Generative Models for Discrete Data

Brandon Kozak

2019-09-19

```
library(tidyverse)
library(BiocManager)
library(Biostrings)
library(BSgenome.Celegans.UCSC.ce2)
```

Questions we will answer in this chapter

- Given a certain model, how can we obtain the probabilities for all possible outcomes?
- How do theoretical frequencies compare with those observed in real data?
- How can we use the Poisson distribution to analyse data on epitope detection?
- How can we apply the Multinomial distribution and Monte Carlo simulation to perform power tests?

Our first example

Let X be the number of mutations along a genome of HIV.

We are told that mutations occur at a rate of .00005 per nucleotide per replication cycle. Furthermore we assume that this genome contains 10000 nucleotides per cycle.

Thus, for one cycle, X ~ Poisson(λ = .00005 * 10000 = 5)

That's cool and all, but what does this tell us about our mutations?

Quite a lot actually! For example:

- We can expect that in the long run, each cycle will contain 5 mutations (aka the expected value of X)
- We can quantify the spread of the distribution of X in two ways
  - The standard deviation, which is equal to $\sqrt{5}$ in our example
  - The variance, which is equal to 5 in our example
  - Note that the standard deviation is just the square root of the variance.

Moreover, for any value(s) of X we can find the several probabilities:

- The probability of seeing exactly x mutations in a cycle
- The probability of seeing at least x mutations in a cycle
- The probability of seeing no more than x mutations in a cycle
- The probability of seeing between x1 and x2 mutations in a cycle (x1 < x2)

In general, a function $f : A \to [0, 1]$ that takes an integer as input (in the case of discrete data) and returns a probability as output is called a **probability mass function (PMF)**. In particular, we obtain the probability of the observable X being observed to have a value of x, denoted $P(X = x)$.

Given this, lets try some examples in R.

To obtain $P(X = x)$ where X follows a Poisson(λ) distribution, we use the function called dpois(x, lambda)

For example, what is the probability that we observe exactly 5 mutations in our genome?

```
dpois(x = 5, lambda = 5)
```

```
## [1] 0.1754674
```

If we want to obtain the probability for multiple values, say 0 through 20, we can set the first parameter (x) to be a vector.

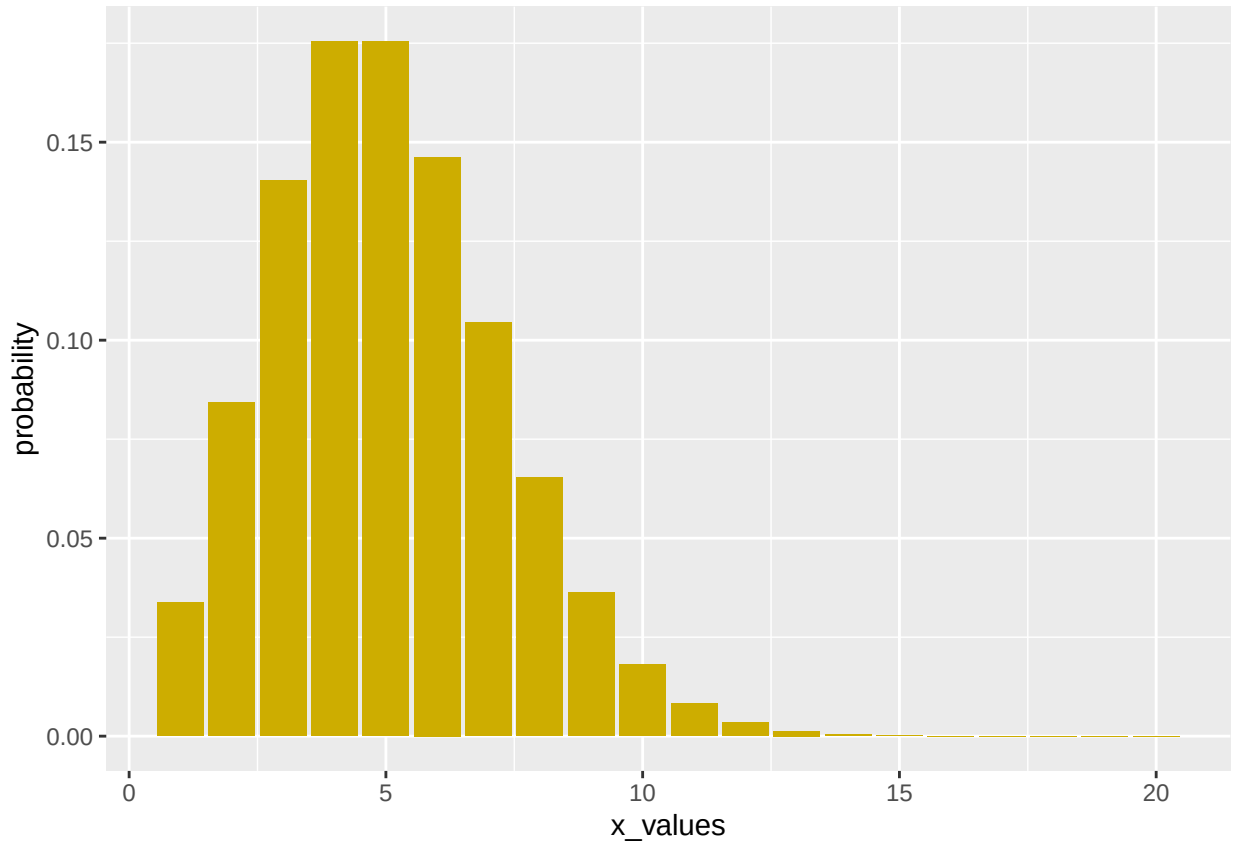
In R, we can use the colon (:) operator to generate a sequence of integers, i.e. 1:3 = (1,2,3).

```
dpois(x = 0:20, lambda = 5)
```

```
##  [1] 6.737947e-03 3.368973e-02 8.422434e-02 1.403739e-01 1.754674e-01
##  [6] 1.754674e-01 1.462228e-01 1.044449e-01 6.527804e-02 3.626558e-02
## [11] 1.813279e-02 8.242177e-03 3.434240e-03 1.320862e-03 4.717363e-04
## [16] 1.572454e-04 4.913920e-05 1.445271e-05 4.014640e-06 1.056484e-06
## [21] 2.641211e-07
```

We could use this to plot the pmf over certain values.

```
tibble(x_values = 1:20, probability = dpois(x = 1:20, lambda = 5)) %>%
  ggplot(aes(x = x_values, y = probability)) +
    geom_bar(stat = "identity", fill = "gold3")
```



Note that after some math we can show that the pmf of a Poisson distribution is $\frac{e^{-\lambda}\lambda^x}{x!}$

Factors

In this section we will discuss factors and how they work in R.

With discrete data, we often associate labels with certain values. For example:

- Sex: Male, Female
- Pain Level: Low, Medium, High
- Blood Genotypes: AA, AB, AO, BB, BO, OO
- Binary outcomes: 0 = No, 1 = Yes

We use the function factor() to convert a vector to a factor vector.

```
pain_level <- c("low", "low", "medium", "high", "low", "medium", "high", "high") %>% factor()

pain_level
```

```
## [1] low    low    medium high   low    medium high   high
## Levels: high low medium
```

You'll notice that by default R sorts factors alphabetically, to fix this we set a parameter "levels" in side the call to factor()

```
pain_level <- c("low", "low", "medium", "high", "low", "medium", "high", "high") %>%
  factor(levels = c("low", "medium", "high"))

pain_level
```

```
## [1] low    low    medium high   low    medium high   high
## Levels: low medium high
```

Question

What if you want to create a factor that has some levels not yet in your data?

Solution

We would make the same call to factor and include the extra factors while setting "levels"

```
pain_level <- c("low", "low", "medium", "high", "low", "medium", "high", "high") %>%
  factor(levels = c("none", "low", "medium", "high"))

pain_level[1] <- "none"

pain_level
```

```
## [1] none   low    medium high   low    medium high   high
## Levels: none low medium high
```

Bernoulli Trials

Consider a binary event with two possible outcomes, say whether a new born baby is male or female. Further assume that each sex is equally likely and that 0 = male, and 1 = female.

If X represents the sex of a random new born, then we can say X follows a Bernoulli distribution with probability p = .5.

Note that a Bernoulli trial with probability p is equivalent to a binomial distribution with size n and probability p. We will talk more about the binomial distribution shorty.

Lets start by generating random samples from a Bernoulli trial with p = .5. To do this, we use the rbinom() function.

Say we wanted to simulated 20 births.

```
rbinom(n = 20, prob = .5, size = 1)
```

```
##  [1] 1 0 1 0 0 1 0 0 0 0 1 1 1 1 1 0 0 0 1 0
```

Distributions in base R

At this point we've seen two ways to work with distributions in r (dpois() and rbinom()). R is actually very systematic about this and has a set of 4 functions for all the common distributions. Each function will start with one of (d, p, q, or r) and will end with an abbreviation of the name of the distribution (pois for Poisson, binom for binomial, norm for normal, etc. . .).

Here, I will state what each letter represents and how the functions work:

- d: probability distribution function, it gives $P(X = x)$ under a certain distribution.
- p: cumulative distribution function, it gives $P(X \leq x)$ under a certain distribution.
- q: quantile function, it gives the value of x for which the cumulative probability equals p under a certain distribution.
- r: random generation, it gives a random sample of size n under a certain distribution.

We will see more examples of theses functions throughout the class.

The Binomial Distribution

If we perform a Bernoulli trial n times and then ask how many of the trials ended in a success, we are sampling data from a **Binomial distribution**

Say we are looking at mothers who are planning to have three children in the future. If X represents the number of boys that a individual mother has. Then we say X follows a binomial distribution with n = 3 and p = .5.

we can take a random sample of 10 mothers

Note that n in rbinom() refers to how many samples we want to take, whereas size refers to the size of each trial (usually denoted n when we refer to the binomial distribution).

```
rbinom(n = 10, prob = .5, size = 3)
```

```
##  [1] 2 3 1 1 1 1 3 2 1 2
```

Monte Carlo Simulation

Monte Carlo is a method to solve problems with the use of randomness. There are three main problems that can be addressed with Monte Carlo: optimization, numerical integration, and generating draws from a probability distribution.

We've seen one example of Monte Carlo so far when we were doing epitope detection.

Now we will switch gears and look at how Monte Carlo can be used to simulate power for a hypothesis test.

In statistics, power is the probability that we reject our null hypothesis given that the null hypothesis is false.

Since we most often wish to reject the null hypothesis, we can view power as the **true positive rate**.

Lets say that we wish to test the null hypothesis (H_0) that DNA data we've collected comes from a fair process. That is:

$H_0 : P_A = .25, P_C = .25, P_G = .25, P_T = .25$ v.s. H_a : At least one P_i differs from the rest.

Before we define the test, lets first take 20 random samples from a multinomial distribution with n = 20 and $P_A = ... = P_G = .25$.

```
set.seed(20190919)
rmultinom(10, prob = rep(.25,4), size = 20)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    8    3    4    2    3    8    5    2    3     1
## [2,]    3    2    2    5    2    6    5    8    4     7
## [3,]    6   10    9    6   11    3    4    5    7     4
## [4,]    3    5    5    7    4    3    6    5    6     8
```

We can see that even with 20 samples, there is quite a lot of variability in the placements of the boxes.

Spoiler alert! The test we are about to simulate is called the chi-squared test for goodness of fit. Although we can do this test without any simulation (one of the focus points in chapter 2), it still a good idea to get experience with Monte Carlo simulation, as it might be the only option for certain situations.

The test statistic is defined to be:

$\chi^2 = \sum_{i=1}^{n} \frac{(E[x_i]-x_i)^2}{E[x_i]}$

That is, for each individual observed x, we take the squared difference of its expected value and the observed value, then divide by the expected value. We will show go though the process of generating this statistic directly from the above sampling.

First, let's store this matrix (this is what rmultinom() gives us as output) into an object.

```
set.seed(1234)
dna_data <- rmultinom(10, prob = rep(.25,4), size = 20)
dna_data
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    3    6    1    5    4    7    3    4    3     8
## [2,]    6    7    5    6    8    3    4    3    7     6
## [3,]    6    4    8    5    3    4    6    3    5     1
## [4,]    5    3    6    4    5    6    7   10    5     5
```

Now we will want the expected values for each x_i, since we have the probabilities, $E[X_i] = p_i * n$

Note: we can multiply a vector by a constant and R will do that multiplication for each element in the vector.

```
exp_x <- rep(.25,4) * 20
exp_x
```

```
## [1] 5 5 5 5
```

We can continue to perform these operations on the entire matrix, R handles this for us in a elegant way.

If we subtract a vector from a matrix, R will recycle the values in the vector until the last element in the matrix is reached. This works perfectly for us if the number of elements in the vector matches the number of rows in the matrix.

```
dna_data <- dna_data - exp_x
dna_data
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]   -2    1   -4    0   -1    2   -2   -1   -2     3
## [2,]    1    2    0    1    3   -2   -1   -2    2     1
## [3,]    1   -1    3    0   -2   -1    1   -2    0    -4
## [4,]    0   -2    1   -1    0    1    2    5    0     0
```

We can perform other operation on the matrix, R will sill do this element wise. Note that squaring a matrix usually means something different. In our context, however, this is exactly what we need.

```
dna_data <- dna_data^2
dna_data
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    4    1   16    0    1    4    4    1    4     9
## [2,]    1    4    0    1    9    4    1    4    4     1
## [3,]    1    1    9    0    4    1    1    4    0    16
## [4,]    0    4    1    1    0    1    4   25    0     0
```

Now we just need to divide by $E[X_i]$

```
dna_data <- dna_data / exp_x
dna_data
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]  0.8  0.2  3.2  0.0  0.2  0.8  0.8  0.2  0.8   1.8
## [2,]  0.2  0.8  0.0  0.2  1.8  0.8  0.2  0.8  0.8   0.2
## [3,]  0.2  0.2  1.8  0.0  0.8  0.2  0.2  0.8  0.0   3.2
## [4,]  0.0  0.8  0.2  0.2  0.0  0.2  0.8  5.0  0.0   0.0
```

In fact, to make things easier, lets make a function that combines all the above steps.

```
generate_stat_test <- function(matrix, exp_value){
  return((matrix-exp_value)^2 / exp_value)
}
```

Now we can generate much larger samples.

```
set.seed(1234)
dna_data <- rmultinom(1000, prob = rep(.25,4), size = 20)
dna_data_matrix_H0 <- generate_stat_test(dna_data, exp_x)
dna_data_matrix_H0[1:4,1:20]
```

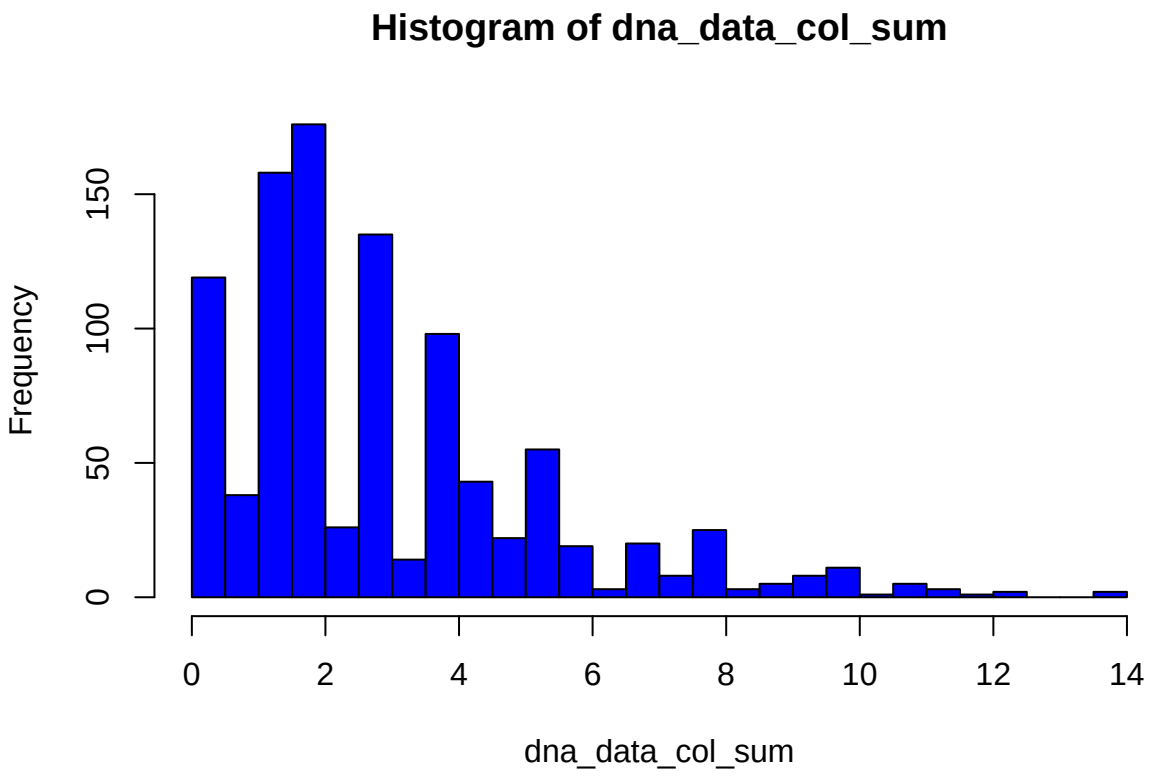
```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
## [1,]  0.8  0.2  3.2  0.0  0.2  0.8  0.8  0.2  0.8   1.8   0.0   0.0   0.8
## [2,]  0.2  0.8  0.0  0.2  1.8  0.8  0.2  0.8  0.8   0.2   0.2   0.8   0.2
## [3,]  0.2  0.2  1.8  0.0  0.8  0.2  0.2  0.8  0.0   3.2   0.0   0.8   7.2
## [4,]  0.0  0.8  0.2  0.2  0.0  0.2  0.8  5.0  0.0   0.0   0.2   0.0   1.8
##      [,14] [,15] [,16] [,17] [,18] [,19] [,20]
## [1,]   0.8   0.2   0.0   0.2   0.2   0.8   0.2
## [2,]   0.0   0.2   0.2   0.8   0.2   0.2   0.8
## [3,]   0.0   0.2   0.2   1.8   0.0   0.0   0.8
## [4,]   0.8   0.2   0.0   0.8   0.0   0.2   0.2
```

To proceed with the simulation, we will need to find the overall variability for each DNA sample. Lucky base R has a function to do this called colSums().

```
dna_data_col_sum <- colSums(dna_data_matrix_H0)
```

This is also useful if we want to plot the distribution of x.

```
hist(dna_data_col_sum, breaks = 25, col = "blue")
```



In particular, we would like to obtain the 95^{th} percentile for the overall variability of all DNA samples.

```
percent_95th <- quantile(dna_data_col_sum,.95)
```

This means that 5% of DNA samples vary by a value of 7.6. We will use this number to detect if a individual DNA sample comes from a different (not fair) process.

To do this, lets first generate DNA data again, but this time we will change the probabilities so that the process is no longer fair. That is, this new DNA will follow a multinomial distribution with $P_A = .375, P_C = .25, P_G = .25, P_T = .125$ and n = 20. We will take 1000 samples.

```
set.seed(1234)
dna_data_nf <- rmultinom(1000, prob = c(.375, .25, .25, .125), size = 20)
dna_data_nf[1:4,1:20]
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
## [1,]    5    8    3    8    6   10    6    6    6    10     7     7     6
## [2,]    7    7    5    6    8    3    4    4    7     5     4     4     4
## [3,]    5    3    7    4    5    5    7    9    5     5     7     5     3
## [4,]    3    2    5    2    1    2    3    1    2     0     2     4     7
##      [,14] [,15] [,16] [,17] [,18] [,19] [,20]
## [1,]     9     6     7     6     6     5     9
## [2,]     5     6     6     7     7     6     3
## [3,]     4     6     5     6     5     6     4
## [4,]     2     2     2     1     2     3     4
```

```
dna_data_matrix_H0_nf <- generate_stat_test(dna_data_nf, exp_x)
dna_data_matrix_H0_nf[1:4,1:20]
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
## [1,]  0.0  1.8  0.8  1.8  0.2  5.0  0.2  0.2  0.2     5   0.8   0.8   0.2
## [2,]  0.8  0.8  0.0  0.2  1.8  0.8  0.2  0.2  0.8     0   0.2   0.2   0.2
## [3,]  0.0  0.8  0.8  0.2  0.0  0.0  0.8  3.2  0.0     0   0.8   0.0   0.8
## [4,]  0.8  1.8  0.0  1.8  3.2  1.8  0.8  3.2  1.8     5   1.8   0.2   0.8
##      [,14] [,15] [,16] [,17] [,18] [,19] [,20]
## [1,]   3.2   0.2   0.8   0.2   0.2   0.0   3.2
## [2,]   0.0   0.2   0.2   0.8   0.8   0.2   0.8
## [3,]   0.2   0.2   0.0   0.2   0.0   0.2   0.2
## [4,]   1.8   1.8   1.8   3.2   1.8   0.8   0.2
```

```
dna_data_col_sum_nf <- colSums(dna_data_matrix_H0_nf)
dna_data_col_sum_nf[1:20]
```

```
##  [1]  1.6  5.2  1.6  4.0  5.2  7.6  2.0  6.8  2.8 10.0  3.6  1.2  2.0  5.2
## [15]  2.4  2.8  4.4  2.8  1.2  4.4
```

Since we know this new DNA data does, in fact, not come from a fair process, we can calculate the power of our test by taking the total number of DNA samples that vary by more than 7.6 (ie, we would have rejected that test), and divide by the total number of tests we performed (1000 in this case).

```
power <- mean(dna_data_col_sum_nf > percent_95th)
power
```

```
## [1] 0.186
```

Thus, our power is only 20% when use 20 samples per test. This brings us to an interesting question.

What is the minimum sample size (per test) required to guarantee a power of 80%?

Solution:

A good approach is to perform the simulation many time for various values of n, and then plot the relationship between the sample size and power.

First, let's define a function that will generate power.

```
get_power <- function(data, probs, crit_v) {

data_matrix_H0 <- generate_stat_test(data, (1/nrow(data)) * colSums(data[])[1])

data_col_sum <- colSums(data_matrix_H0)

power <- mean(data_col_sum > crit_v)

return(power)
}
```

The first thing we will need to do is generate 100 matrices, each containing 1000 DNA samples with sizes varying from 1 to 100.

To do this we will use a purrr function called map. A typical call to map is as follows:

map(.x, .f, ...) where:

- .x is a vector that contains the parameter you wish to vary per call.
- .f is a function that you want to use on the above vector.
- ... are any other (constant) parameters in the above function.

```
size <- c(1:200)
dna_data_n_1_to_200 <- map(size, rmultinom, n = 1000, prob = c(.375, .25, .25, .125))

# The first matrix where each DNA sample has size 1.
dna_data_n_1_to_200[[1]][,1:10]
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    1    0    0    1    0    0    1    0    0     0
## [2,]    0    0    0    0    1    1    0    0    1     0
## [3,]    0    1    1    0    0    0    0    1    0     1
## [4,]    0    0    0    0    0    0    0    0    0     0
```

```
# The last matrix where each DNA sample has size 200.
dna_data_n_1_to_200[[200]][,1:10]
```

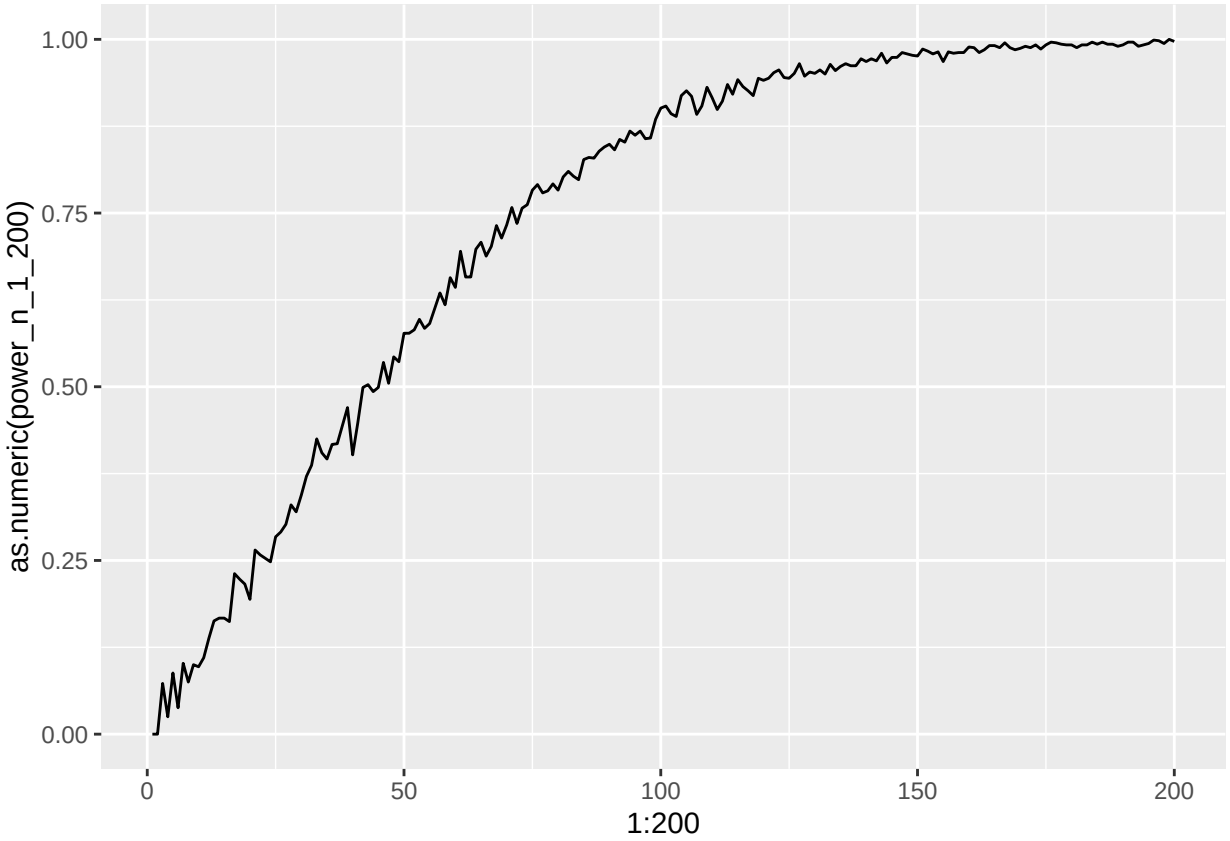
```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]   89   74   83   72   77   72   75   78   66    74
## [2,]   38   52   43   62   42   55   57   47   55    48
## [3,]   46   55   51   45   49   52   44   47   51    51
## [4,]   27   19   23   21   32   21   24   28   28    27
```

We will use map again, this time on the list of matrices using our get_power function.

```
data = dna_data_n_1_to_200
power_n_1_200 = map(data, get_power, probs = rep(.25,4), crit_v = percent_95th)
```

We can plot the relationship

```
qplot(x=1:200, y=as.numeric(power_n_1_200), geom="line")
```



And finally find the answer we were looking for!

```
detect_index(as.numeric(power_n_1_200), ~ .x >= .8)
```

```
## [1] 81
```

So, if we want our test to have a power equal to 80% we must have a minimum sample size of 81.

Exercises

1.1

Geometric Distribution:

Given a probability p, how many failures will it take to see the first success?

```
# A random sample of size 5 from a geometric distribution with p=.25
rgeom(5, .25)
```

```
## [1] 2 1 2 5 2
```

```
# What is the probability that we will see 4 failures before the first success?
dgeom(4, .25)
```

```
## [1] 0.07910156
```

```
# What is the probability that we will see no more than 3 failures before the first success?
pgeom(3, .25)
```

```
## [1] 0.6835938
```

Hypergeometric Distribution:

Given a population of size N where K of the N objects are "success states." How many success state objects will I obtain from drawing a sample of size n without replacement?

```
# A random sample of size 5 from a hyper geometric distribution with a population of N=25, K=5 success
# given a sample of n=10
rhyper(5, 5, 20, 10)
```

```
## [1] 3 1 2 2 2
```

```
# What is the probability that we will see 5 success state objects?
dhyper(5, 5, 20, 10)
```

```
## [1] 0.004743083
```

```
# What is the probability that we will see at least 1 success state object?
phyper(0, 5, 20, 10, lower.tail = F)
```

```
## [1] 0.9434783
```

1.2

$P(X = 2 \mid X \sim Bin(10, .3))$

```r
dbinom(x = 2, size = 10, p = .3)
```

```
## [1] 0.2334744
```

$P(X <= 2 \mid X \sim Bin(10, .3))$

```r
# Using only dbinom()
dbinom(x = 0, size = 10, p = .3) + dbinom(x = 1, size = 10, p = .3) + dbinom(x = 2, size = 10, p = .3)
```

```
## [1] 0.3827828
```

```r
# Using pbinom()
pbinom(q = 2, size = 10, p = .3)
```

```
## [1] 0.3827828
```

1.3

```r
pois_max = function(n, max, lamda) {
  # First calculate P(X >= max) = 1 - P(X <= max - 1)
  prob = ppois(max-1, lamda)
  # Then, as we showed before using order statistics, calculate P(X(n) >= max) = P(X(n) <= max-1)
  prob_max = 1 - prob^n
  return(prob_max)
}
```

1.4

```r
pois_max = function(n = 100, max = 0, lamda = 1) {
  # First calculate P(X >= max) = 1 - P(X <= max - 1)
  prob = ppois(max-1, lamda)
  # Then, as we showed before using order statistics, calculate P(X(n) >= max) = P(X(n) <= max-1)
  prob_max = 1 - prob^n
  return(prob_max)
}
```

1.5

```r
# Real answer
pois_max(100, 9, .5)
```

```
## [1] 3.43549e-07
```

```r
# Simulation

pois_has_max = function(n, max, lamda) {

result_vector = rpois(n, lamda)

return(max(result_vector >= max))
}

a = replicate(1e5, pois_has_max(100, 9, .5))
mean(a)
```

```
## [1] 0
```

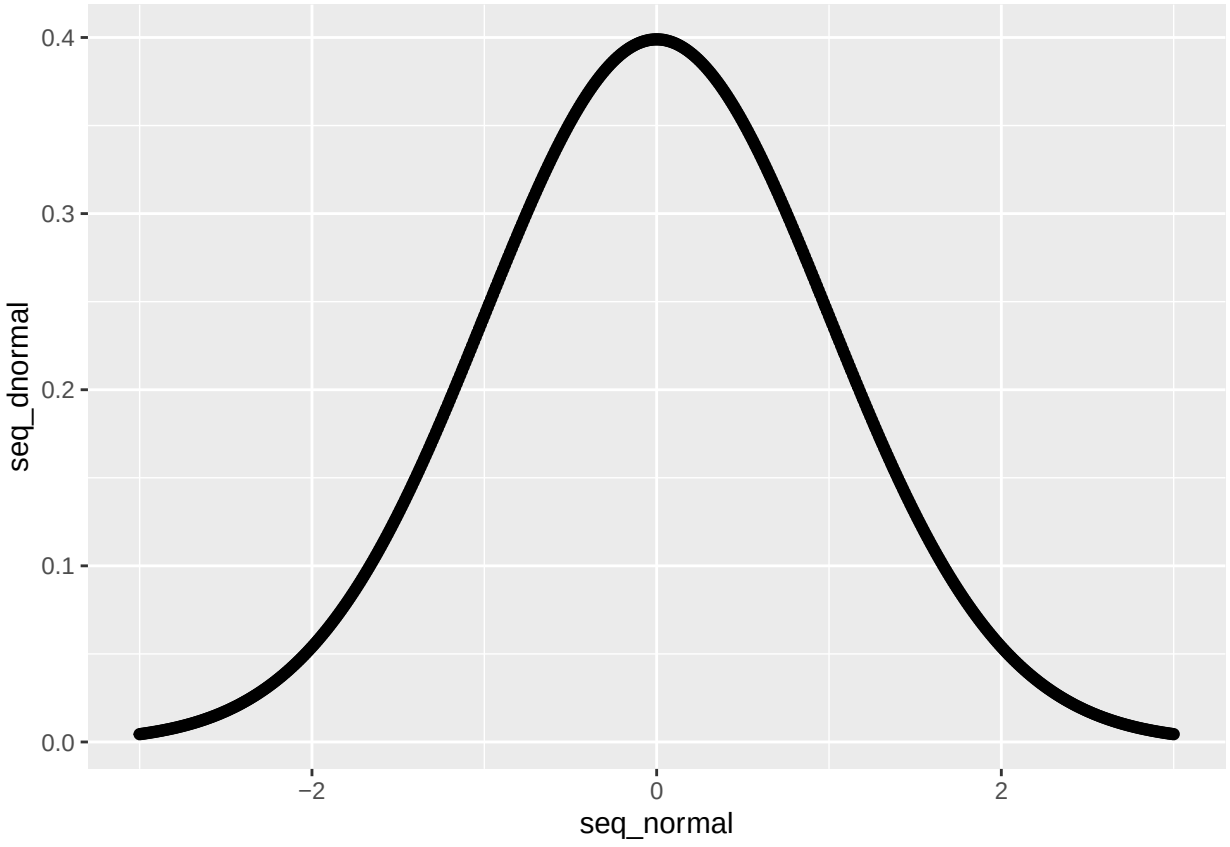
1.6

```r
# Standard normal

seq_normal = seq(-3, 3, .01)

seq_dnormal = dnorm(seq_normal, 0, 1)

qplot(x = seq_normal, y = seq_dnormal)
```

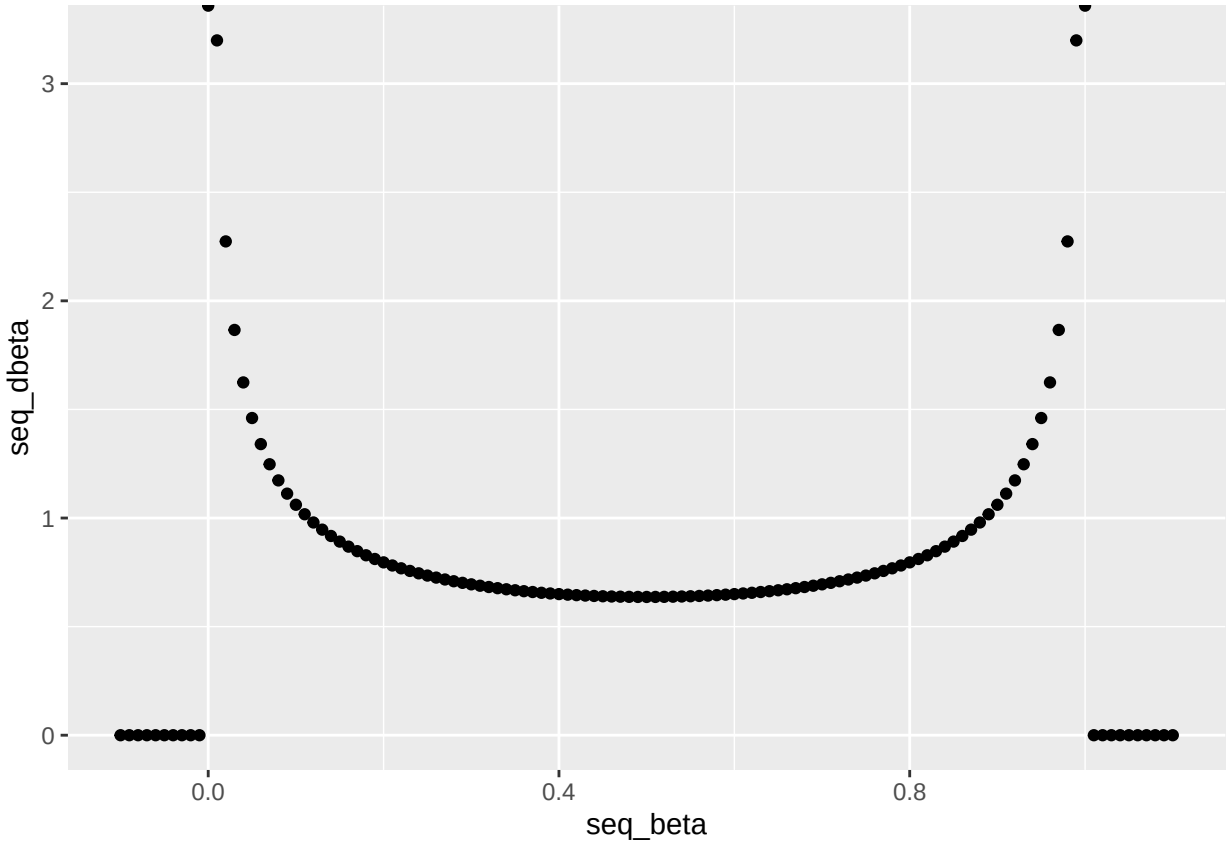


```r
# Beta(.5,.5)

seq_beta = seq(-.1, 1.1, .01)

seq_dbeta = dbeta(seq_beta, .5, .5)

qplot(x = seq_beta, y = seq_dbeta)
```

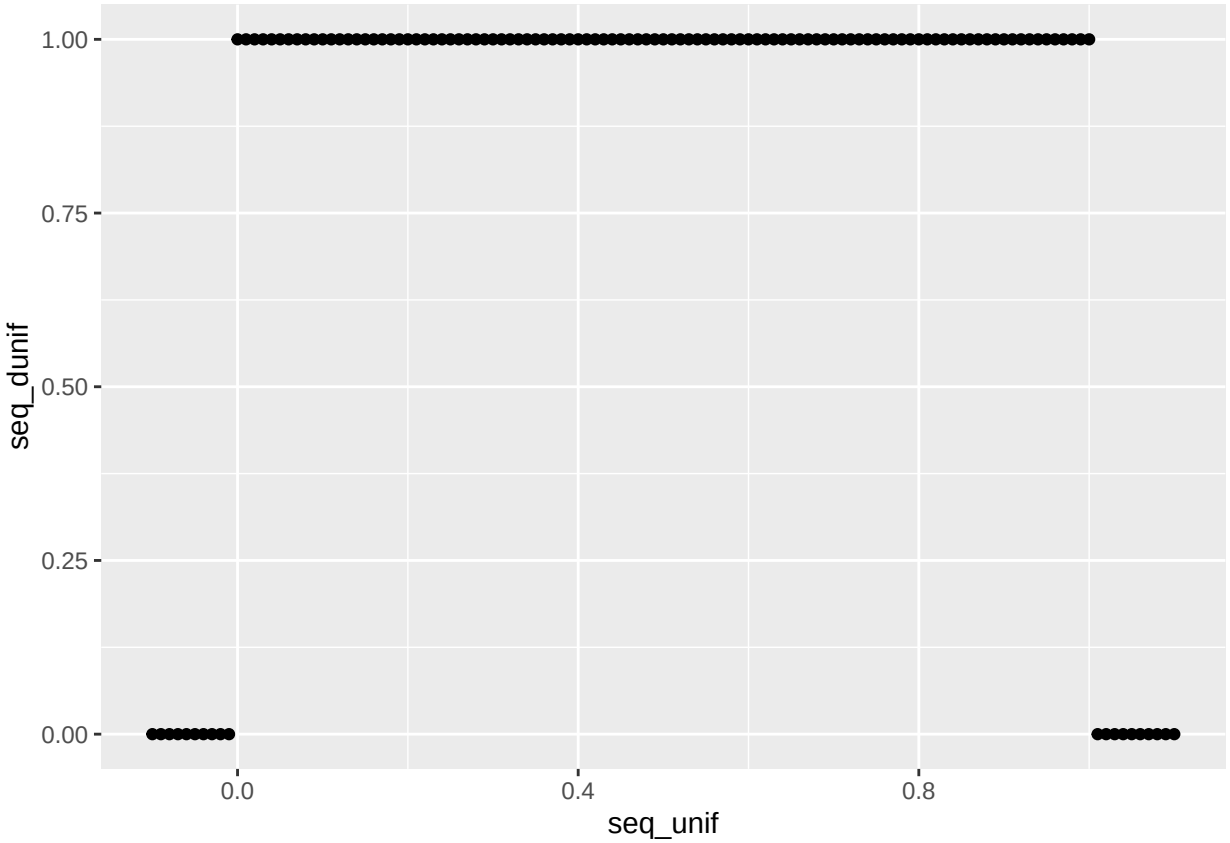



```r
# Uniform (0,1)

seq_unif = seq(-.1, 1.1, .01)

seq_dunif = dunif(seq_unif, 0, 1)

qplot(x = seq_unif, y = seq_dunif)
```

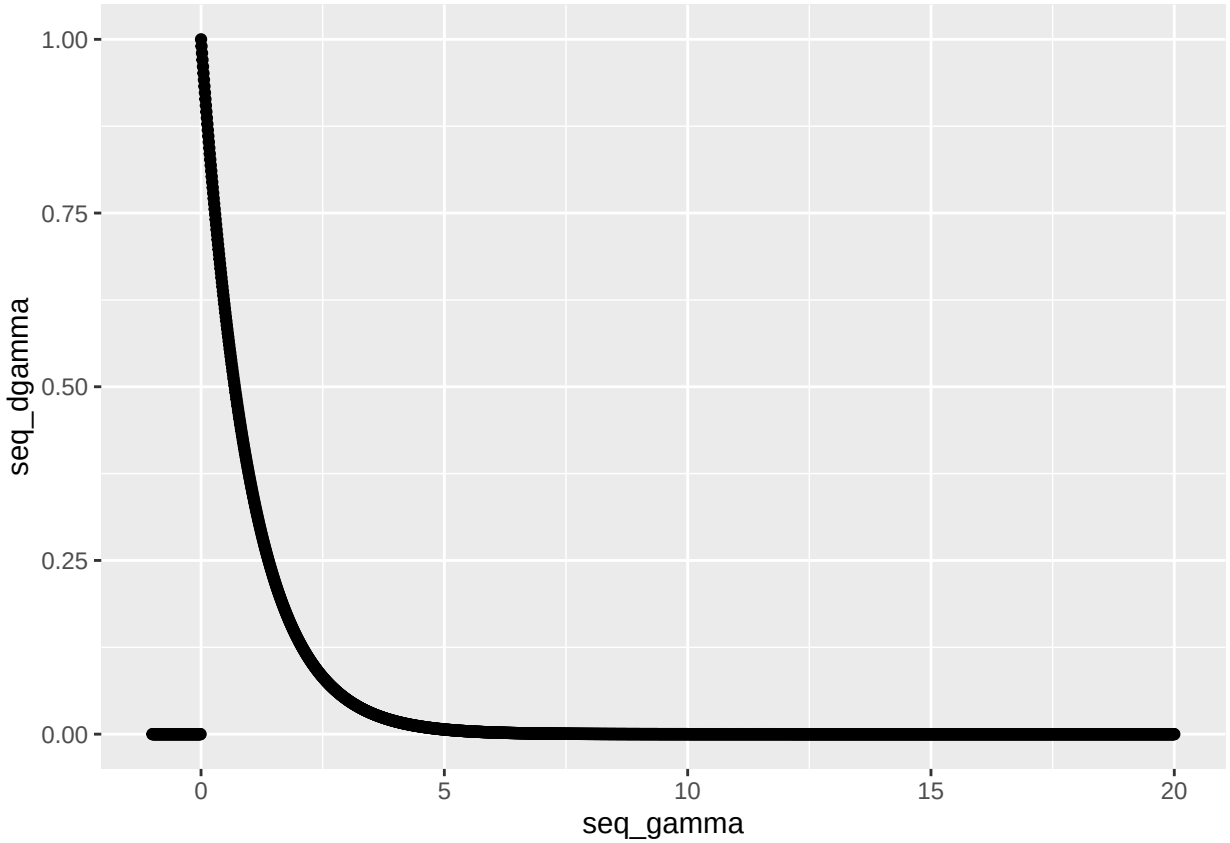



```
# Gamma(1,1)

seq_gamma = seq(-1, 20, .01)

seq_dgamma = dgamma(seq_gamma, 1, 1)

qplot(x = seq_gamma, y = seq_dgamma)
```

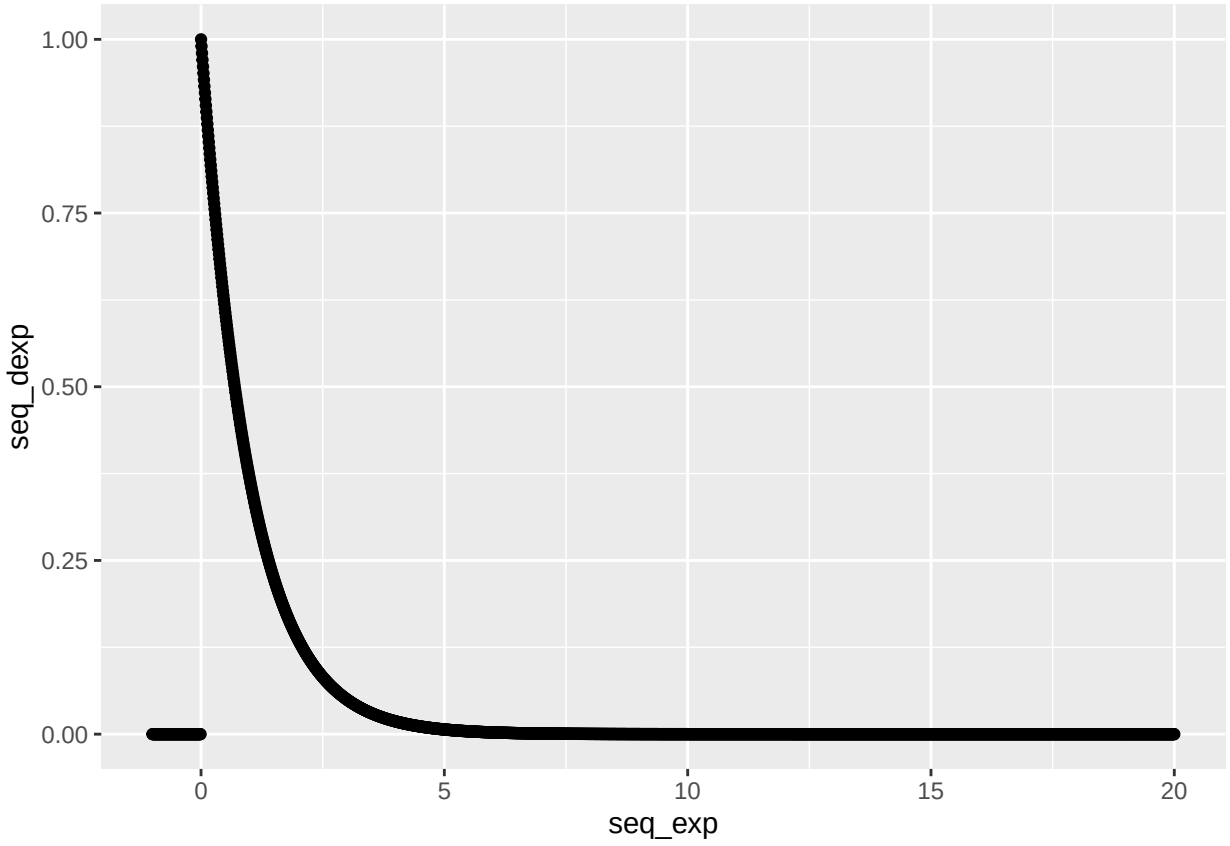



```
# Exponential(1)

seq_exp = seq(-1, 20, .01)

seq_dexp = dexp(seq_exp, 1)

qplot(x = seq_exp, y = seq_dexp)
```



1.7

Note that the mean of a pois(3) is 3, and the variance is also 3.

```
poisson_rv = rpois(100,3)

mean(poisson_rv)
```

```
## [1] 3.1
```

```
var(poisson_rv)
```

```
## [1] 2.414141
```

1.8

```
cel = BSgenome.Celegans.UCSC.ce2

dna_seq = cel$chrM
```