this graph high values are good and low values are bad.

```
ConditionalSmoothedScatterPlot(
  evals_plot[evals_plot$metric %in%
               c('minus_mean_train_deviance',
'minus_mean_test_deviance'), ],
  xvar = 'm',
  yvar = 'value',
  point_color = "Blue",
  point_alpha = 0.01,
  k = 101,
  groupvar = 'metric',
  title = 'smoothed deviance as a function of training set size') +
  coord_cartesian(ylim = c(-2, -0.5)) +
  geom_hline(
    yintercept = ideal_linear_deviance,
    color = "Blue",
    linetype = 2) +
  geom_hline(
    yintercept = ideal_const_deviance,
    color = "Brown",
    linetype = 3)
```





The above graph shows that: `-deviance` (the quantity that the logistic regression model fitter maximizes, though this is more commonly said that it minimizes `deviance`, not maximizes `-deviance`) varies as training set size increases. The lesson to internalize is:

- More training data makes the training performance *worse* with respect to the metric being optimized. This is because more ideas that look good on small data (funny coincidences that won't repeat often in practice) are falsified as we have more training data.
- More training data often makes test performance *better*. This is because with the bad ideas falsified, we hope the good ones move forward (even subtle ones).
- The difference in height between the two curves is called the "excess generalization error." It is how much worse the model performs on test data than on training data.
- Both measurements are noisy estimates of ideal population values, with more uncertainty for smaller samples. We *can* see test performance above training performance from time to time. That is evidence that at least one of these data sets is too small for reliable estimation.
- The estimate on training data is upward biased. It tells you good news, even if there is none.

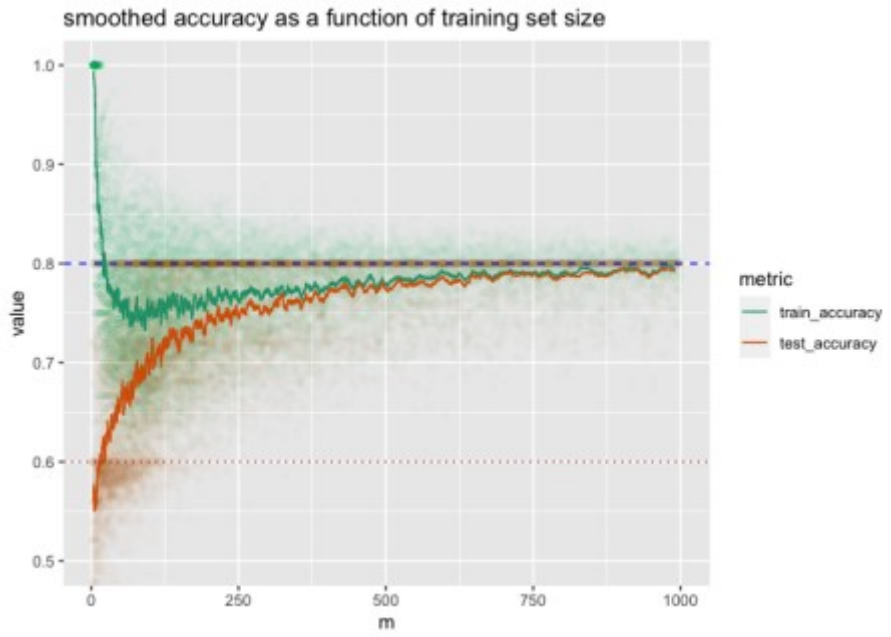
The above graph as produced by smoothing or averaging repeated runs into batches of `101`. We see the graph is still quite jumpy, so the underlying performance data (depicted as the high transparency dots) is quite wild with significant variance. The horizontal dashed blue line is the optimal `-deviance` achievable by this type of model on this problem. The dotted brown line is the `-deviance` achievable by a constant or null model. Notice for small training sets (10 rows or so) the test model performance is worse than the constant model!

Accuracy Results

Let's repeat the plotting procedure for the accuracy metric. Logistic regression optimizes deviance, not accuracy- but better models tend to have better accuracy (though the relation is not always very tight). Again on this graph, higher is better.

```
ConditionalSmoothedScatterPlot(
  evals_plot[evals_plot$metric %in%
               c('train_accuracy', 'test_accuracy'), ],
  xvar = 'm',
  yvar = 'value',
  point_color = "Blue",
  point_alpha = 0.01,
  k = 101,
  groupvar = 'metric',
  title = 'smoothed accuracy as a function of training set size') +
```

```
  coord_cartesian(ylim = c(0.5, 1)) +
  geom_hline(
    yintercept = ideal_linear_accuracy,
    color = "Blue",
    linetype = 2) +
  geom_hline(
    yintercept = ideal_const_accuracy,
    color = "Brown",
    linetype = 3)
```



The horizontal dashed blue line is the optimal accuracy achievable by this type of model on this problem. The dotted brown line is the accuracy achievable by a constant or null model. Again the two estimates are asymptotic to the ideal possible performance. Notice the accuracy is not monotone in data set size, this is more evidence accuracy isn't a great metric and one has reasons to prefer deviance (or some of its synonyms, please see here for deviance's relation to entropy).

Conclusion

We now have the shared experience of trying many fresh samples and different training set sizes for the same problem. This lets us see how the expected generalization error or the expected performance of the model on held out data vary as functions of training data set size. The thing to remember is: more data gives better results (as only test or out of sample results count!) and gives worse training results (as training performance is not in fact our end goal!).

Appendix

Note: we *are* aware `ggplot2` supplies default smoothers through `geom_smooth()`. We didn't use the plotting strategy as both `gam` and `loess` showed major artifacts in the plot that appear to be functions of the smoother, and not from the data. We will address this issue a later note using the following data extract.

```
sus_shape <- evals_plot[
  evals_plot$metric == 'minus_mean_test_deviance', ]
sus_shape <- sus_shape[, c('m', 'value')]
```

```
rownames(sus_shape) <- NULL
write.csv(
  sus_shape,
  file = 'sus_shape.csv',
  row.names = FALSE)
```