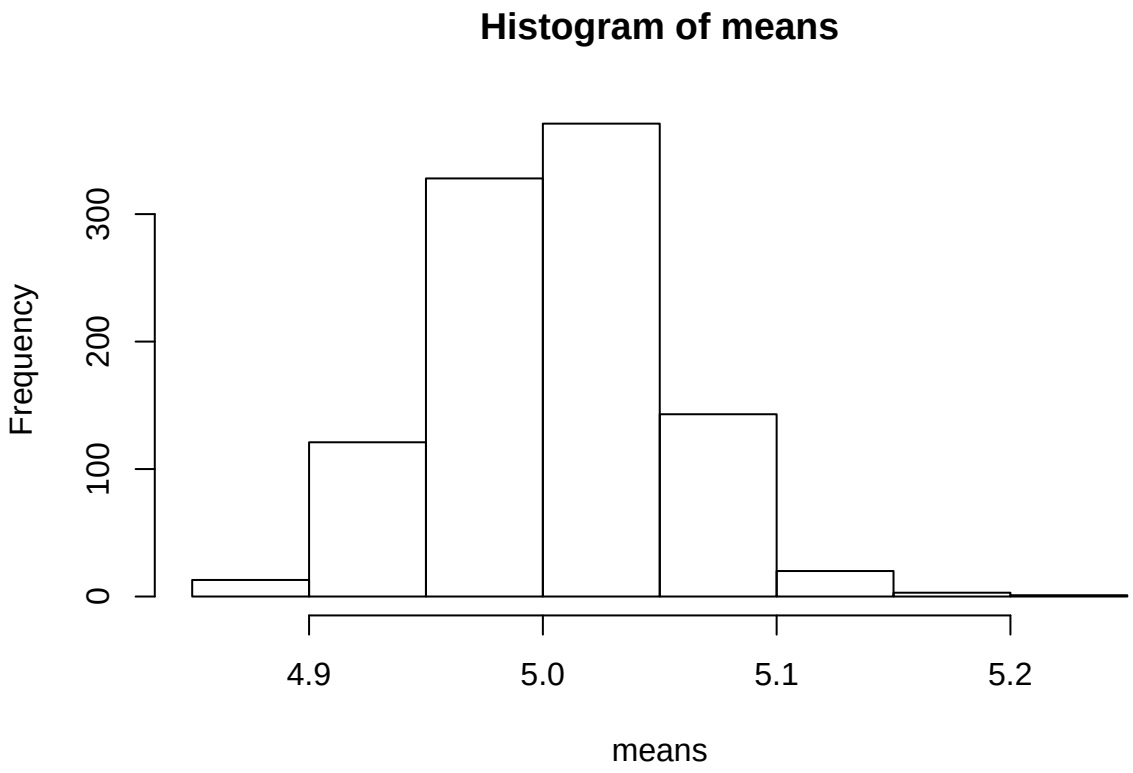
Estimation & EM Algorithm

Bootstrapping

Bootstrapping can be used to simulate a sampling distribution when the sample size is small. To bootstrap, you make draws from the original sampling distribution **with replacement** to create many simulated sampling distributions.

Example: Bootstrapping Sampling Distribution of the Mean

```
set.seed(1)
nboot <- 1000 # Number of bootstraps

## Getting a small sample of data
data(iris)
original_sample <- iris$Sepal.Length[1:50]

means <- rep(0, nboot) # Empty variable to store means

## Bootstrapping
for (b in 1:nboot) {
  bootstr <- sample(original_sample, length(original_sample), rep=T)
  means[b] <- mean(bootstr)
}

hist(means)
```



The mean and standard error of this sampling distribution is:

```
mean(means) # Mean of the sampling disitribution
```

```
## [1] 5.004366
```

```
mean(original_sample) # Mean of the original sample
```

```
## [1] 5.006
```

```
sqrt(var(means)) # Bootstrap standard error of the sample mean
```

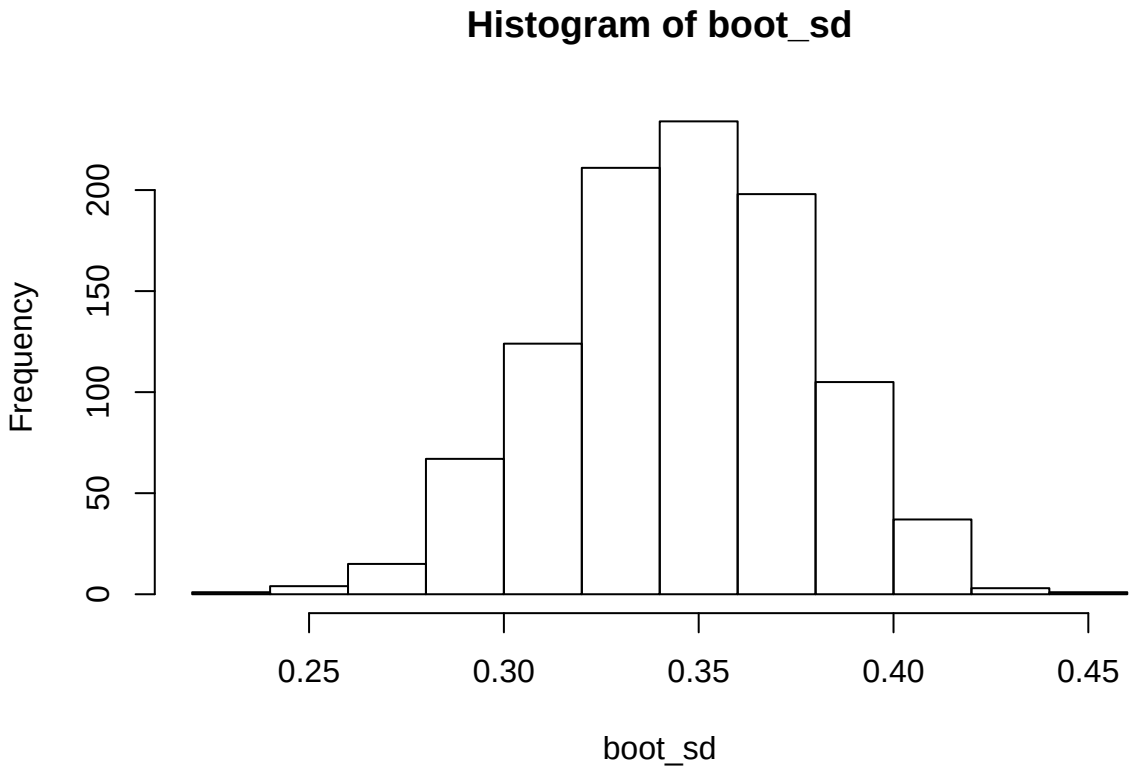
```
## [1] 0.04824751
```

```
sd(original_sample)/sqrt(50) #Theoretical standard error
```

```
## [1] 0.04984957
```

Bootstrap the sampling distribution of standard deviation

```
## Making function to be simpler
boot <-function(obs_samp, fun, nboot) {
  ## obs_samp is the observed sample
  ## fun is a function of data, the statistic whose sampling distribution we care about
  ## nboot is the number of bootstrap samples
  out <- rep(0, nboot)
  n <- length(obs_samp)

  for (b in 1:nboot) {
    out[b] <- fun(sample(obs_samp, n, rep=T))
  }

  return(out)
}

boot_sd <- boot(original_sample, sd, 1000)

hist(boot_sd)
```




```
mean(boot_sd)
```

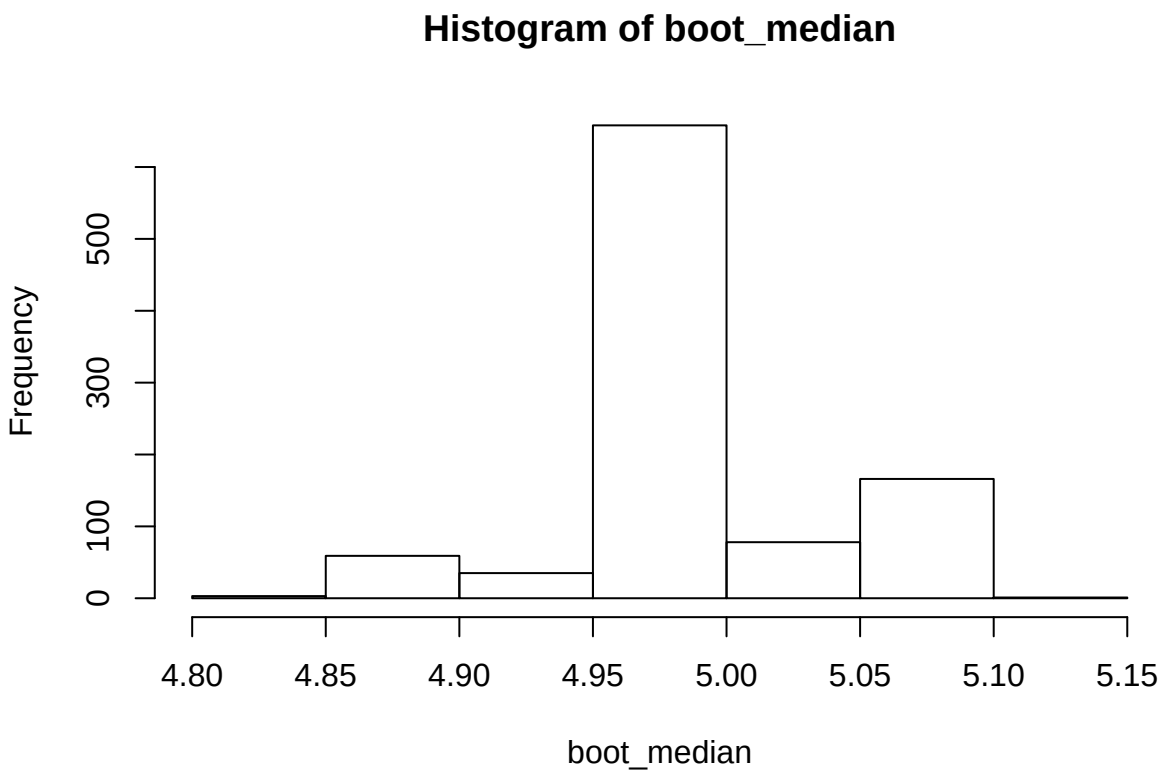
```
## [1] 0.3457202
```

```
sd(boot_sd)
```

```
## [1] 0.03189157
```

Exercise: Adapt the above code to generate a sampling distribution for the median.

```
## Insert Code Here
boot_median <- boot(original_sample, median, nboot)
hist(boot_median)
```




```
mean(boot_median)
```

```
## [1] 5.0125
```

```
sd(boot_median)
```

```
## [1] 0.04986217
```

Unbiased Estimators

Show that $\bar{X}$ is an unbiased estimator for μ.

First, try a small sample size.

```
mean_vec = vector()
for (i in 1:10000){
  mean_vec[i] = mean(rnorm(100, 0, 1))
}
mean(mean_vec)
```

```
## [1] 0.0003907275
```

Now, try a large sample size:

```
mean_vec = vector()
for (i in 1:10000){
  mean_vec[i] = mean(rnorm(10000, 0, 1))
}
mean(mean_vec)
```

```
## [1] 6.186009e-05
```

Both are close to zero, but the bias decreases as the sample size increases. This is because $\bar{X}$ is also consistent.

Exercise: Now show that s^2 is unbiased for σ^2.

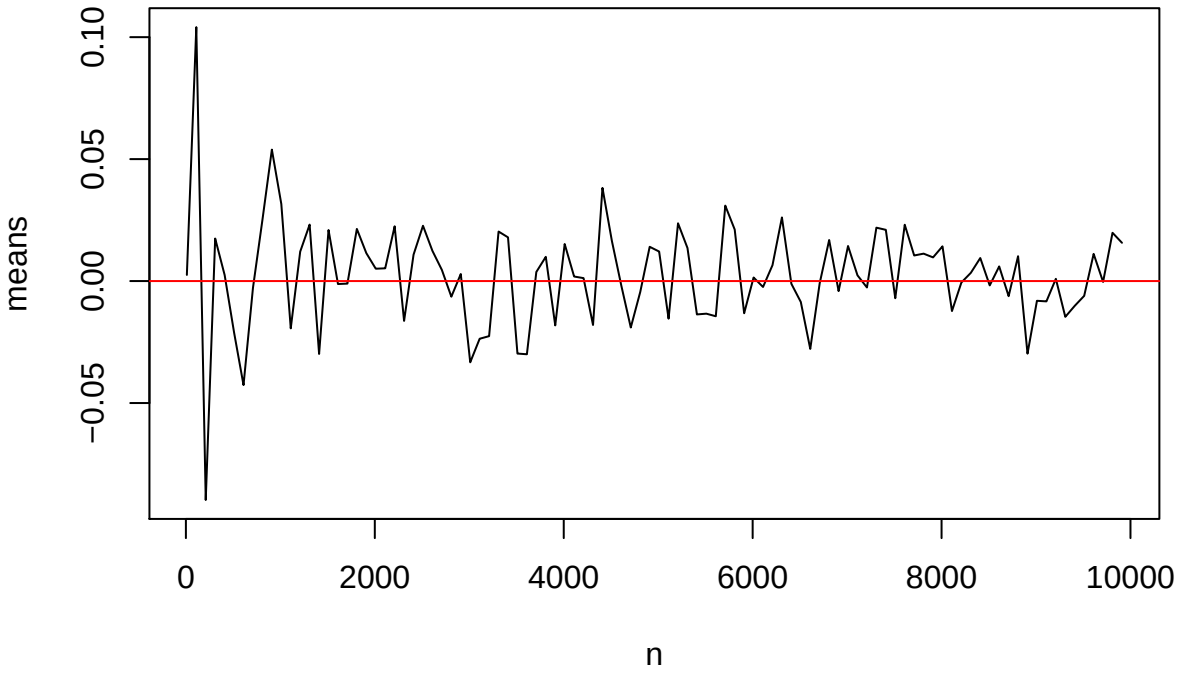
```
## Insert Code Here
for (n in c(100, 1000, 5000)){
  sd_vec <- c()
  for (i in 1:10000){
    sd_vec[i] <- sd(rnorm(n, 0, 1))
  }
  print(mean(sd_vec))
}
```

```
## [1] 0.9967953
## [1] 0.9998247
## [1] 0.9999171
```

Consistency

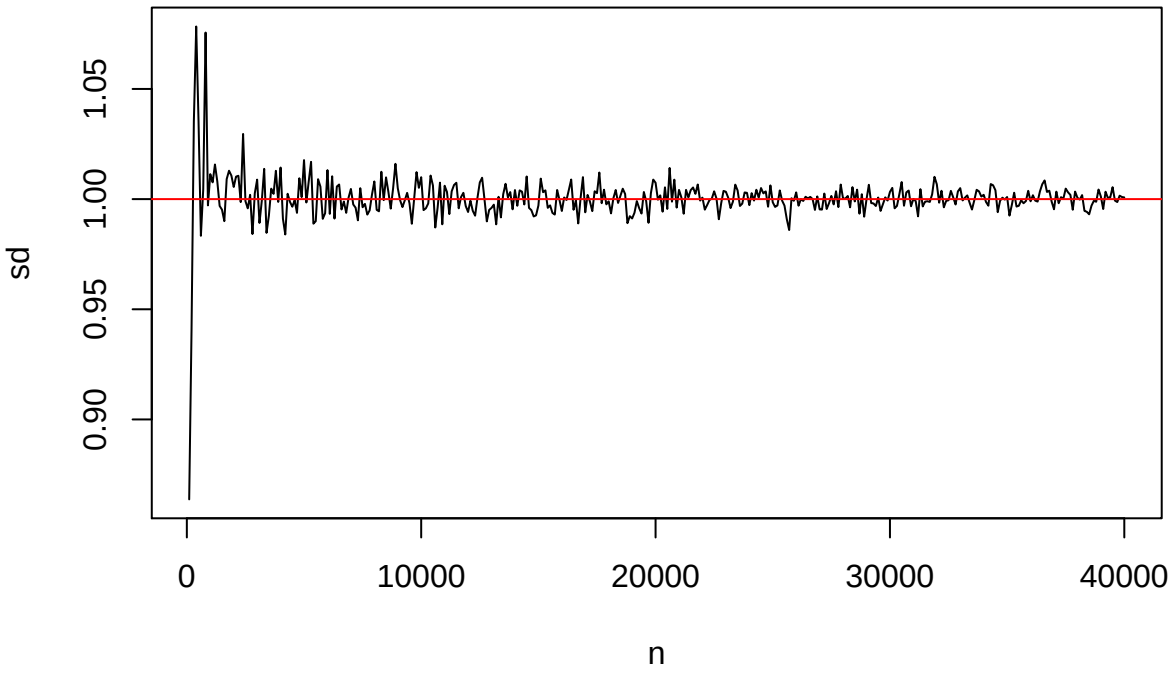
Let's show that the sample mean is consistent for μ.

```
n = seq(10, 10000, by = 100)
means = vector()
for(i in 1:length(n)){
  means[i] = mean(rnorm(n[i], 0, 1))
}

plot(means~n, type = "l")
abline(h=0, col = "red")
```



Exercise: Now, show that s^2 is consistent for σ^2.

```
##Insert Code Here
n = seq(100, 40000 ,by = 100)
sd = c()
for (i in 1:length(n)){
  sd[i] <- sd(rnorm(n[i], 0, 1))
}

plot(sd ~ n, type = "l")
abline(h = 1, col = 2)
```



Mean Squared Error

Mean Squared Error is a measure of closeness of an estimate to the true value, and is calculated the way it sounds. Take the error (e.g. the difference between each point in the sample and the population mean), square it, and find the mean of those squared errors.

$$MSE_{\hat{\theta}} = \frac{\sum_i (\hat{\theta}_i - \bar{\theta})^2}{n}$$

Its also equivalent to the Variance plus Bias Squared

$$MSE_{\hat{\theta}} = V_\theta(\hat{\theta}) + B_\theta(\hat{\theta})^2$$

The smaller the MSE, the better we say the estimator is (the less distance there is between the parameter θ and the estimate $\hat{\theta}$.)

Consider two estimators for the variance:

- Sample variance $S^2 = \sum_{i=1}^{n} \frac{(X_i - \bar{X})^2}{n-1}$

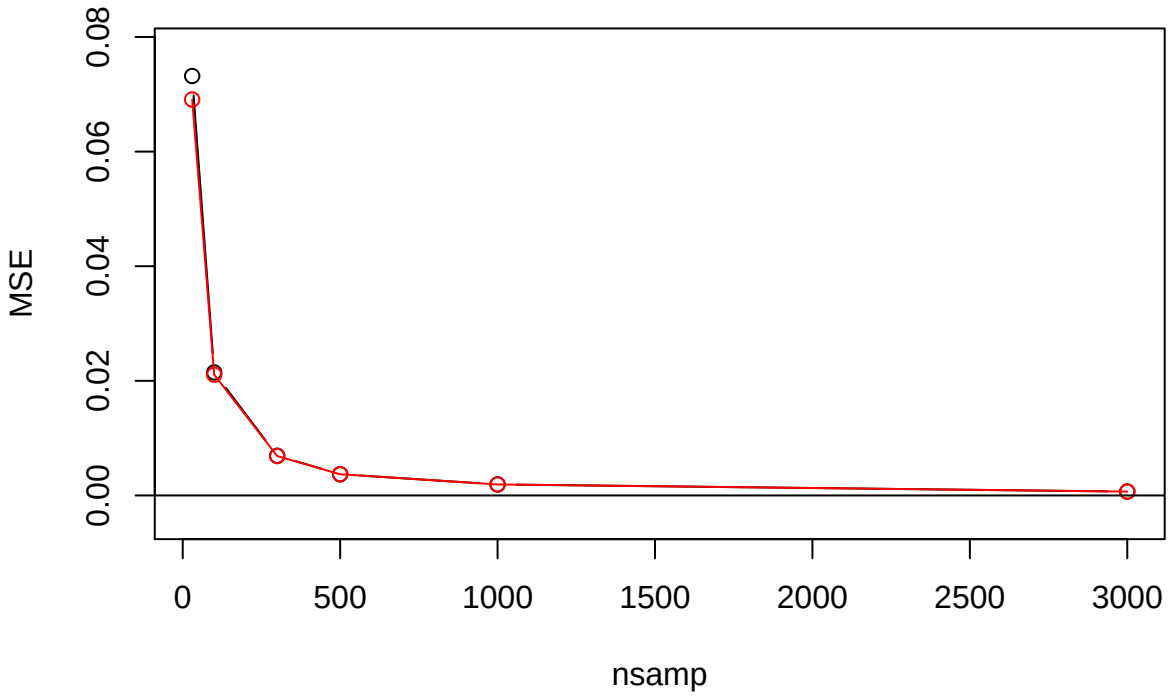
- A similar quantity $\tilde{S}^2 = \sum_{i=1}^{n} \frac{(X_i - \bar{X})^2}{n}$.

Using data simulated from a normal distribution:

Which estimator has smaller Mean Squared Error?

```
nsim <- 1000
nsamp <- c(30, 100, 300, 500, 1000, 3000)

## Creating empty variables
MSE.S1 <- rep(0,length(nsamp))
MSE.S2 <- rep(0,length(nsamp))

## Matrix (columns are different sample sizes, rows are iterations)
S1.mat <- S2.mat <- matrix(0, nsim, length(nsamp))

for (k in 1:length(nsamp)) {
  for (j in 1:nsim) {
    n <- nsamp[k]
    X <- rnorm(n, 1, 1) # Originally mean and sd weren't set - but population
                        # parameter assumed to be 1 for bias & MSE
    S1.mat[j,k] <- sum((X-mean(X))^2)/(n-1) # Estimator 1
    S2.mat[j,k] <- (n-1)/n*S1.mat[j,k]      # Estimator 2
  }

    ## Calculate MSE
    MSE.S1[k] <- mean((S1.mat[,k]-1)^2)
    MSE.S2[k] <- mean((S2.mat[,k]-1)^2)
}

c(MSE.S1, MSE.S2)
```

```
plot(nsamp, MSE.S1, type="b", ylab="MSE",    # type = "both"
     ylim=c(min(MSE.S1, MSE.S2)-0.005,
            max(MSE.S1, MSE.S2)+0.005))
lines(nsamp, MSE.S2, col="red")
points(nsamp, MSE.S2, col="red")
abline(h=0)
```




```
library(pander)
pander(round(rbind(nsamp, MSE.S1, MSE.S2), 4))
```

nsamp	30	100	300	500	1000	3000
MSE.S1	0.0732	0.0215	0.0069	0.0037	0.0019	7e-04
MSE.S2	0.0691	0.0211	0.0069	0.0037	0.0019	7e-04

```
MSE.S1 < MSE.S2
```

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

The second estimator has a smaller Mean Squared Error (despite being biased, as we saw in class)

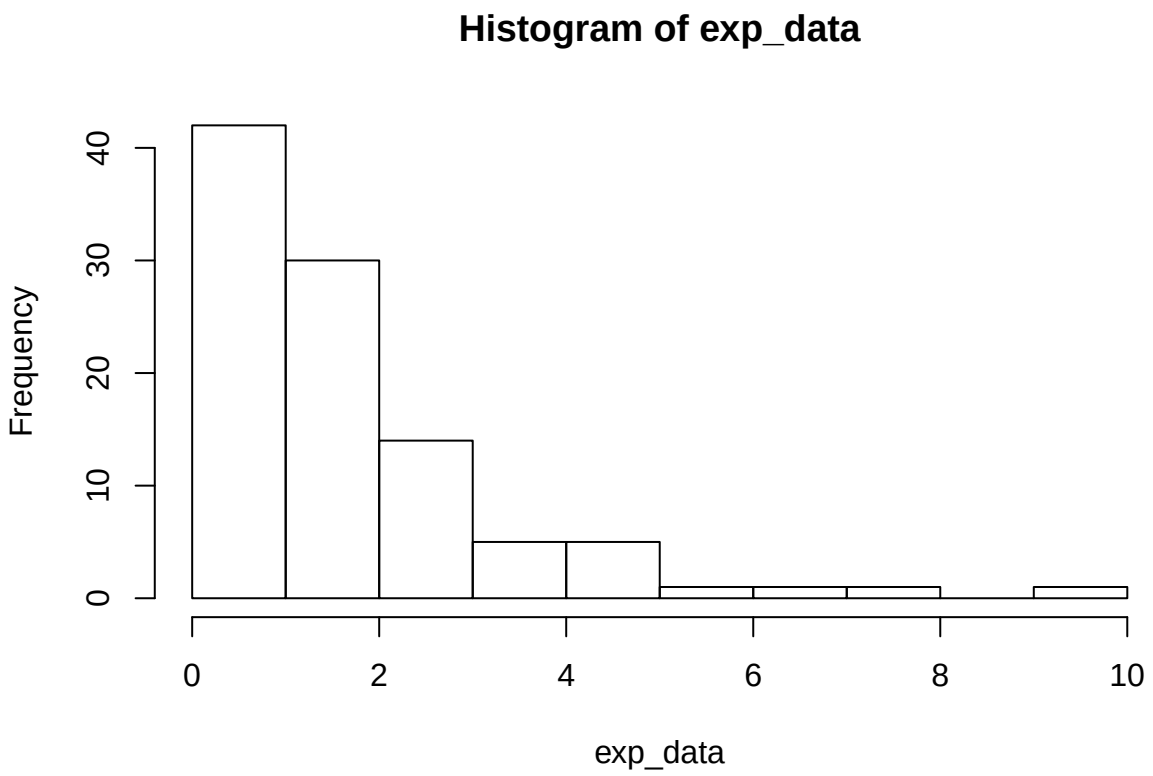
Maximum Likelihood Estimates

To show an example with the exponential distribution, we'll first simulate some data from the exponential distribution. We'll set λ (in R referred to as `rate`) to 0.5, though later we will be estimating this value as if we didn't know it.

```
set.seed(456) #For consistency

exp_data <- rexp(100, 0.5) #Generate 100 observations from exp distribution with lambda=0.5
summary(exp_data)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## 0.04196 0.60960 1.26659 1.68070 2.06758 9.40917
```

```
hist(exp_data)
```



The log-likelihood function of the exponential distribution is

$$l(\lambda) = \sum_{i=1}^{n} (\log \lambda - \lambda x_i)$$

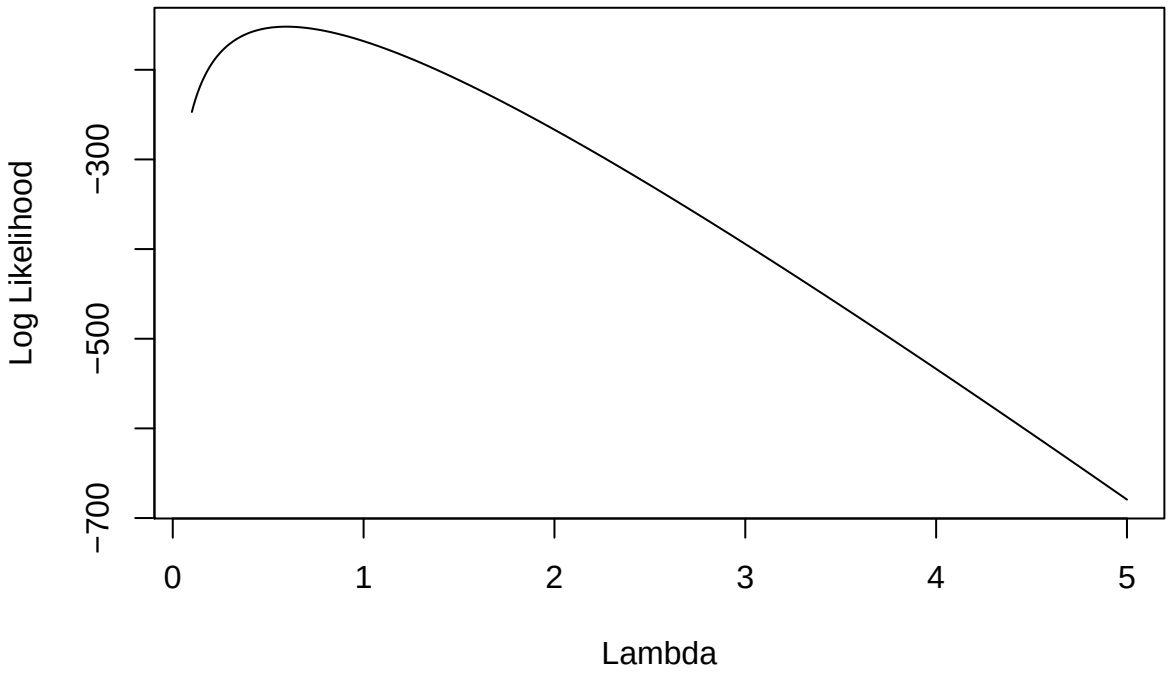
Let's turn this into a function in R:

```
logLikExp <- function(x_vec, lambda) {
  log_lik <- sum(log(lambda) - lambda * x_vec)
  return(log_lik)
}

logLikExp(x_vec = exp_data, lambda=1)
```

```
## [1] -168.0704
```

Now with that function ready, we can decide on what range of lambdas we want to try. We know from our extensive knowledge of distributions (or from wikipedia) that the exponential distribution is defined so that $\lambda > 0$, so we can start off looking at a range of 0.1 to 5, in increments of 0.01.

```
lambdas <- seq(from = 0.1, to = 5, by = 0.01)

logliks <- sapply(lambdas, function(j) logLikExp(exp_data, j))
plot(lambdas, logliks, type='l', xlab = "Lambda",  ylab = "Log Likelihood")
```



The plot shows the change in log likelihoods (on the Y-axis) for values of λ (on the x-axis), and we can see there is a maximum value. If there wasn't an obvious maximum here, we would probably want to expand the range of lambdas.

```
lambdas[which.max(logliks)] # Which lambda gives us the maximum?
```

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

Not exactly the 0.5 we used to generate the data, but a close estimate. We can also try the formula that we could derive ($\hat{\lambda} = \frac{n}{\sum_{i=1}^{n} x_i}$) with this data to see how consistent it is with the computational solution. Remember that for exponential distribution, $\mu = \frac{1}{\lambda}$

```
1 / mean(exp_data)
```

```
## [1] 0.5949887
```

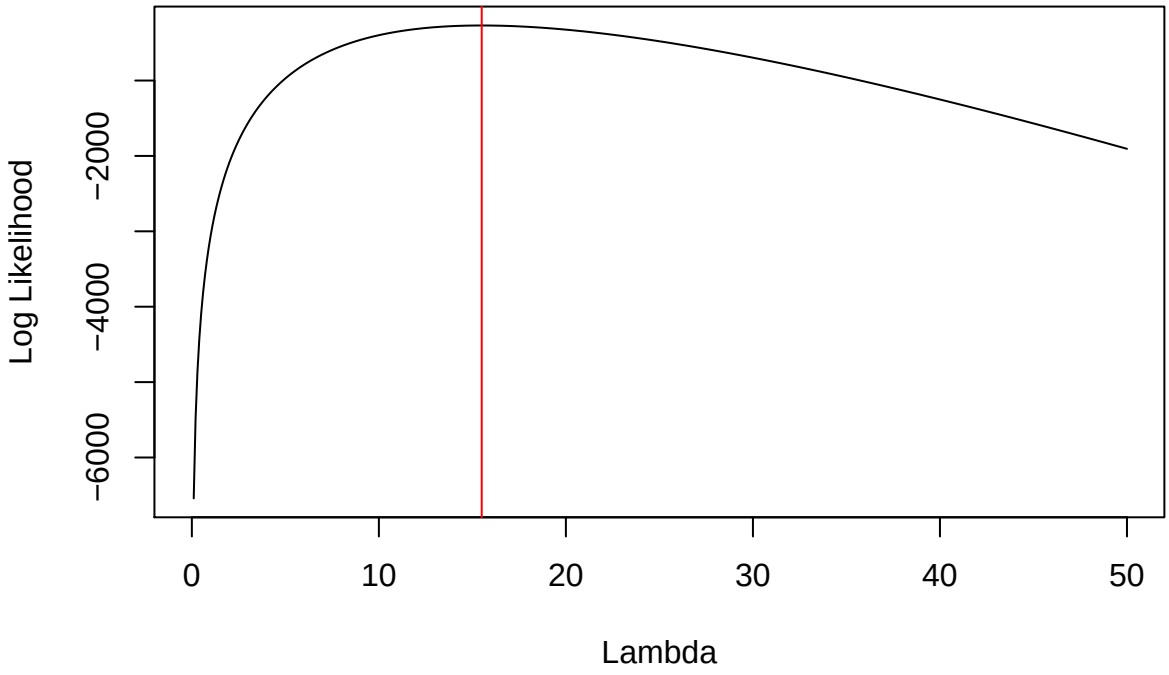
Exercise: Poisson Distribution

Generate 100 data points from a Poisson($\lambda = 15$) distrbution. Write a function for the log-likehood of the Poisson distribution. Create a vector that is a sequence from 0.1 to 50 with increments of 0.1 to function as a range of parameters to search. Calculate the log-likelihood for each of your possible lambda values in the sequence you just generated. Plot the log-likelihood across your sequence. Use the *which.max* function to find the maximum value of the log-likelihood (your MLE). The log-likelihood for Possion distribution

$$\sum_{i=1}^{n} log(\lambda^{x_i} * e^{-\frac{\lambda}{x_i!}})$$

```
## Insert Code Here
set.seed(64) #For consistency

poi_data <- rpois(100, 15) #Generate 100 observations from exp distribution with lambda=0.5
poi <- function(x_vec_poi, lambda){
  logLike_poi <- sum(log(lambda^x_vec_poi * exp(-lambda)/factorial(x_vec_poi)))
```

```
  return(logLike_poi)
}

lambdas_poi <- seq(0, 50, 0.1)
logLikes_poi <- sapply(lambdas_poi, function(j) poi(poi_data, j))
plot(lambdas_poi, logLikes_poi, type='l', xlab = "Lambda",  ylab = "Log Likelihood")
abline(v = lambdas_poi[which.max(logLikes_poi)], col = 2)
```



R Functions to do the same thing

Now that you've done it by hand, you can see the shortcuts in the form of built-in R functions. Most allow for alternate methods as well, that may be much more efficient at maximizing the log-likelihood function.

`dexp` (and other builtin distribution functions)

Although its a good exercise to write out the formula for log likelihood, we can more easily use the built-in denisty functions, such as `dexp`, `dpois`, `dnorm`, etc. To get the log likelihood we just take the sum of the log of the densities for our values of x.

```
## (This function is also being formatted for use in the following sections)
ll.exp <- function(lambda, x) sum(dexp(x, lambda, log = T))
```

Optimize

This function lets you specify an `interval`, and tolerance `tol` which indicates how accurate to optimize. It optimizes the function over the first variable that is not otherwise specified (here we specify `x_vec`, so it moves on to `lambda`).

```
optimize(f = ll.exp, interval = c(0, 50),  tol = 0.00001, maximum = T, x = exp_data)
```

```
## $maximum
## [1] 0.5949883
##
## $objective
## [1] -151.9213
```

The maximum is the *value of the parameter* at the maximum log likelihood, whereas "objective" is the log likelihood.

NLM

This function uses the "Nonlinear Minimizer" to optimize the function based on a Newton-type method. This is designed that you can give a "best" guess at the value of the parameter for the algorithm to use as a starting point (this is the `p` argument). Based on this formula to calculate where the first derivative is 0.

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

One drawback is it can only **minimize** a function, so we have to do a quick rewrite of our function to make its minimum actually our maximum.

```
ll.exp.neg <- function(lambda, x)  -sum(dexp(x, lambda, log = T))

nlm(f = ll.exp.neg, p = 0.8, x = exp_data)
```

```
## $minimum
## [1] 151.9213
##
## $estimate
## [1] 0.5949879
##
## $gradient
## [1] -8.839152e-05
##
## $code
## [1] 1
##
## $iterations
## [1] 4
```

```
nlm(f = ll.exp.neg, p = 50, x = exp_data)
```

```
## $minimum
## [1] 151.9213
##
## $estimate
## [1] 0.5949882
##
## $gradient
## [1] 2.756906e-06
##
## $code
## [1] 1
##
## $iterations
## [1] 8
```

We can see that the difference between the "good" and the "bad" guesses is only in how many iterations were required to arrive at the answer.

MLEs for multiple parameters

For a distribution with multiple parameters, such as the Normal distribution, we would have to do a grid search.

First, let's generate some Normal data:

```
norm_data <- rnorm(100, mean=5, sd=3)
```

Next, we program in the log-likelihood of the normal distribution. Note that here we are working with σ^2 while above the *rnorm* function takes parameter σ.

$$-0.5 * [n * log(2\pi) + n * log(\sigma^2) + \sum_{i=1}^{n} \frac{(x_i - \mu)^2}{\sigma^2}]$$

```
logLikNorm <- function(x_vec, mu, sigsq) {
  n <- length(x_vec)
  log_lik <- -0.5*(n * log(2*pi) + n * log(sigsq) + sum(((x_vec - mu)^2)/sigsq))
  return(log_lik)
}
```

We'll use the `expand.grid` function to get all combinations of the two parameter ranges.

```
mu_pars <- seq(-10, 10, by = 0.1)
sigsq_pars <- seq(0, 10, by = 0.1)

par_grid <- expand.grid(mu = mu_pars, sigsq = sigsq_pars)

head(par_grid)
```

```
##      mu sigsq
## 1 -10.0     0
## 2  -9.9     0
## 3  -9.8     0
## 4  -9.7     0
## 5  -9.6     0
## 6  -9.5     0
```

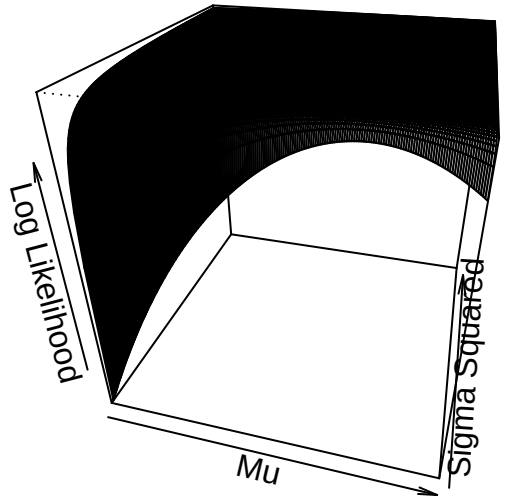
```
tail(par_grid)
```

```
##         mu sigsq
## 20296  9.5    10
## 20297  9.6    10
## 20298  9.7    10
## 20299  9.8    10
## 20300  9.9    10
## 20301 10.0    10
```

Now we can use the `apply` function to find the maximum log-likelihood.

```
lik1 <- apply(par_grid, 1, function(pars) logLikNorm(norm_data, pars[1], pars[2]))
  ## ?apply: margin: 1 indicates rows, 2 indicates columns
lik_mat1 <- matrix(lik1, nrow = length(mu_pars), ncol = length(sigsq_pars)) # Rows are Mu, columns are
```

```
persp(x = mu_pars, y = sigsq_pars, z = lik_mat1, phi = 30, theta = 15,
      xlab = "Mu", ylab = "Sigma Squared", zlab = "Log Likelihood")
```




```
par_grid[which.max(lik1),]
```

```
##        mu sigsq
## 20252 5.1    10
```

These values are close to the true values of $\mu = 5$ and $\sigma^2 = 9$. Note that this is only based on 100 data points. If we increased the sample size, these values would get closer to the truth since these estimators are consistent for their true parameter values.

EM Algorithm

This section is cribbed from http://tinyheero.github.io/2016/01/03/gmm-em.html

Parameter Estimation in the "Complete Data" Scenario

Let's start with the scenario where you are given the following 1-dimensional data with the colors representing the "source" they are from:

```
library("ggplot2")
library("dplyr")
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

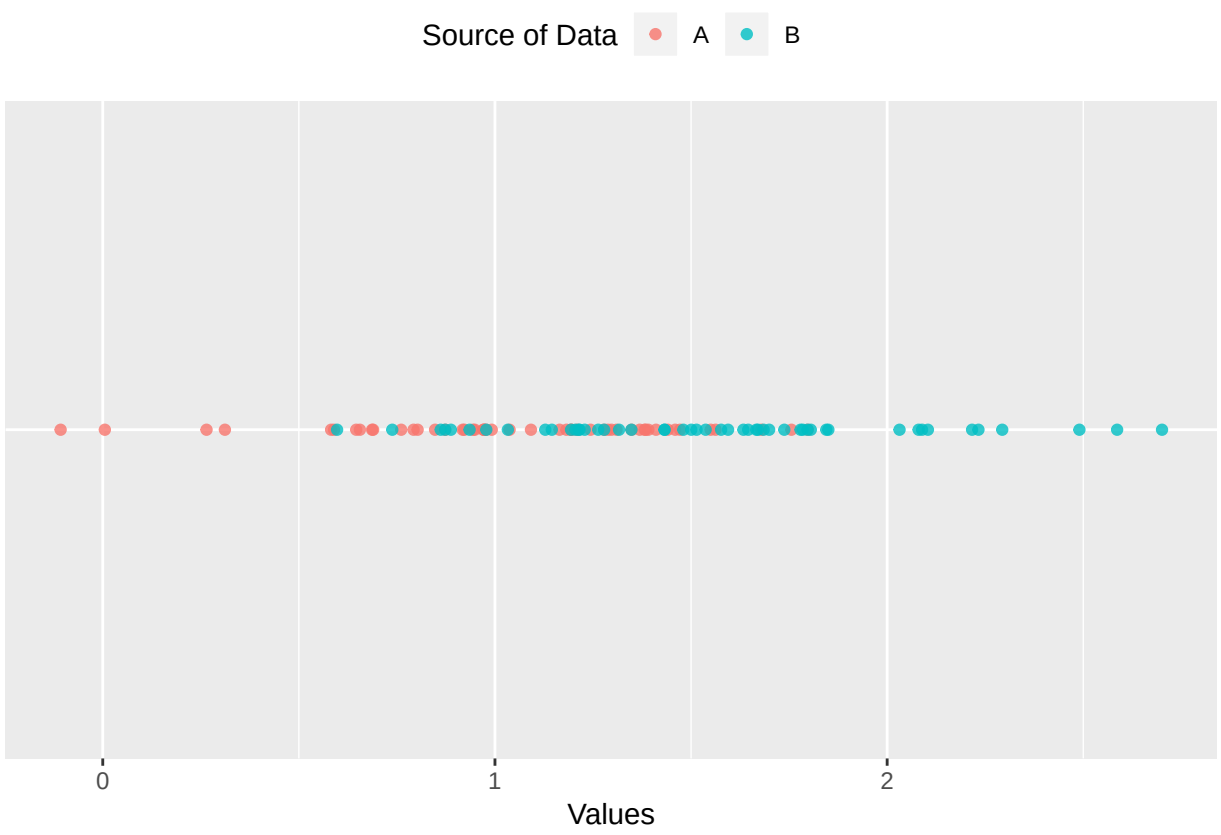
```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library("reshape2")

options(scipen = 999)
```

```
set.seed(1)

comp1.vals <- data_frame(comp = "A",
                         vals = rnorm(50, mean = 1, sd = 0.5))    #head(comp1.vals)
```

```
## Warning: `data_frame()` is deprecated, use `tibble()`.
## This warning is displayed once per session.
```

```
comp2.vals <- data_frame(comp = "B",
                         vals = rnorm(50, mean = 1.5, sd = 0.5))

vals.df <- bind_rows(comp1.vals, comp2.vals)   # #rows=#rowsA + #rowsB

vals.df %>%
  ggplot(aes(x = vals, y = "A", color = factor(comp))) +
  geom_point(alpha = 0.8) +
  scale_color_discrete(name = "Source of Data") +
  xlab("Values") +
  theme(axis.ticks.y = element_blank(),
        axis.text.y = element_blank(),
        axis.title.y = element_blank(),
        legend.position = "top")
```



Let's say you believe that the data sources are actually a Normal distribution. Then with these values and knowledge of their data source, we can estimate the parameter values using either MLE, or an unbiased estimate for σ^2.

```
vals.df %>%
  group_by(comp) %>%
```

```
  summarize(mean_vals = mean(vals),
            sd_vals = sd(vals))
```

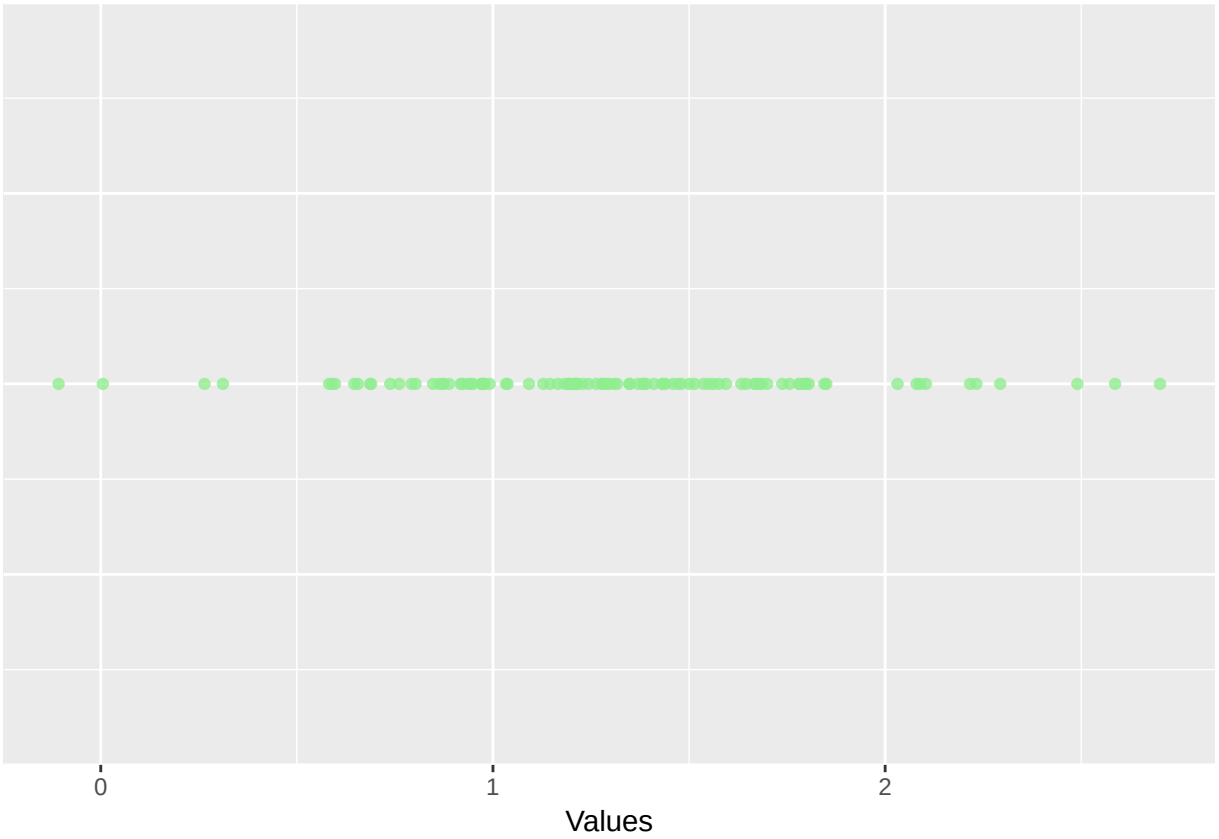
```
## # A tibble: 2 x 3
##   comp  mean_vals sd_vals
##   <chr>     <dbl>   <dbl>
## 1 A          1.05   0.416
## 2 B          1.56   0.484
```

And we're done.

Parameter Estimation in the Incomplete Data Scenario

Now let us consider the following scenario where you have the same data, but you don't know the source now.

```
vals.df %>%
  ggplot(aes(x = vals, y = 0)) +
  geom_point(alpha = 0.8, col = "lightgreen") +
  xlab("Values") +
  theme(axis.ticks.y = element_blank(),
        axis.text.y = element_blank(),
        axis.title.y = element_blank())
```



Without the labels on the data (often called latent variables) indicating their source, we are hopeless in applying MLE. But if we somehow find a way to "complete" the data (i.e. find the labels for each data point), then we can go back to using MLE. One way around this could be to:

1. Set some initial parameter estimates on your gaussians.

2. Assign (label) the data to one of the gaussians based on which one most likely generated the data.
3. Treat the labels as being correct and then use MLE to re-estimate the parameters for the different gaussians.

Then repeat steps 2 and 3 until there is convergence.

Fitting a Gaussian Mixture Model (GMM) using Expectation Maximization

The EM algorithm consists of three major steps:

1. Initialization
2. Expectation (E - step)
3. Maximization (M - step)

Steps 2 and 3 are repeated until convergence. We will cover each of these steps and how convergence is reached below. But first we must understand how to mathematically represent a GMM:

$$P(X|\mu, \sigma, \alpha) = \sum_{k=1}^{K} \alpha_k \mathcal{N}(X|\mu_k, \sigma_k^2)$$

+ X = Dataset of n elements $(x_1, \ldots, x_n)$

- α_k = Mixing weight of k^{th} component. $\sum_{k=1}^{K} \alpha_k = 1$. i.e. α_k is a set of simplex.
- $\mathcal{N}(x|\mu_k, \sigma_k)$ = Normal PDF for component k defined by parameters μ_k and σ_k

So for a two component GMM, we would mathematically represent this as:

$$P(X|\mu, \sigma, \alpha) = \alpha_1 \mathcal{N}(X|\mu_1, \sigma_1^2) + \alpha_2 \mathcal{N}(X|\mu_2, \sigma_2^2)$$

Initialization: Determining the Initial GMM Parameters

When it comes to initialization of a GMM, we are asking the fundamental question of what model parameters do we first assign? This can be done in different ways, but for GMMs it's very common to first run **k-means** on your data to get some hard-labels on the data. With these hard-labels, we can use MLE to estimate the component parameters for our initialization (remember MLE works in the "Complete Data" scenario):

- $\mu_k = \frac{\sum_i^{N_k} x_{i,k}}{N_k}$
- $\sigma_k^2 = \frac{\sum_i^{N_k} (x_{i,k} - \mu_k)^2}{N_k}$
- $\alpha_k = \frac{N_k}{N}$

Where N_k indicates the number of datapoints in the kth component. Let's try that here:

```
wait <- faithful$waiting

wait.kmeans <- kmeans(wait, 2)
wait.kmeans.cluster <- wait.kmeans$cluster

wait.df <- data_frame(x = wait, cluster = wait.kmeans.cluster)

wait.df %>%
  mutate(num = row_number()) %>%
  ggplot(aes(y = num, x = x, color = factor(cluster))) +
  geom_point() +
```

```
  xlab("Values") +
  ylab("Data Point Number") +
  scale_color_discrete(name = "Cluster") +
  ggtitle("K-means Clustering")
```

Since we specified 2 clusters, k-means nicely splits the data into 2 clusters with means and standard deviation as follows:

```
wait.summary.df <- wait.df %>%
  group_by(cluster) %>%
  summarize(mu = mean(x), variance = var(x), std = sd(x), size = n())

wait.summary.df %>%
  select(cluster, mu, variance, std)
```

```
## # A tibble: 2 x 4
##   cluster    mu variance   std
##     <int> <dbl>    <dbl> <dbl>
## 1       1  54.8     34.8  5.90
## 2       2  80.3     31.7  5.63
```

We can also generate the initial mixing weights as follows:

```
wait.summary.df <- wait.summary.df %>%
  mutate(alpha = size / sum(size))

wait.summary.df %>%
  select(cluster, size, alpha)
```

```
## # A tibble: 2 x 3
##   cluster  size alpha
##     <int> <int> <dbl>
## 1       1   100 0.368
## 2       2   172 0.632
```

Expectation: Calculating the "Soft Labels" of Each Data Point (E-Step)

Now that we have the initial parameters of our GMM, we now have to determine what is the probability (soft label; responsibility) that the data point (x_i) belongs to component (k_j)? This is considered the expectation step (E-step) of MLE where we are calculating the "expectation values" of the soft labels for each data point.

Mathematically, the question can be posed like this $P(x_i \in k_j|x_i)$. How do we actually solve this equation? To help us, we can apply **Bayes' rule** here:

$$P(x_i \in k_j|x_i) = \frac{P(x_i|x_i \in k_j)P(k_j)}{P(x_i)}$$

The parts of this equation are related to the GMM equation above as follows:

- $P(x_i|x_i \in k_j) = \mathcal{N}(x_i|\mu_{k_j}, \sigma^2_{k_j})$
- $P(k_j) = \alpha_{k_j}$
- $P(x_i) = \sum_{k=1}^{K} \alpha_k \mathcal{N}(x_i|\mu_k, \sigma^2_k)$

What we are interested in is $P(x_i \in k_j|x_i)$ which is called the posterior probability. Knowing these equations, we can easily calculate this. For instance, what is the posterior probability of x = 66 belong to the first component? We can first calculate the top part of the equation like this in R:

```
comp1.prod <-
  dnorm(66, wait.summary.df$mu[1], wait.summary.df$std[1]) *
  wait.summary.df$alpha[1]
```

To calculate the bottom part of the equation, we actually need to calculate this value for both components and sum them up:

```
comp2.prod <-
  dnorm(66, wait.summary.df$mu[2], wait.summary.df$std[2]) *
  wait.summary.df$alpha[2]

normalizer <- comp1.prod + comp2.prod
```

Now that we have all the components of the equation, let's plug and solve this:

```
comp1.prod / normalizer
```

```
## [1] 0.6926023
```

We can easily calculate this for every data point as follows:

```
comp1.prod <- dnorm(x = wait, mean = wait.summary.df$mu[1],
                    sd = wait.summary.df$std[1]) * wait.summary.df$alpha[1]

comp2.prod <- dnorm(x = wait, mean = wait.summary.df$mu[2],
                    sd = wait.summary.df$std[2]) * wait.summary.df$alpha[2]

normalizer <- comp1.prod + comp2.prod
```

```r
comp1.post <- comp1.prod / normalizer
comp2.post <- comp2.prod / normalizer
```

Maximization: Re-estimate the Component Parameters (M-step)

Now that we have posterior probabilites (i.e. soft labels), we can re-estimate our component parameters. We simply have to make a little adjustment to the MLE equations that we specified early. Previously, we simply assume each observation is 100% belonged to one certain component, thus, each observation was regarded as 1; now that we have the posterior probability, the N_k is replaced with the posterior probability $P(x_i \in k_j|x_i)$ in each equation.

- $\mu_k = \frac{\sum_i^N P(x_i \in k_j|x_i)x_i}{\sum_i^N P(x_i \in k_j|x_i)}$
- $\sigma_k^2 = \frac{\sum_i^N P(x_i \in k_j|x_i)(x_i - \mu_k)^2}{\sum_i^N P(x_i \in k_j|x_i)}$
- $\alpha_k = \frac{\sum_i^N P(x_i \in k_j|x_i)}{N}$

With these equations, we can now plug in our values and calculate the components parameters using our example from above:

```r
comp1.n <- sum(comp1.post)
comp2.n <- sum(comp2.post)

comp1.mu <- 1/comp1.n * sum(comp1.post * wait)
comp2.mu <- 1/comp2.n * sum(comp2.post * wait)

comp1.var <- sum(comp1.post * (wait - comp1.mu)^2) * 1/comp1.n
comp2.var <- sum(comp2.post * (wait - comp2.mu)^2) * 1/comp2.n

comp1.alpha <- comp1.n / length(wait)
comp2.alpha <- comp2.n / length(wait)

comp.params.df <- data.frame(comp = c("comp1", "comp2"),
                             comp.mu = c(comp1.mu, comp2.mu),
                             comp.var = c(comp1.var, comp2.var),
                             comp.alpha = c(comp1.alpha, comp2.alpha))

comp.params.df
```

```
##    comp  comp.mu comp.var comp.alpha
## 1 comp1 54.74109 35.60339  0.3649454
## 2 comp2 80.18137 33.30966  0.6350546
```

Checking for Convergence

As mentioned above, we repeat the expectation and maximization step until we reach "convergence". But what exactly is convergence? The concept of convergence means that we have a change that is minimal enough for us to consider it to neligible and stop running EM. So the question becomes what is the change we are measuring? Answer: The likelihood (or log likelihood)!

$$P(X|\mu,\sigma,\alpha) = \sum_{k=1}^{K} \alpha_k \mathcal{N}(X|\mu_k,\sigma_k^2)$$

As we know, the likelihood is essentially the fit of your model. Really what we are asking in layman terms is given these model parameters (μ,σ,α), what is the probability that our data X was generated by them. A slight modification of this is the log likelihood which equates to:

$$\ln P(X|\mu,\sigma,\alpha) = \sum_{n=1}^{N} \ln \sum_{k=1}^{K} \alpha_k \mathcal{N}(x_n|\mu_k,\sigma_k^2)$$

The reason why we do this is because if we simply calculate the likelihood we would end up dealing with very small values which can be problematic. So we take the natural logarithm of the likelihood to circumvent this.

For instance, the log likelihood of our first EM step:

```
sum.of.comps <- comp1.prod + comp2.prod
sum.of.comps.ln <- log(sum.of.comps, base = exp(1))
sum(sum.of.comps.ln)
```

```
## [1] -1034.246
```

So to test for convergency, we can calculate the log likelihood at the end of each EM step (i.e. model fit with these parameters) and then test whether it has changed "significantly" (defined by the user) from the last EM step. If it has, then we repeat another step of EM. If not, then we consider that EM has converged and then these are our final parameters.

Putting it all together

Now that we have all these pieces of information together, let's put it all together:

```
#' Expectation Step of the EM Algorithm
#'
#' Calculate the posterior probabilities (soft labels) that each component
#' has to each data point.
#'
#' @param sd.vector Vector containing the standard deviations of each component
#' @param mu.vector Vector containing the mean of each component
#' @param alpha.vector Vector containing the mixing weights  of each component
#' @return Named list containing the loglik and posterior.df
e_step <- function(x, mu.vector, sd.vector, alpha.vector) {
  comp1.prod <- dnorm(x, mu.vector[1], sd.vector[1]) * alpha.vector[1]  # given a certain distribution,
  comp2.prod <- dnorm(x, mu.vector[2], sd.vector[2]) * alpha.vector[2]  # yields each data point
  sum.of.comps <- comp1.prod + comp2.prod   # marginal likelihood
  comp1.post <- comp1.prod / sum.of.comps   #posterior: given each data point, the proba it belongs to
  comp2.post <- comp2.prod / sum.of.comps

  sum.of.comps.ln <- log(sum.of.comps, base = exp(1))
  sum.of.comps.ln.sum <- sum(sum.of.comps.ln)  #log-likelihood the stacked distribution produces all th

  list("loglik" = sum.of.comps.ln.sum,
       "posterior.df" = cbind(comp1.post, comp2.post))
}
```

```r
#' Maximization Step of the EM Algorithm
#'
#' Update the Component Parameters
#'
#' @param x Input data.
#' @param posterior.df Posterior probability data.frame.
#' @return Named list containing the mean (mu), variance (var), and mixing
#'   weights (alpha) for each component.
m_step <- function(x, posterior.df) {
  comp1.n <- sum(posterior.df[, 1])   # updated N for each components
  comp2.n <- sum(posterior.df[, 2])

  comp1.mu <- 1/comp1.n * sum(posterior.df[, 1] * x)
  comp2.mu <- 1/comp2.n * sum(posterior.df[, 2] * x)

  comp1.var <- sum(posterior.df[, 1] * (x - comp1.mu)^2) * 1/comp1.n
  comp2.var <- sum(posterior.df[, 2] * (x - comp2.mu)^2) * 1/comp2.n

  comp1.alpha <- comp1.n / length(x)
  comp2.alpha <- comp2.n / length(x)

  list("mu" = c(comp1.mu, comp2.mu),
       "var" = c(comp1.var, comp2.var),
       "alpha" = c(comp1.alpha, comp2.alpha))
}
```

Now we just need to write a loop to go between the functions for each EM step. Each iteration will consist of 2 steps, first calling the e_step function and then calling the m_step function (if needed). We will run this for 50 iterations or when the log likelihood difference between two iteration is less than 1e-6 (whichever comes first):

```r
for (i in 1:50) {
  if (i == 1) {
    # Initialization
    e.step <- e_step(wait, wait.summary.df[["mu"]], wait.summary.df[["std"]],
                     wait.summary.df[["alpha"]])
    m.step <- m_step(wait, e.step[["posterior.df"]])
    cur.loglik <- e.step[["loglik"]]
    loglik.vector <- e.step[["loglik"]]
  } else {
    # Repeat E and M steps till convergence
    e.step <- e_step(wait, m.step[["mu"]], sqrt(m.step[["var"]]),
                     m.step[["alpha"]])
    m.step <- m_step(wait, e.step[["posterior.df"]])
    loglik.vector <- c(loglik.vector, e.step[["loglik"]])

    loglik.diff <- abs((cur.loglik - e.step[["loglik"]]))
    if(loglik.diff < 1e-6) {
      break
    } else {
      cur.loglik <- e.step[["loglik"]]
    }
  }
}
```

```
loglik.vector
```

```
##  [1] -1034.246 -1034.047 -1034.020 -1034.010 -1034.005 -1034.003 -1034.002
##  [8] -1034.002 -1034.002 -1034.002 -1034.002 -1034.002 -1034.002 -1034.002
## [15] -1034.002 -1034.002
```

As you can see, we actual stopped running EM after 16 iterations because the log likelihood didn't change much (specifically, the difference between the 15th and 16th iteration was $< 1e6$). We classify this as convergence of the algorithm and this represents our final fit.

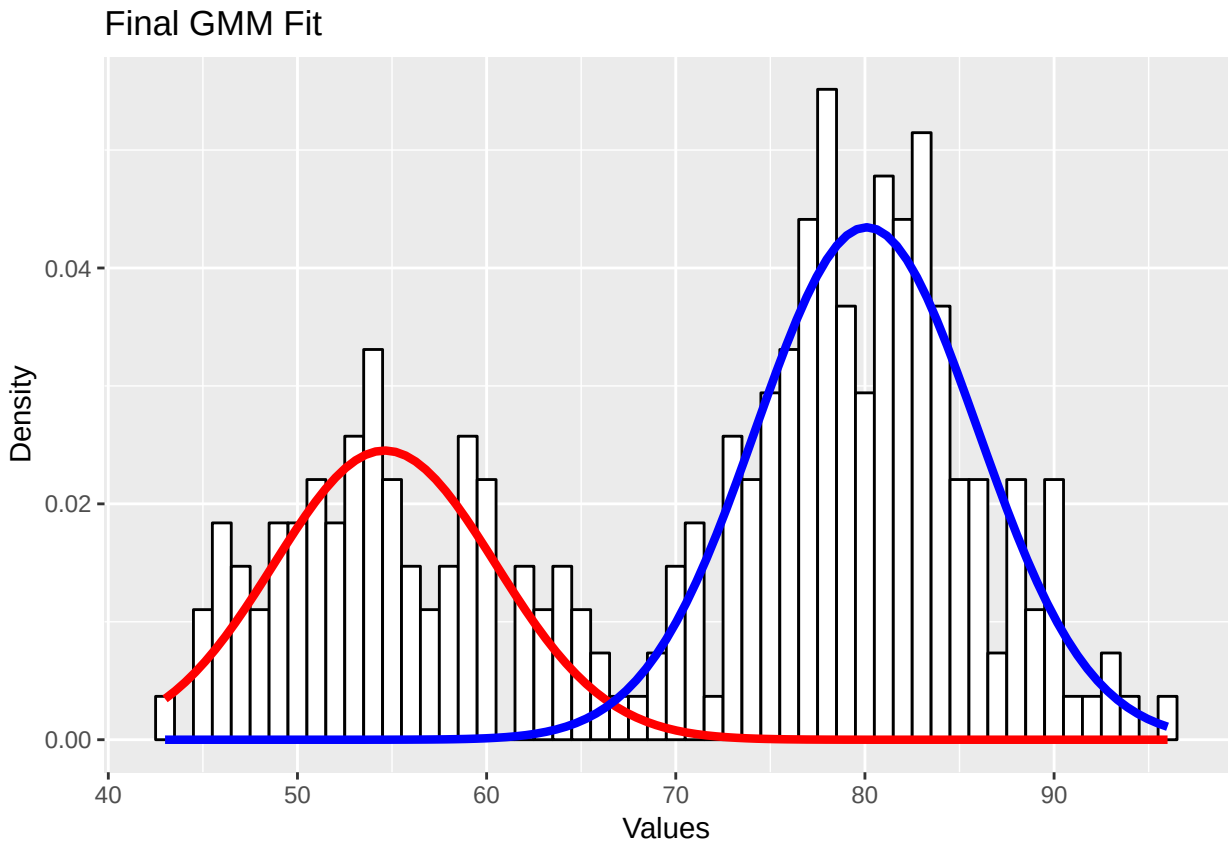
So our final component parameters are as follows:

```
m.step
```

```
## $mu
## [1] 54.61510 80.09122
##
## $var
## [1] 34.47368 34.42849
##
## $alpha
## [1] 0.3608934 0.6391066
```

Which produces the following plot:

```
#' Plot a Mixture Component
#'
#' @param x Input ata.
#' @param mu Mean of component.
#' @param sigma Standard of component.
#' @param lam Mixture weight of component.
plot_mix_comps <- function(x, mu, sigma, lam) {
  lam * dnorm(x, mu, sigma)
}

data.frame(x = wait) %>%
  ggplot() +
  geom_histogram(aes(x, ..density..), binwidth = 1, colour = "black",
                 fill = "white") +
  stat_function(geom = "line", fun = plot_mix_comps,
                args = list(m.step$mu[1], sqrt(m.step$var[1]),
                            lam = m.step$alpha[1]),
                colour = "red", lwd = 1.5) +
  stat_function(geom = "line", fun = plot_mix_comps,
                args = list(m.step$mu[2], sqrt(m.step$var[2]),
                            lam = m.step$alpha[2]),
                colour = "blue", lwd = 1.5) +
  ylab("Density") +
  xlab("Values") +
  ggtitle("Final GMM Fit")
```


Final GMM Fit
Density
0.04
0.02
0.00
40
50
60
70
80
90
Values