# Information, Entropy, Divergence, Deviance

*Stat 341*

*March, 2017*

## Introduction

Now let's take a bit of a detour on the road to another method of assessing the predictive accuracy of a model. The route will look something like this:

Information → (Shannon) Entropy → Divergence → Deviance → Inforation Criteria (DIC and WAIC)

DIC (Deviance Information Criterion) and WAIC (Widely Applicable Information Criterion) are where we are heading. For now, you can think of them as improvements to (adjusted) $R^2$ that will work better for Bayesian models. Details to come.

## Information

Let's begin by considering the amount of information we gain when we observe some random process. Suppose that the event we observed has probability $p$. Let $I(p)$ be the amount of information we gain from observing this outcome. $I(p)$ depends on $p$ but not on the outcome itself, and should satisfy the following properties.

1. $I(1) =$

2. $I(0) =$

3. Between $p = 0$ and $p = 1$, $I()$ is

4. $I(p_1 p_2) =$

This is motivated by

The function $I()$ should remind you of a function you have seen before. We get a function with the desired properties if we define

$$I(p) =$$

Another way to think about information is that $I(p)$ measures our _____ at observing an event that has probability $p$.

## Entropy

### Definition of Entropy

Now consider a random process $X$ with $n$ outcomes having probabilities $\mathbf{p} = p_1, p_2, \ldots, p_n$. That is,

$$P(X = x_i) = p_i,$$

The amount of information for each outcome depends on $p_i$. The **Shannon entropy** of $\mathbf{p}$ (denoted $H(X)$ or $H(\mathbf{p})$ is the

average amount of entropy gained

from each observation of the random process:

$$H(X) = H(\mathbf{p}) = \sum p_i \cdot I(p_i) = -\sum p_i \log(p_i)$$

### Examples

Compute the entropy of each of the following random processes

- A "random" process $X$ that always yields the same result
- A single toss of a fair coin
- Two tosses of a fair coin
- A single toss of a 30-70 coin
- The roll of a die

In the text, the default is natural logarithms. In that case, the unit for entropy is called a _____. If we use base 2 logarithms, then the unit is called a _____ or a _____.

### Properties of Entropy

- $H(X) \geq 0$ because
- Outcomes with probability 0:
- $H(X)$, like $I(p_i)$ depends only on the probabilities, not on the outcomes themselves.
- $H$ is a **continuous** function.
- Among all distributions with a fixed number of outcomes, $H$ is **maximized** when

*all outcomes are equally likely*

- among equiprobable distributions, $H$ **increases as the number of outcomes *increases*.**
- $H$ is **additive** in the following sense:

if $X$ and $Y$ are independent, then $H(\langle X, Y\rangle) = H(X) + H(Y)$.

$H$ can be thought of as a measure of **uncertainty**. Uncertainty decreases as we make observations.

### Entropy in R

It's pretty easy to write a function to compute entropy. Give it a try. It should work something like this.

```
# in nats
H(c(0.5, 0.5))
```

```
## [1] 0.6931472
```

```
H(c(0.3, 0.7))
```

```
## [1] 0.6108643
```

```
# in shannons
H(c(0.5, 0.5), base = 2)
```

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

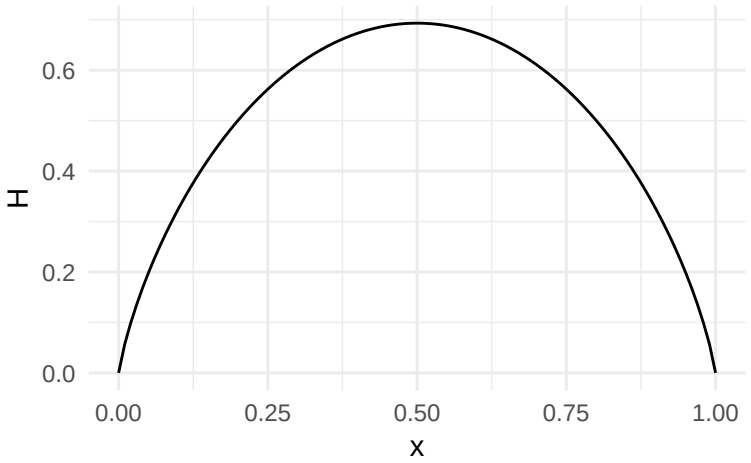
```
H(c(0.3, 0.7), base = 2)
```

```
## [1] 0.8812909
```

```
# If you want to get fancy, allow user to give all but one probability
H(0.3, base = 2)
```

```
## [1] 0.8812909
```

```
# alternatively, allow the sum to not be 1 and rescale for the user.
H_example <-
  data_frame(
    x = seq(0, 1, by = 0.01),
    H = sapply(x, H)
  )
gf_line(H ~ x, data = H_example)
```




```
H <- function(p, base = exp(1)) {
  if (sum(p) > 1) stop( "too much probability")
  if (sum(p) < 1) p <- c(p, 1 - sum(p))
  p <- p[p!= 0]
  - sum(p * log(p, base = base))
}
```

## Divergence

Kullback-Leibler (KL) divergence compares two distributions and asks "if we are anticipating $\mathbf{q}$, but get $\mathbf{p}$, how much more surprised will we be than if we had been expecting $\mathbf{p}$ in the first place?"

Here's the definition

$$\begin{aligned} D_{KL}(\mathbf{p}, \mathbf{q}) &= \text{expected difference in "surprise"} \\ &= \sum p_i \left( I(q_i) - I(p_i) \right) \\ &= \sum p_i I(q_i) - \sum p_i I(p_i) \\ &= \sum p_i (\log p_i - \log q_i) \end{aligned}$$

This looks like the difference between two entropies. It almost is. The first one is actually a **cross entropy** where we use probabilities from one distribution and information from the other. We denote this

$$H(\mathbf{p}, \mathbf{q}) = \sum p_i I(q_i) = -\sum p_i \log(q_i)$$

Note that $H(\mathbf{p}) = H(\mathbf{p}, \mathbf{p})$, so

$$\begin{aligned} D_{KL} &= H(\mathbf{p}, \mathbf{q}) - H(\mathbf{p}, \mathbf{p}) \\ &= H(\mathbf{p}, \mathbf{q}) - H(\mathbf{p}) \end{aligned}$$

Note: $H(p, q)$ is *not* symmetric.

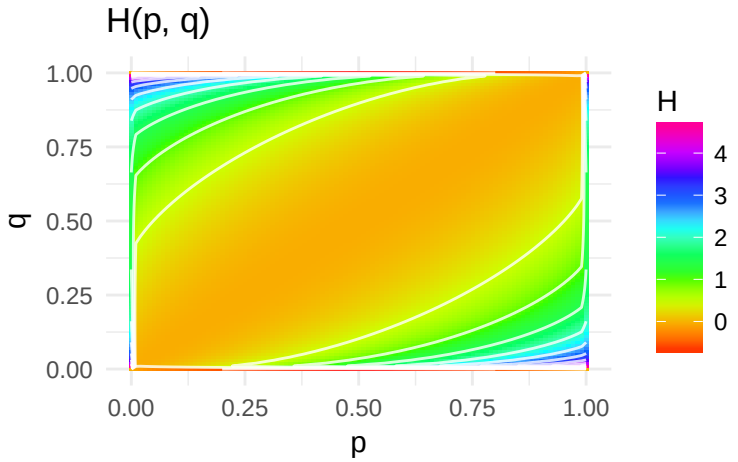
```
H(0.7, 0.01)
```

```
## [1] 2.61577
```

```
H(0.01, 0.7)
```

```
## [1] 1.139498
```

The image below shows $H(p, q)$ comparing tosses of "biased coins".



But if we want to know how well **q** approximates an unknown **p**, we still have a problem – we don't know **p**! Time for the next leg of our journey.

## Deviance

Fortunately, it usually suffices to know which of two distributions approximates our target **p** better (and by how much). And in the context of comparing models, there is a way to approximate the difference in fit without needing *direct knowledge* of **p**. So we can know (approximately) which attempt is closer to the target, and by how much, without knowing where (exactly) the target is. Of course, the target will matter some, so there must be a bit of a trick here. (There is: stay tuned.)

Let's start with some algebra. Simplify the difference below to eliminate some occurances of **p**.

- $D_{KL}(\mathbf{p}, \mathbf{q}) - D_{KL}(\mathbf{p}, \mathbf{r}) =$

**How do we apply this idea to one of our Bayesian models?**

First we need to define **p** and **q**.

- $p_i = 1/n$ for each observation $i$ in the data

  This an approximation based on our data, assuming the data were sampled according to the model specification and eliminates the final occurance of **p**.

- $q_i =$ probability (according to the model) of observing $y_i$ given the $i$th set of predictors.

If $D_{KL}(\mathbf{p}, \mathbf{q})$ is small, then our model (**q**) fits the data (**p**) well.

Using this approximation we get our definition of deviance.

$$\text{Deviance} = -2 * \text{loglik(model)}$$

Note: the scaling here is different for historical reasons, but largely irrelevant to us.

### Calculating Deviance: More Brains

How can you calculate deviance for the following model?

```
Brains <-
  data_frame(
    species =  c("afarensis", "africanus", "habilis", "boisei",
                 "rudolfensis", "ergaster", "sapiens"),
    brain_size = c(438, 452, 612, 521, 752, 871, 1350),
    body_mass =  c(37.0, 35.5, 34.5, 41.5, 55.5, 61.0, 53.5),
    body_mass.s = zscore(body_mass)
  )
```

```
m6.8 <- map(
  alist(brain_size ~ dnorm(mu, sigma),
        mu <- a + b * body_mass.s),
  data = Brains,
  start = list(
    a = mean(Brains$brain_size),
    b = 0,
    sigma = sd(Brains$brain_size)
  ),
  method = "Nelder-Mead"
)

# extract MAP estimates
theta <- coef(m6.8); theta
```

```
##        a        b    sigma
## 713.6931 225.5774 212.9408
```

**Solution 1: Rolling our own**

```
# compute deviance
dev <- (-2) * sum(dnorm(
  Brains$brain_size,
  mean = theta[1] + theta[2] * Brains$body_mass.s,
  sd = theta[3],
  log = TRUE
))
dev %>% setNames("dev")  # setNames just labels the out put
```

```
##      dev
## 94.92499
```

```
-2 * logLik(m6.8)        # for comparison
```

```
## 'log Lik.' 94.92499 (df=3)
```

**Solution 2: Out of the box**

```
# fit model with lm
m6.1 <- lm(brain_size ~ body_mass, data = Brains)

# compute deviance by cheating
(-2) * logLik(m6.1)
```

```
## 'log Lik.' 94.92499 (df=3)
```

## Deviance: In-sample vs out-of-sample

The deviance computed using our data underestimates the deviance of the model using new data. Here's a relatively simple simulation to demonstrate.
(The book uses a fancier version, but this will give you a sense for how the fancier version works.)

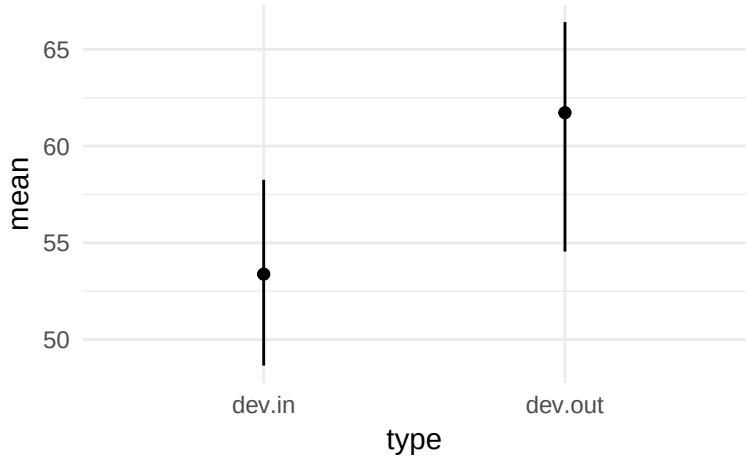
```
Dev.in.out <-
  function(n = 20, k = 3, b = c(0.15, -0.4), sigma = 1) {
  p <- length(b)

  # Create two data sets
  X <- runif(n * p, 0, 1) %>% matrix(nrow = n)
  y <- X %*% b + rnorm(n, 0, sigma)
  D.in <- cbind(y, X) %>% data.frame() %>% setNames(c("y", paste0("x", 1:p)))

  X <- runif(n * p, 0, 1) %>% matrix(nrow = n)
  y <- X %*% b + rnorm(n, 0, sigma)
  D.out <- cbind(y, X) %>% data.frame() %>% setNames(c("y", paste0("x", 1:p)))

  # Fit using the first one
  m <-
    map(
      alist(
        y ~ dnorm(mu, sigma),
        mu <- b1 * x1 + b2 * x2,
        b1 ~ dnorm(0, 5),
        b2 ~ dnorm(0, 5),
```

```
        sigma ~ dunif(0,5)
      ),
      data = D.in
    )
  # compute deviance twice: in-sample and out-of-sample
  c(
    dev.in = -2 * logLik(m),
    dev.out = -2 * sum(dnorm(D.out$y, as.matrix(D.out[,-1]) %*% coef(m)[-3], coef(m)[3],
                             log = TRUE))
  )
  }
Dev.in.out()
```

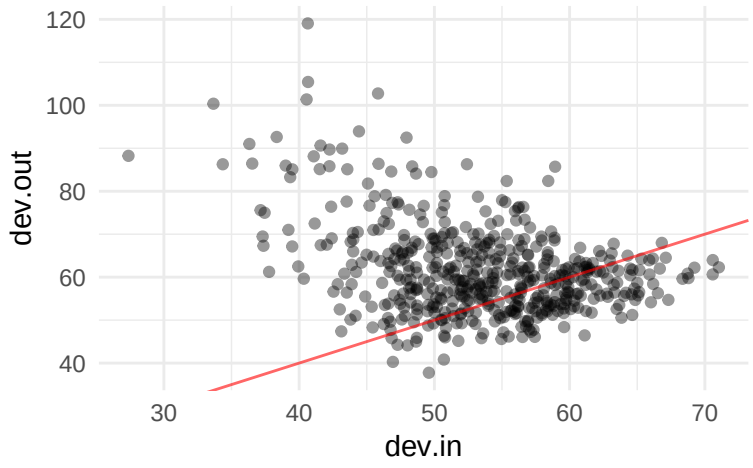
```
##   dev.in  dev.out
## 61.38255 62.19363
```

```
Sim.in.out <- do(500) * Dev.in.out()
```

```
Sim.in.out %>% gather(type, dev) %>%
  group_by(type) %>%
  summarise(mean = mean(dev), lo = quantile(dev, 0.25), hi = quantile(dev, 0.75)) %>%
  gf_pointrange( mean + lo + hi ~ type)
```




```
gf_point(dev.out ~ dev.in, data = Sim.in.out, alpha = 0.4) %>%
  gf_abline(slope = 1, intercept = 0, color = "red", alpha = 0.6)
```



```r
mean(~(dev.out - dev.in), data = Sim.in.out)
```

```
## [1] 8.345467
```

This is the same problem that (unadjusted) $R^2$ has. So we need an adjustment.

### Fancier simulation (book version)

#### R code 6.12

```r
require(tidyr)
Sims0 <- expand.grid(n = 20, k = 1:5, rep = 1:1e3)
Sims <-
  bind_cols(
    Sims0,
    apply(Sims0, 1, function(x) sim.train.test(N = x["n"], k = x["k"])) %>%
      t() %>%          # flip rows/cols
      data.frame() %>% # convert to data.frame
      setNames(c("dev.in", "dev.out"))  # give better names to vars
  )

# reshape, then compute mean and PI
Sims2 <-
  Sims %>%
  gather(dev_type, dev, dev.in : dev.out) %>%
  group_by(n, k, dev_type) %>%
  summarise(
    mean = mean(dev),
    lo = PI(dev, .50)[1],
    hi = PI(dev, .50)[2]
  )
```

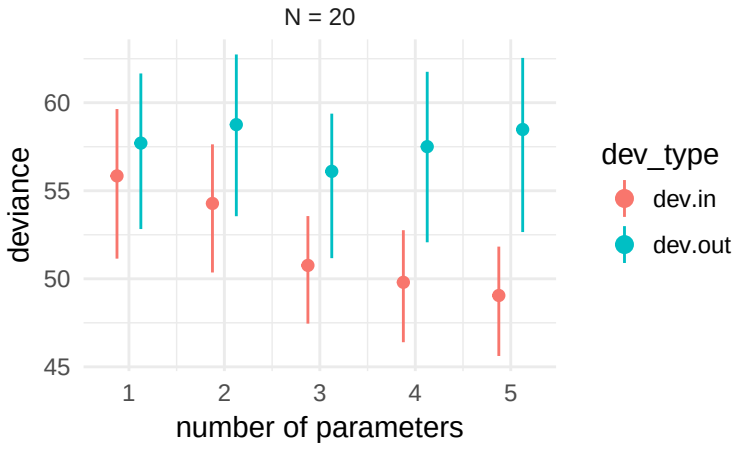
#### R code 6.13

```r
r <- mcreplicate(1e4, sim.train.test(N = N, k = k), mc.cores = 4)
```

#### R code 6.14

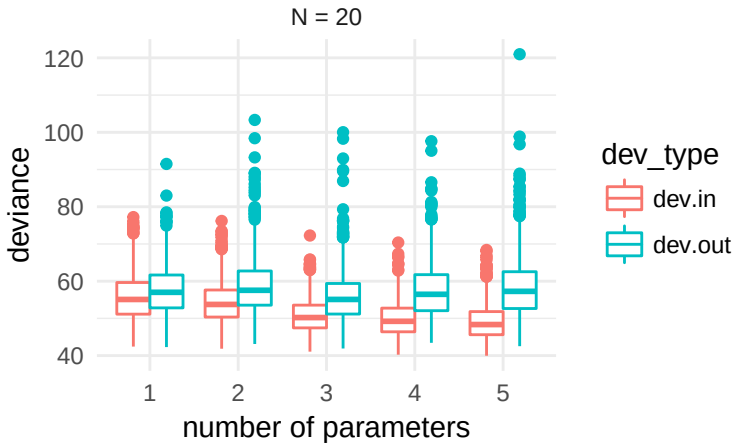
This plot uses 50% PIs (rather than $\pm$ 1 sd, as in the book) for the bars.

```r
gf_pointrange(
  mean + lo + hi ~ k + col:dev_type  | ~ paste("N =", n), data = Sims2,
  position = position_dodge(width = 0.5)) %>%
  gf_labs(x = "number of parameters", y =  "deviance")
```




Alternativley, we could use boxplots. This avoids the need to group and summarise of simulated data, but the tails of the distributions make the main story harder to see.

```
Sims %>%
  gather(dev_type, dev, dev.in : dev.out) %>%
gf_boxplot( dev ~ factor(k) + color:dev_type) %>%
  gf_facet_wrap(~ paste("N =", n)) %>%
  gf_labs(x = "number of parameters", y =  "deviance")
```



## Information Criteria

Below we present three information criteria that attempt to adjust deviance ($D$) so that it provides a better estimate for out-of-sample performance.

- $AIC = D + 2p$

  This penalizes the model for each parameter, but it is only an accurate estimate of out-of-sample performance if

  - priors are uniform (or very flat)
  - posterior distributions are multivariate normal
  - $n >> k$ (more data than parameters)

  Since flat priors are rarely best, we won't do much with AIC.

- $DIC = \overline{D} + (\overline{D} - \hat{D}) = \hat{D} + 2(\overline{D} - \hat{D})$

  Idea:

  - Take advantage of the posterior distribution for $D$. We can compute a version of $D$ for each posterior sample of the parameters of our model.
  - The difference between our MAP estimate and the average of the posterior estimates is a measure of the bias, so we can correct for it.

  Notation:

  - $\hat{D}$ is the MAP value of $D$. (Use the MAP paramter estimates to compute $D$.)
  - $\overline{D}$ is the the mean value of $D$ over the posterior sample of parameters. (Compute $D$ for each row of parameters in the posterior distribution, then take the average.)
  - $\overline{D} - \hat{D}$ is the "effective number of parameters" of our model and replaces the one-size-fits-all adjustment used in AIC. This quantity is generally bigger for "more flexible models" and smaller for "less flexible models". Priors can constrain a models flexibility and reduce the effective number of parameters.

  This includes the priors into the mix, but still assumes that

  - posterior distirbutions are multivariate normal

  We can calculate this using `DIC()`, which also provides the effective number of parameters.

```
DIC(m6.8)
```

```
## [1] 108.5619
## attr(,"pD")
## [1] 6.818456
```

- $\text{WAIC} = -2\text{llpd} - 2p_{\text{WAIC}}$.
  - $\text{llpd} = \sum_{i=1}^{n} Pr(y_i)$
  - replaces log likelihood in AIC And DIC
  - $Pr(y_i)$ is **average** likelihood (over posterior samples of parameters) for observation $i$
  - better notation: $E_\theta(L(y_i \mid \theta))$.
  - similar to log likelihood but we use posterior samples and sum the average log-likelihood for each observation over the posterior distribution of parameter values (instead of just using the MAP values)
  - this is computationally more challenging, but `WAIC()` computes it for us from a model

- $p_{\text{WAIC}} = \sum_{i=1}^{n} V(y_i)$
- effective number of parameters
- $V(y_i)$ is the variances in the log likelihood estimates for each observation (over the posterior distribution of parameters)