Part 3 Section 3: Statistical Tests and Sampling

Aaron Robotham

Libraries needed for this chapter: magicaxis, foreach.

```
library(magicaxis, quietly=TRUE)
library(foreach, quietly=TRUE)
```

So everybody running this chapter gets the same outputs, we should first set the seed.

```
set.seed(1)
```

Frequentist Statistics

This lecture series is predominantly focussed on Bayesian methods, bit it is worth spending a small amount of time discussing that notorious alternative world view: Frequentist Statistics.

The two main approaches (there are others, not part of this course) differ in how they phrase their core question.

Ultimately Bayesian statistics asks:

"How likely is my model of reality given that I have observed some data?".

Frequentist statistics asks:

"Given I believe my model, how likely is it that we have observed something that is at least as extreme as our data?".

These two questions might, to the uninitiated, sound fairly similar. But they diverge strongly in terms of how the analysis is done. With Bayesian statistics what we do is infer the model, and establish errors on the parameters that define our model. With Frequentist statistics we instead make decisions on an established model, namely do we reject the Null hypothesis (where the Null hypothesis is usually that we believe the model)?

A key thing to realise with Frequentist statistics is that we only reject the Null, we do not strictly accept the model. This is because there are in reality infinity models, so all we can do is reject ones in serious tension with our data. Sometimes the Null is easy to establish, e.g. we might expect a random coin to be un-biased, so the Null becomes a 50/50 chance of getting heads or tails. Other times it is not clear what the model to be tested actually is. A key thing to take away from this is that you can never do model fitting (or parameter inference) using Frequentist statistics.

P-Values

P-values have acquired a bad reputation over the years, but in their defence they mean exactly what they are supposed to mean, and here we will show that quite clearly and concisely. It narrowly provides the following:

"Under the assumption that the Null hypothesis is true, the p-value is the chance that we observe some characteristic of the data at least as extreme as actually observed."

As we mentioned above the Null hypothesis is the fiducial model in most applications. So for instance we could ask the following:

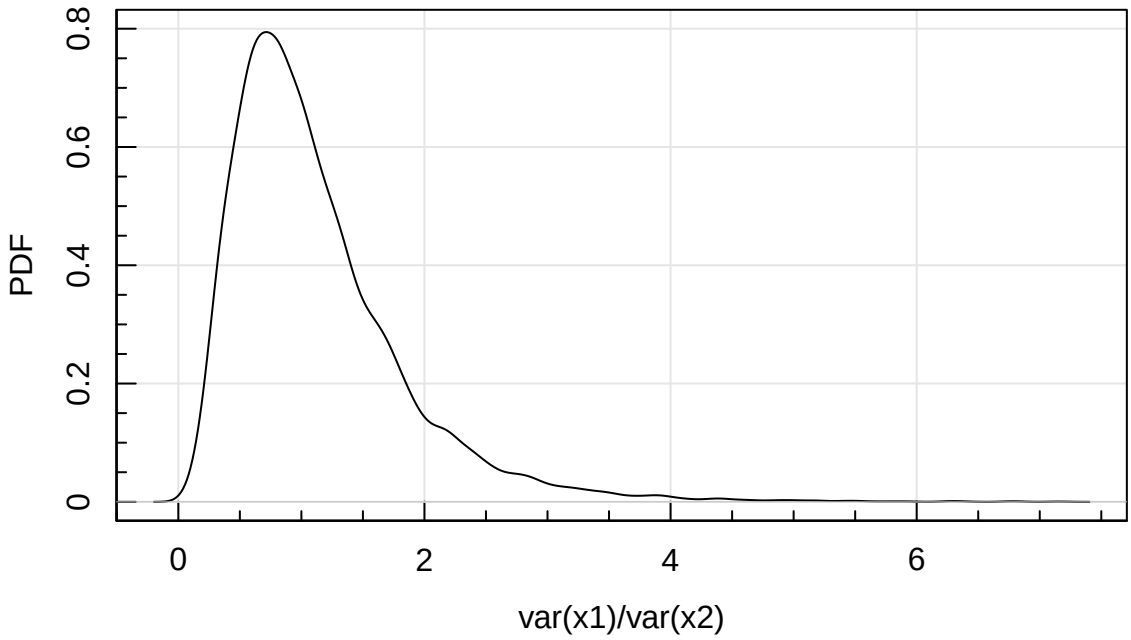
"Assuming two populations share the same parent Normal distribution, what is the chance that the ratio of their variances is as extreme as that measured?"

We can do this for a specific scenario where the two distributions are in fact the same (a Normal with mean 0 and sd 1), where one population has 10 samples and the other has 20 samples. We can do this experiment 1,000 times using the **foreach** function.

```
var_test = foreach(i=1:1e4, .combine='c')%do%{
  samp1 = rnorm(10)
  samp2 = rnorm(20)
  var(samp1)/var(samp2)
  }
```

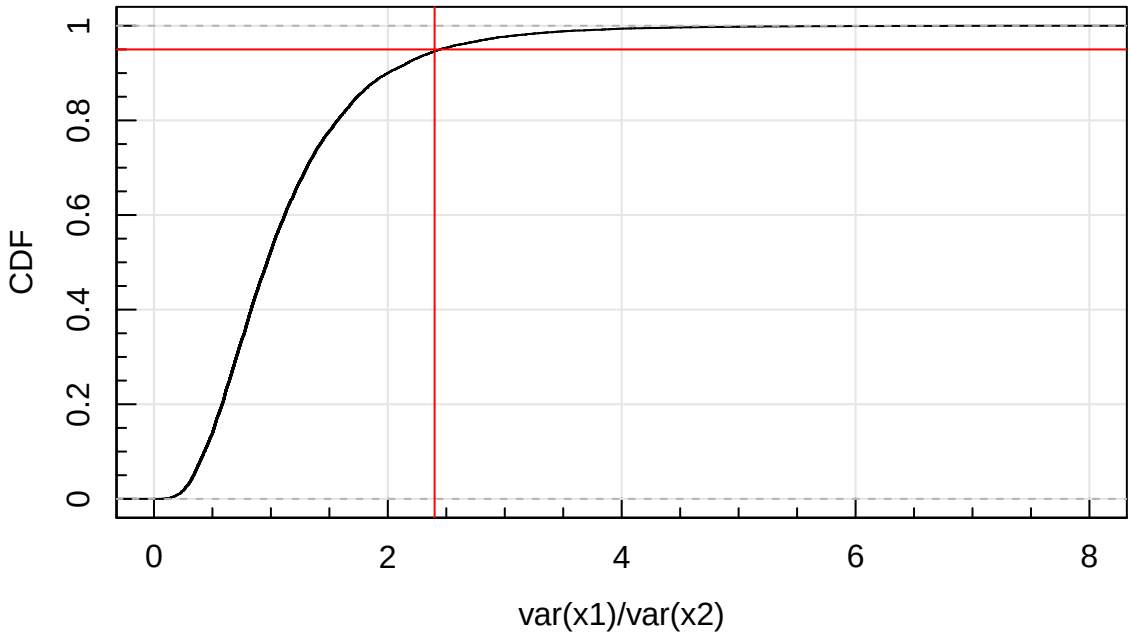
We can then plot the density of the result:

```
magplot(density(var_test), ylim=c(0,0.8), xlab='var(x1)/var(x2)', ylab='PDF')
```



And the cumulative density:

```
magplot(ecdf(var_test), xlim=c(0,8), ylim=c(0,1), xlab='var(x1)/var(x2)', ylab='CDF')
abline(h=0.95, col='red')
abline(v=2.4, col='red')
```



The red lines above indicates the point where approximately 5% of our most extreme simulated experiments lie (i.e. largest ratio of variances). This means if we find a variance ratio between two samples (of size 10 and 20) larger than 2.4 we can say there is less than a 5% chance of this happening by chance.

Unlucky or Not?

In the last two years I have cycled to work 600 times, I pass 8 lights and not once in 600 journeys have all 8 been green. Is this weirdly bad luck or not assuming all lights are red/green 50% of the time?

```
pbinom(0,600,0.5^8)*100
```

```
## [1] 9.552765
```

So 9.6% of the time you might expect this to happen, so the lights being 50:50 red/green cannot be rejected at the 5% threshold. How many more journeys would I need to take before I would 'reject' this model of reality?

```
pbinom(0,766,0.5^8)*100
```

```
## [1] 4.98844
```

The Bayesian might say instead 'there's a 50% chance such an outcome will have happened, so given that what's the most likely fraction of time the lights are green?'. This means calculating (I will let you figure out the origin of the maths):

```
greenfrac = (1 - (0.5^(1/600)))^(1/8)
print(greenfrac)
```

```
## [1] 0.4293416
```

So putting this back in:

```
pbinom(0,600,greenfrac^8)*100
```

```
## [1] 50
```

Which is what we expected to find.

You could also ask what is the 'expectation' of the fraction of time the lights are green, i.e. calculating the mean weighted ($p(x)$) probability (x, our green fraction we want to know) over the domain 0 - 1:

$$\mathrm{E}(x) = \frac{\int_0^1 x.p(x).dx}{\int_0^1 p(x).dx}$$

```r
int_x.p.dx = integrate(function(x){x*dbinom(0,600,x)}, lower=0, upper=1)$value

int_p.dx = integrate(function(x){dbinom(0,600,x)}, lower=0, upper=1)$value

(int_x.p.dx/int_p.dx)^(1/8)
```

```
## [1] 0.448139
```

This answer is actually very close to the simpler median (which makes sense because the true value seems to be close to 50%).

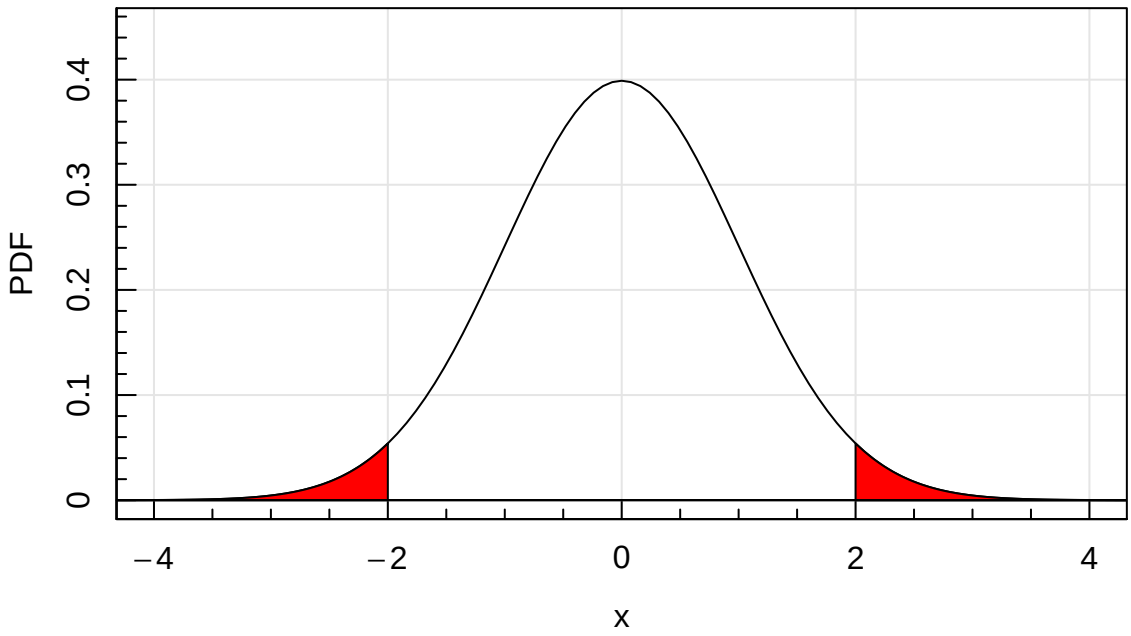
Hopefully you can see we are asking fundamentally different questions when approaching things the Frequentist or Bayesian way!

The Tail Wags the Dog

A complicating factor in the above form of test is asking whether both extremes (high and low probability) should be considered (so a two tailed test) or just one of them (one tailed test).

Usually we are agnostic about what constitutes an extreme event, e.g. consider a random sample of the Normal distribution. Since it is a symmetrical distribution then we should care equally about an event at $x = -2$ as $x = 2$:

```r
magcurve(dnorm, xlim=c(-4,4), ylim=c(0,0.45), xlab='x', ylab='PDF')
polygon(c(-5,-2,seq(-2,-5,by=-0.025),-5), c(0,0,dnorm(seq(-2,-5,by=-0.025)),0), col='red')
polygon(c(5,2,seq(2,5,by=0.025),-5), c(0,0,dnorm(seq(2,5,by=0.025)),0), col='red')
```



In the above case the integral of all of the red areas is simply

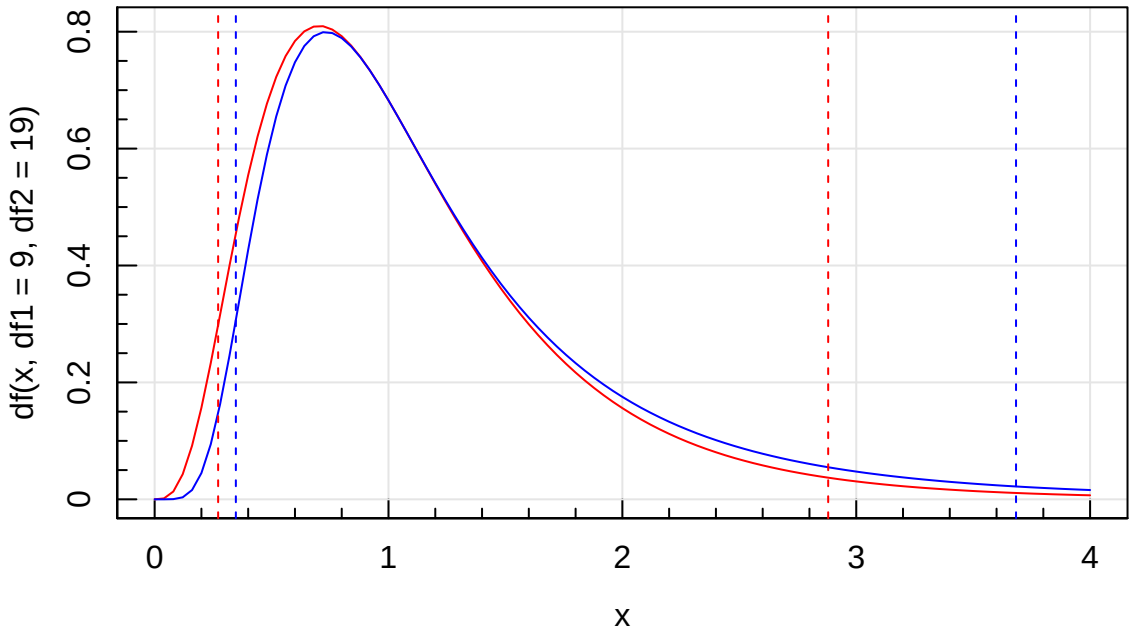
```r
pnorm(-2)*2
```

```
## [1] 0.04550026
```

Again, since we tend to be agnostic about what constitutes "extreme", we tend to put probability mass into both extremes. This is easy for the Normal since it is symmetrical, but it has less clear analogues for other distributions such as the exponential and the F. In the case of the F distribution, there are two ways of inputting the variables that impact the shape of the PDF:

```r
magcurve(df(x, df1=9, df2=19), col='red', from=0, to=4)
curve(df(x, df1=19, df2=9), col='blue', add=TRUE)
```

```
abline(v=qf(c(0.025,1-0.025), df1=9, df2=19), col='red', lty=2)
abline(v=qf(c(0.025,1-0.025), df1=19, df2=9), col='blue', lty=2)
```



You can see from the above how these probabilities pair off with each other. e.g. these statements are equivalent:

```
pf(0.5,df1=9,df2=19)
```

```
## [1] 0.1435525
```

```
1 - pf(1/0.5,df1=19,df2=9)
```

```
## [1] 0.1435525
```

As are these:

```
pf(0.5,df1=19,df2=9)
```

```
## [1] 0.0974132
```

```
1 - pf(1/0.5,df1=9,df2=19)
```

```
## [1] 0.0974132
```

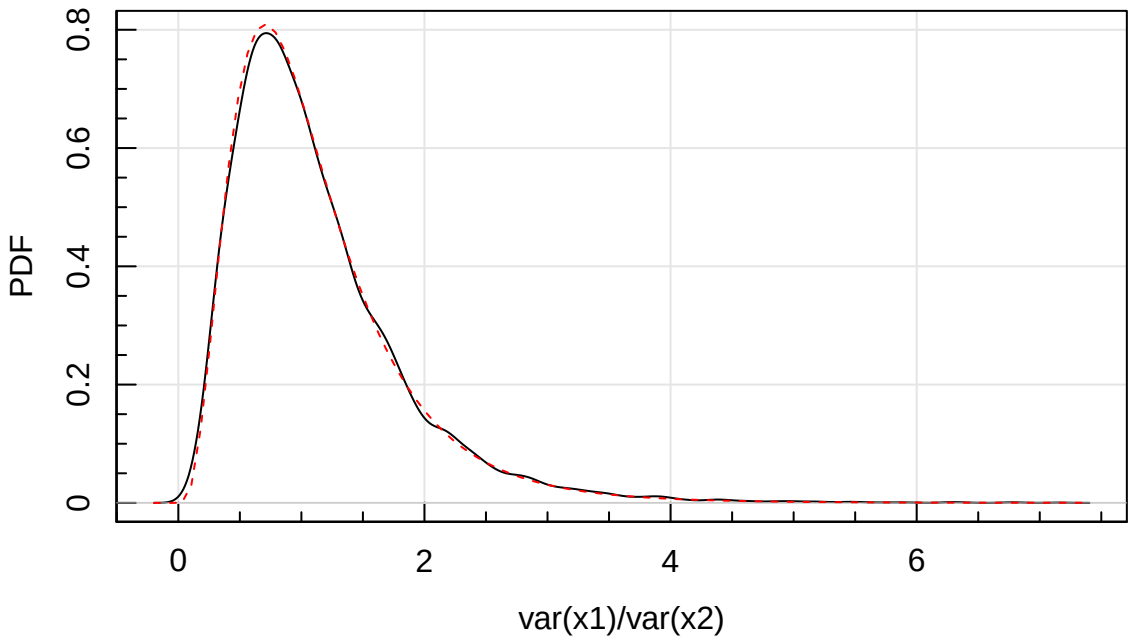
In the case of the F distribution (and other non symmetrical distributions) we need to be careful to consider whether it matters that we are in the sub 0.5 or over 0.5 part of the integrated PDF.

The F-Test

Whilst we can measure the likelihood of obtaining an extreme result through experiments like this, repeating this process for every combination of sample sizes with enough samples to be accurate would be an impossible task. However, the exact experiment we described above is actually described analytically by a known distribution which we have already covered: the F distribution. This depends only on two parameters, the degrees of freedom of the two samples (which is the sample size minus one: $N - 1$). The detailed form of the F distribution is discussed in an earlier part of the course.

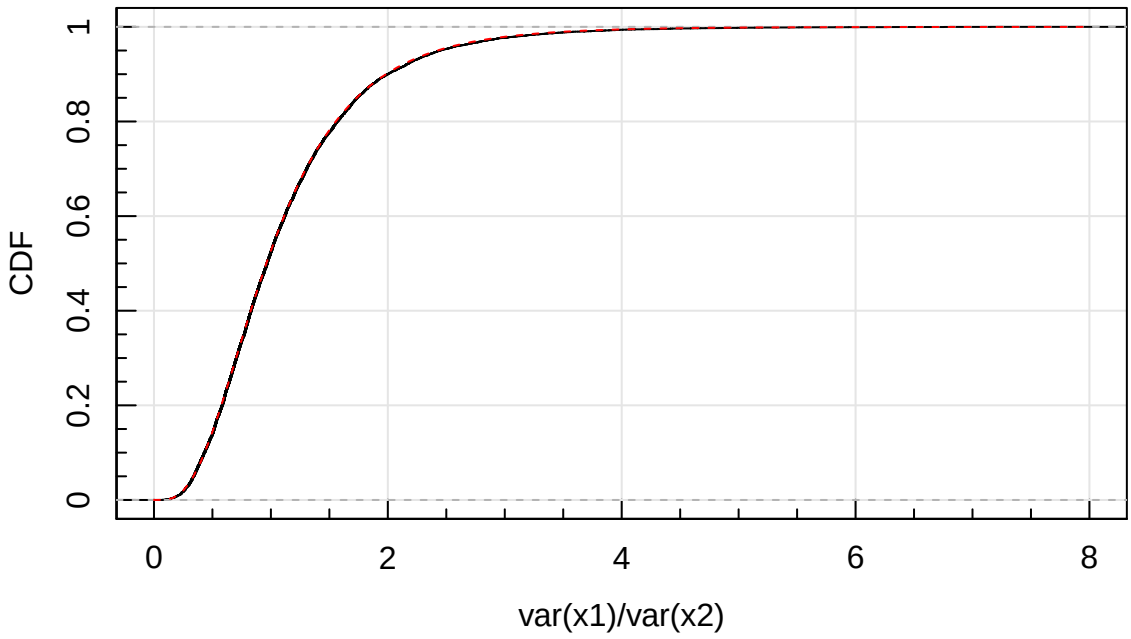
We can overlay the F distribution PDF and CDF on the above plots:

```
magplot(density(var_test), ylim=c(0,0.8), xlab='var(x1)/var(x2)', ylab='PDF')
curve(df(x, df1=9, df2=19), col='red', lty=2, add=TRUE)
```



Looks to be a good fit. We can further show the empirical cumulative distribution function version of the above using the **ecdf** function:

```
magplot(ecdf(var_test), xlim=c(0,8), ylim=c(0,1), xlab='var(x1)/var(x2)', ylab='CDF')
curve(pf(x, df1=9, df2=19), col='red', lty=2, add=TRUE)
```



This does an excellent job of representing our experiment simulations. Using this distribution it is trivial to ask what the chance is of observing up to a given ratio of variance for any two samples, e.g.:

```
pf(2.5, 9, 19)
```

```
## [1] 0.9556282
```

```
pf(1.4, 99, 199)
```

```
## [1] 0.9763306
```

```
pf(3, 5, 3)
```

```
## [1] 0.8025771
```

This means the most unlikely of the above three experiments is the second one. Even through the ratio of the variance is the least extreme numerically, it is unusual since the sample sizes are very large, and for this reason we would expect ratios much nearer to 1.

The output from the **pf** function is a probability (p), and used in this context the p-value of the so called F-test (based on sampling the analytic F distribution) is $2p$ if $p < 0.5$ and $2(1-p)$ otherwise. This is because the Frequentist p-value is interested in the chance of observing an outcome *at least as extreme* as some data. So for the last example we can compute:

```
var_test[1e4] #our last experiment
```

```
## [1] 1.738991
```

```
p_f_test = pf(var_test[1e4], 9, 19)
ifelse(p_f_test<0.5, 2*p_f_test, 2*(1-p_f_test))
```

```
## [1] 0.2965654
```

Or we can skip all these fiddly computations, and just use **R**'s built in F-test function **var.test**:

```
var.test(samp1, samp2)
```

```
##
##  F test to compare two variances
##
## data:  samp1 and samp2
## F = 1.739, num df = 9, denom df = 19, p-value = 0.2966
## alternative hypothesis: true ratio of variances is not equal to 1
## 95 percent confidence interval:
##  0.6038055 6.4052928
## sample estimates:
## ratio of variances
##           1.738991
```

It is important to note that the F-test is only sensitive to differences in the variances, not the means (μ), this means the following gives the same result, even though there is a big shift in the distribution mean for sample 1:

```
var.test(samp1+100, samp2)
```

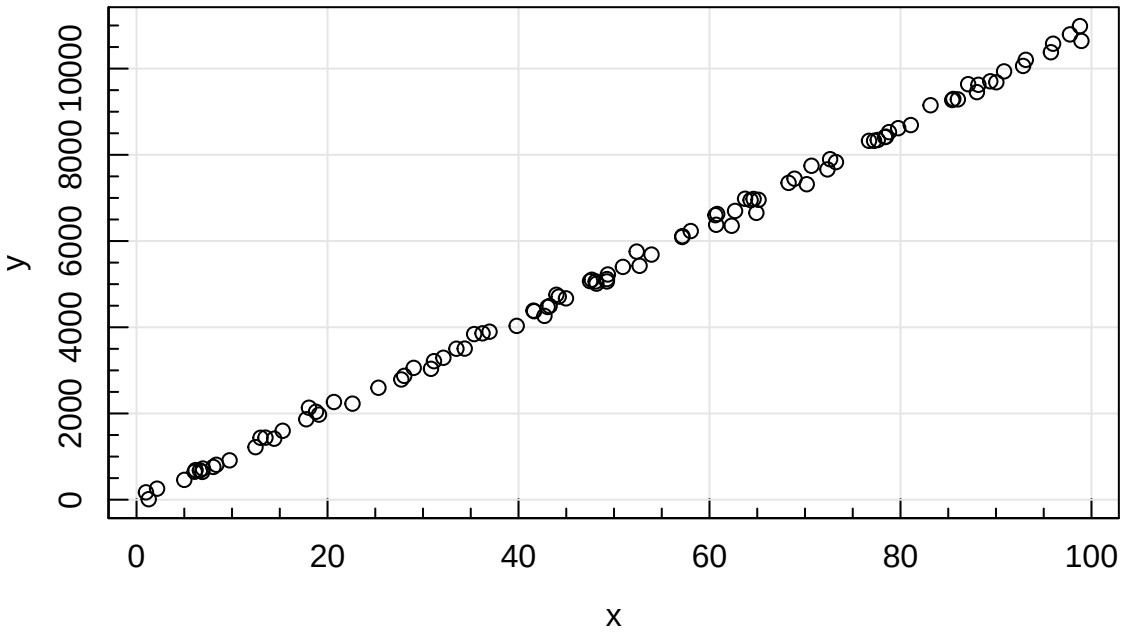
```
##
##  F test to compare two variances
##
## data:  samp1 + 100 and samp2
## F = 1.739, num df = 9, denom df = 19, p-value = 0.2966
## alternative hypothesis: true ratio of variances is not equal to 1
## 95 percent confidence interval:
##  0.6038055 6.4052928
## sample estimates:
## ratio of variances
##           1.738991
```

This is important when deciding what test to apply. If the distributions only differ in the means then the F-test will *not* be sensitive, and a different test should be used (such as the following t-test).

F-Test for Model Complexity

A common, but perhaps not obvious, use of the F-test is to check whether we can justifiably fit a more complex model to data. Say we have some data that might have been produced by a linear or higher order polynomial with some scatter in both cases (in fact in this case we can see it is made with a weak quadratic x^2 term, so it is not in detail a pure linear relationship):

```
tempx = runif(100,0,100)
tempy = 100*tempx + 0.1*tempx^2 + rnorm(1e2,sd=100)
magplot(tempx, tempy, xlab='x', ylab='y', grid=TRUE)
```

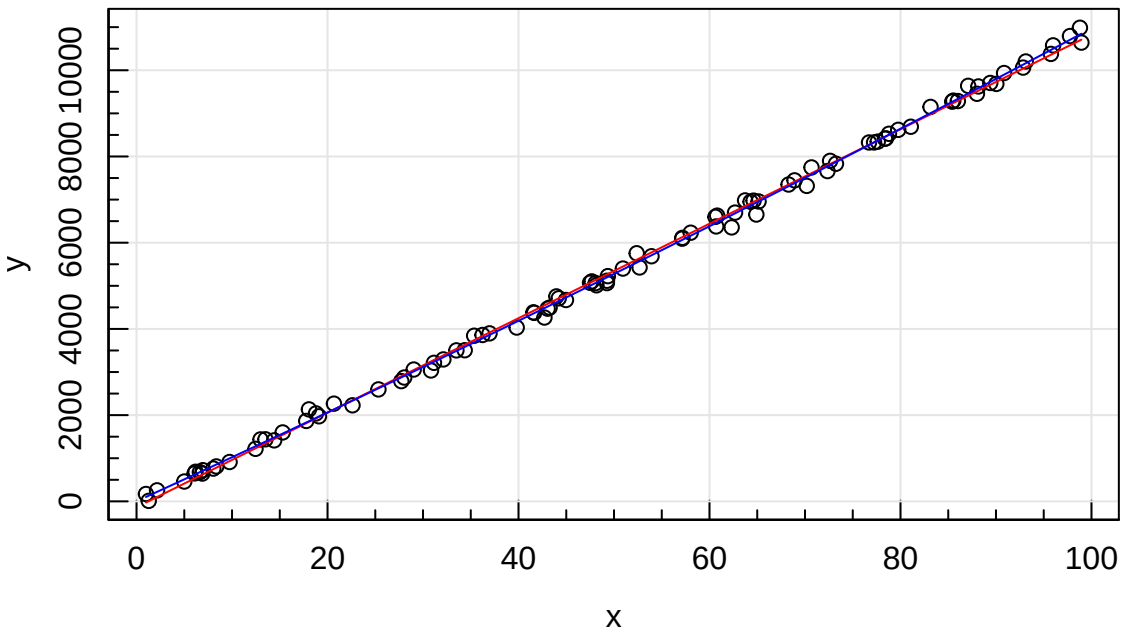


Now we can try fitting a linear model with pure linear and quadratic terms (we discuss linear model fitting in detail elsewhere in the course).

```
linmod = lm(tempy~tempx)
quadmod = lm(tempy~tempx + I(tempx^2))
```

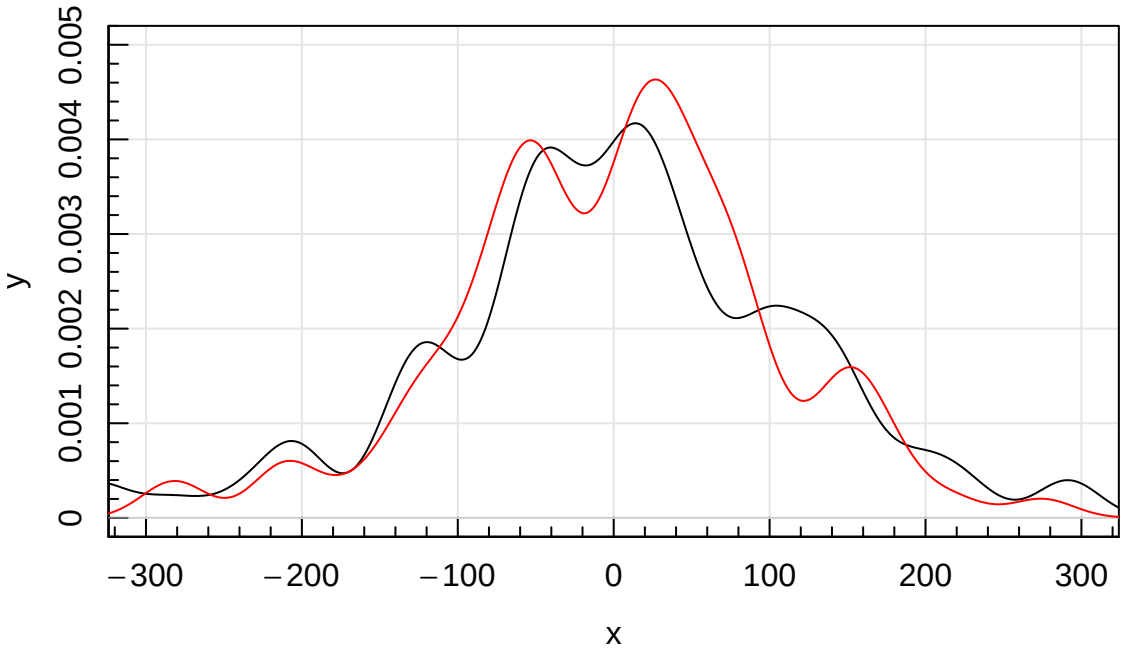
We can check how these fits look:

```
magplot(tempx, tempy, xlab='x', ylab='y', grid=TRUE)
curve(linmod$coefficients[1]+x*linmod$coefficients[2], add=TRUE, col='red')
curve(quadmod$coefficients[1]+x*quadmod$coefficients[2]+x^2*quadmod$coefficients[3],
      add=TRUE, col='blue')
```



A handy thing that comes out of using **lm** is it directly returns the residual scatter around the best fit, i.e.:

```
magplot(density(linmod$residuals, bw=20), xlim=c(-300,300), ylim=c(0,5e-3),
        xlab='x', ylab='y')
lines(density(quadmod$residuals, bw=20), col='red')
```



We can see above that the quadratic fit (red line) has smaller residuals then the black. But we can use the F-test to quantify whether this improvement is *significant*. It turns out that for such a scenario we can form an F statistic, that should itself be distributed under the F distribution:

$$F = \frac{\frac{\chi_1^2 - \chi_2^2}{\nu_2 - \nu_1}}{\frac{\chi_2^2}{n - \nu_2}}$$

Where χ_1^2 and χ_2^2 are the error weighted residuals for model 1 and 2 respectively (just the squared residuals in the example above, since we do no have any stated errors), ν_1 and ν_2 are the number of parameters for model 1 and 2 respectively (where $\nu_2 > \nu_1$), and n is the number of observations.

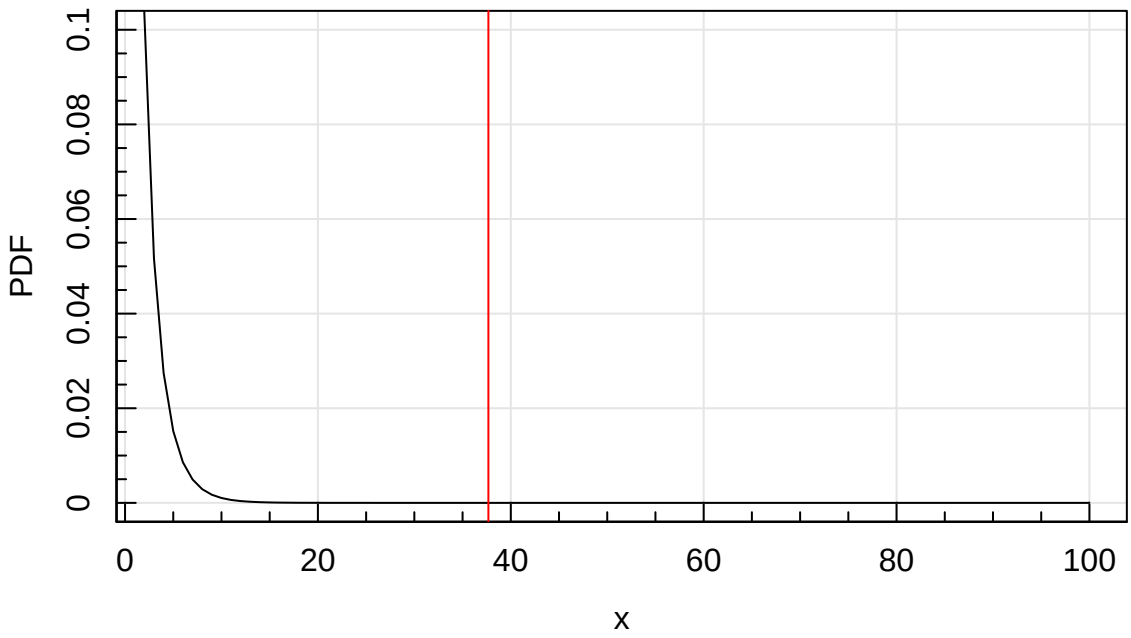
In our case this means we compute:

```
Fcomp = ((sum(linmod$residuals^2)-sum(quadmod$residuals^2))/(3-2))/
  ((sum(quadmod$residuals^2))/(100-3))
Fcomp
```

```
## [1] 37.67825
```

In principle the above should be distributed as an F distribution with $df_1 = \nu_2 - \nu_1$ and $df_2 = n - \nu_2$, i.e.:

```
magcurve(df(x, df1=3-2, df2=100-3), to=100, xlab='x', ylab='PDF', ylim=c(0,0.1))
abline(v=Fcomp, col='red')
```



The Null we test with the F distribution is the assumption that model 2 *does not* provide a significantly better fit to the data, given the additional freedom we have. In this case the p-value we would find is

```
1 - pf(Fcomp, df1=3-2, df2=100-3)
```

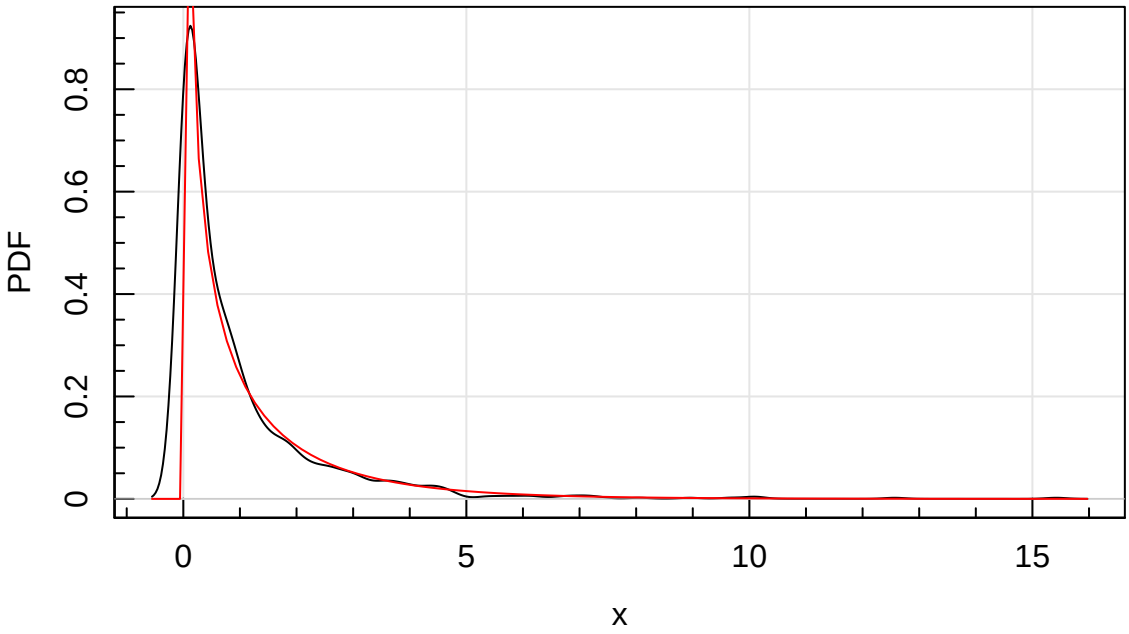
```
## [1] 1.823425e-08
```

If we are doing our typical 5% significance test, then we would certainly reject the Null and say that model 2 does a better job of fitting the data, even penalising ourselves for the fact we have more parameters.

For a final test of intuition, lets do the above steps 1,000 times for a linear model with no quadratic term, and check that we get the F distribution we expect.

```
Fcomp_multi= foreach(i=1:1e3, .combine='c')%do%{
  tempx = runif(100,0,100)
  tempy = 100*tempx + rnorm(1e2,sd=100)
  linmod = lm(tempy~tempx)
  quadmod = lm(tempy~tempx + I(tempx^2))
  ((sum(linmod$residuals^2)-sum(quadmod$residuals^2))/(3-2))/
  ((sum(quadmod$residuals^2))/(100-3))
}
```

```
magplot(density(Fcomp_multi), xlab='x', ylab='PDF')
curve(df(x, df1=3-2, df2=100-3), add=TRUE, col='red')
```



The above concept can be generalised to quite a broad range of tests between models, and it allows an elegant way of preventing a specific class of over-fitting of data. This is a common issue in data analysis, since most complex models contain simpler models as natural subsets, so we need to be careful not make models unnecessarily complex given the data available.

The t-Test

We can make a similar comparison of Normal samples using the t-test, which (it will not surprise you to find out, given its origin) uses the Student-t distribution to compute the p-values. Critically this test is sensitive to differences in both the mean (μ) and the standard deviation (σ). First we define our t statistic:

$$t = \frac{\bar{X}_1 - \bar{X}_2}{s_p\sqrt{\frac{1}{n_1} + \frac{1}{n_2}}},$$

where $\bar{X}_1$ and $\bar{X}_2$ and the sample means for population 1 and 2, and n_1 and n_2 are the sample size respectively. s_p is the pooled standard deviation for the two samples, given as:

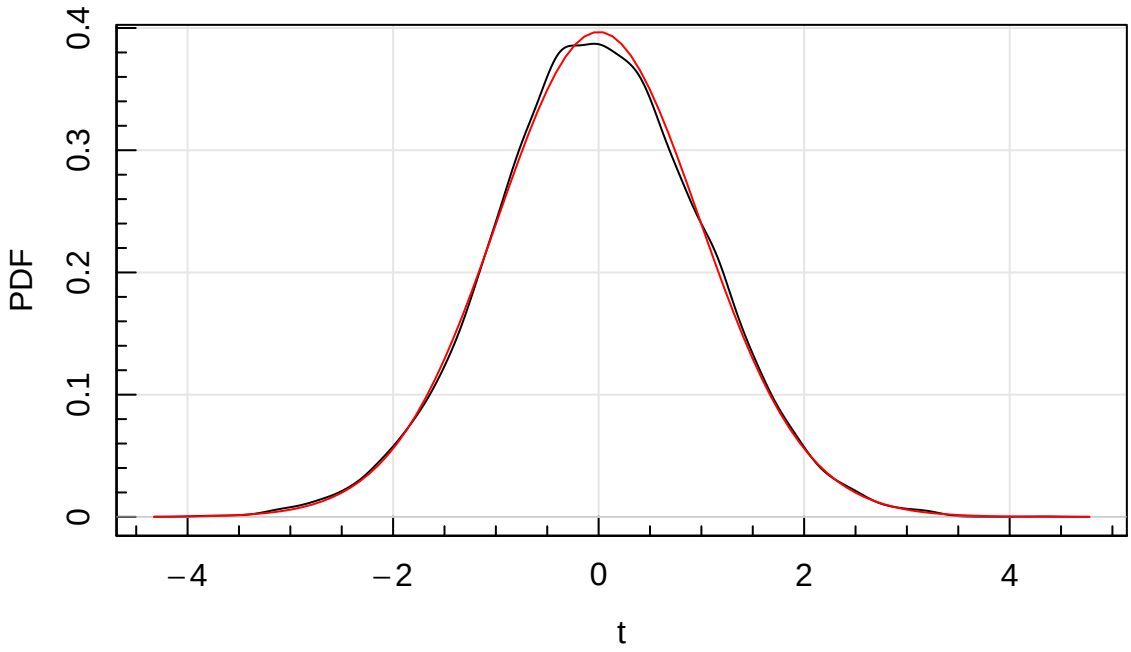
$$s_p = \sqrt{\frac{(n_1-1)s_1^2 + (n_2-1)s_2^2}{n_1 + n_2 - 2}},$$

where s_1 and s_2 are the unbiased estimators for the population standard deviations. Defined in this way, the total number of degrees-of-freedom for comparing to the t distribution is $\nu = n_1 + n_2 - 2$. We can easily simulate the above test statistic and compare to the t distribution:

```
t_test = foreach(i = 1:1e4, .combine='c')%do%{
  samp1 = rnorm(20)
  samp2 = rnorm(30)
  sp = sqrt(((20 - 1)*sd(samp1)^2+(30 - 1)*sd(samp2)^2)/(20 + 30 - 2))
  (mean(samp1) - mean(samp2))/(sp*sqrt(1/20 + 1/30))
}
```

And now plotting the test statistic versus the expected t distribution:

```
magplot(density(t_test), xlab='t', ylab='PDF')
curve(dt(x, df=20+30-2), col='red', add=TRUE)
```



Clearly it represents the simulated data well. Much like the F-test, we can now calculate the probability of witnessing an extreme event. For the above example with distributions of size 20 and 30, we would say they are significantly different at the 5% level if the t statistic computed is either below qt(0.025,df=20+30-2) = -2.0106348, or above qt(0.975,df=20+30-2) = 2.0106348. This should be make sense through symmetry arguments, since it is arbitrary which way round we define sample 1 and sample 2.

We can calculate the final t-test statistic and p-value with

```
t_test[1e4]
```

```
## [1] -0.2985857
```

```
p_t_test = pt(t_test[1e4], df=20+30-2)
ifelse(p_t_test<0.5, 2*p_t_test, 2*(1-p_t_test))
```

```
## [1] 0.766545
```

Also like the F-test, **R** has a built in function to compute the t-test p-value so we do not have to remember all the above steps.

```
t.test(samp1, samp2, var.equal = TRUE)
```

```
##
##  Two Sample t-test
##
## data:  samp1 and samp2
## t = -0.29859, df = 48, p-value = 0.7665
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.7185354  0.5327200
## sample estimates:
##  mean of x  mean of y
## 0.07579976 0.16870747
```

Setting *var.equal* = TRUE means we are not only expecting our two samples to share the same population mean, but also the same population variance (which in our example we do). It is worth emphasising that the t-test is actually sensitive to differences in the population *means*, not the variances. If samples are believed to significantly vary in their variances then you should be using the F-test above.

Z-Test

If the samples being considered are very large then people often switch to using the Z-test instead of the t-test for convenience since it uses the Normal approximation to the t-test for large samples. The statistic is therefore defined as:

$$Z = \frac{\bar{X} - \mu_0}{s},$$

where μ_0 is the true population mean and s is the true sample standard-deviation given by $s = \sqrt{\sigma^2/n}$ (where σ is the true population variance. Since these *true* values are rarely known (and you would need large n to accurately approximate them), it is usually a better idea to use the more complex but more accurate t-test. If these are known then confidence intervals can be computed directly from the normal distribution. I.e. the chance of observing a sample Z less than -1.96 is:

```
pnorm(-1.96)
```

```
## [1] 0.0249979
```

And therefore the Z-test would fail a two-tailed p-value threshold of 0.05 for $Z < -1.96 \vee Z > 1.96$.

ANOVA Test

Closely related to the above tests, and in particular the F-test, is the analysis of variance test (ANOVA). This answers a specific and narrow question: can the variance between populations be purely explained by them all sharing a common parent population with different sampling (i.e. that is our Null hypothesis we wish to test)? ANOVA is a class of omnibus test, which means it is a test that is run on lots of samples at once.

The ANOVA F statistic is defined such that:

$$F = \frac{\sum_{j=1}^{k} n_j(\bar{x_j} - \bar{x})^2/(k-1)}{\sum_{j=1}^{k}\sum_{i=1}^{n_j}(\bar{x_{ij}} - \bar{x_j})^2/(n-k)}$$

Where, $\bar{x}$ is the overall sample mean, $\bar{x}_j$ is the group j sample mean, k is the number of groups, n_j is sample size of group j and n is the total number of samples. This should form a PDF under the F distribution with $\nu_1 = k - 1$ and $\nu_2 = n - k$.

To test this we will first make some test data:

```
samp1 = rnorm(30, mean=60, sd=210)
samp2 = rnorm(20, mean=-20, sd=190)
samp3 = rnorm(50, mean=20, sd=240)
samp4 = rnorm(40, mean=-5, sd=200)
samp5 = rnorm(60, mean=-40, sd=180)
samp6 = rnorm(10, mean=30, sd=235)

samp_omni = list(samp1,samp2,samp3,samp4,samp5,samp6)
```

We will now compute our F statistic:

```
samp_mean = mean(unlist(samp_omni))
samp_N = length(unlist(samp_omni))
samp_groupN = length(samp_omni)
nu_1 = (samp_groupN - 1)
nu_2 = (samp_N - samp_groupN)

top = 0
for(i in 1:samp_groupN){
  top = top + length(samp_omni[[i]])*(samp_mean-mean(samp_omni[[i]]))^2/nu_1
}
```

```
bottom = 0
for(i in 1:samp_groupN){
  for(j in 1:length(samp_omni[[i]])){
    bottom = bottom + (samp_omni[[i]][j] - mean(samp_omni[[i]]))^2/nu_2
  }
}

F_aov = top/bottom
F_aov
```

```
## [1] 4.235636
```

A subtle point is that the ANOVA test is defined to detect significant differences in sample versus ensemble means. This means we are only interested in knowing if the variance seen between group means is much larger than we would expect (we do not care if they are smaller). As such we have to conduct a one-tailed F-test which will always be on the high probability side of the F distribution. So to compute our final F statistic we do:

```
F_pval = 1 - pf(F_aov, df1 = nu_1, df2 = nu_2)
F_pval
```

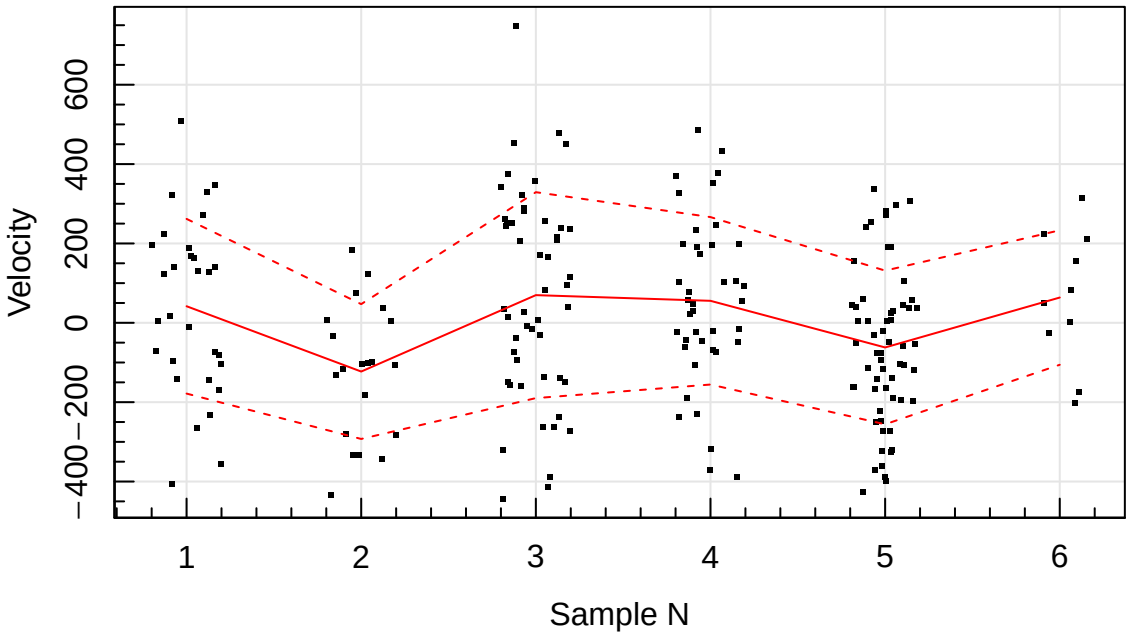
```
## [1] 0.001099542
```

To use these samples in an ANOVA test using the base **R** function **aov** we first need to put them into a data frame format, turning the sample numbers into factors. Factors are a manner of identifying common groups in **R**.

```
comb = rbind(cbind(samp1,1), cbind(samp2,2), cbind(samp3,3), cbind(samp4,4), cbind(samp5,5),
           cbind(samp6,6))
colnames(comb) = c('Velocity', 'sampN')
comb = as.data.frame(comb)
comb$sampN = as.factor(comb$sampN)
```

We can see how much the groups differ by plotting a running mean and standard deviation ranges using **magrun**:

```
magplot(jitter(as.numeric(comb$sampN)), comb$Velocity, pch='.',
        cex=3, xlab='Sample N', ylab='Velocity')
temprun = magrun(as.numeric(comb$sampN), comb$Velocity, type='mean', bins=6, equalN=FALSE)
lines(temprun, col='red')
lines(temprun$x, temprun$ysd[,1], col='red', lty=2)
lines(temprun$x, temprun$ysd[,2], col='red', lty=2)
```



With the data now set up correctly, we can show the **summary** output of the **aov** ANOVA function in **R**:

```
summary(aov(formula=Velocity~sampN, data=comb))
```

```
##              Df  Sum Sq Mean Sq F value Pr(>F)
## sampN         5  981988  196398   4.236 0.0011 **
## Residuals   204 9459057   46368
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

At the top right of this summary we can see the F statistic and p-value of interest, which happily agrees with the ones we just computed. In this example we do not have enough evidence to reject the Null hypothesis at the 5% significance level.

Since the omnibus test asks a narrow question, and does not actually specify where any sample tension arises (i.e. which sample is actually the outlier) it is instructive to do the full 6x6 F-test omnibus directly:

```
F_omni = matrix(0,6,6)

for(i in 1:6){
  for(j in 1:6){
    F_omni[i,j] = var.test(samp_omni[[i]],samp_omni[[j]])$p.value
  }
}

F_omni
```

```
##           [,1]       [,2]       [,3]      [,4]       [,5]      [,6]
## [1,] 1.0000000 0.24152962 0.34561878 0.7947837 0.39984397 0.4151649
## [2,] 0.2415296 1.00000000 0.04618394 0.3125920 0.53773107 0.9524522
## [3,] 0.3456188 0.04618394 1.00000000 0.1836032 0.03193805 0.1708373
## [4,] 0.7947837 0.31259202 0.18360322 1.0000000 0.54314403 0.4963376
## [5,] 0.3998440 0.53773107 0.03193805 0.5431440 1.00000000 0.7038721
## [6,] 0.4151649 0.95245220 0.17083726 0.4963376 0.70387213 1.0000000
```

We can extract the upper triangle and make a histogram of these individual F-test p-values:

```
F_omni[upper.tri(F_omni)]
```

```
##  [1] 0.24152962 0.34561878 0.04618394 0.79478374 0.31259202 0.18360322
```

```
##  [7] 0.39984397 0.53773107 0.03193805 0.54314403 0.41516493 0.95245220
## [13] 0.17083726 0.49633762 0.70387213
```

```
maghist(F_omni, xlab='p-value')
```

```
## Summary of used sample:
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## 0.03194 0.24153 0.45575 0.50976 0.79478 1.00000
```

```
## Pop Std Dev: 0.32612
```

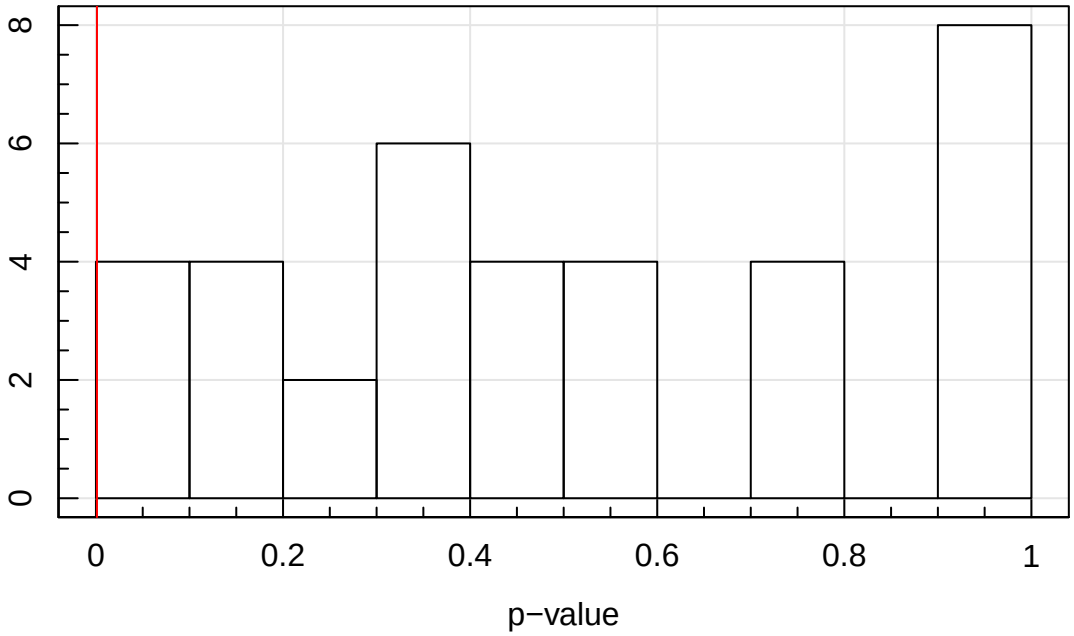
```
## MAD: 0.38568
```

```
## Half 16-84 Quan (1s): 0.39791
```

```
## Half 02-98 Quan (2s): 0.48403
```

```
## Using 36 out of 36
```

```
abline(v=F_pval, col='red')
```



So most 2 way tests are in less tension than the global ANOVA test. We can look at which group is in most common tension:

```
rowMeans(F_omni)
```

```
## [1] 0.5328235 0.5150815 0.2963635 0.5550768 0.5360882 0.6231107
```

The third group appears to be in the most tension (smallest value), which is consistent with the fact it has the largest dispersion (as we created it earlier).

And we can check the 2 groups with the most tension between them:

```
which(F_omni==min(F_omni), arr.ind=TRUE)
```

```
##      row col
## [1,]   3   5
```

So groups 3 and 5, which are also the 2 groups with the most individual members, making any tension easier to detect when F-testing.

Wald Test

Conceptually similar to the t-test is the Wald test. Here rather than comparing two samples, we assume we have one measurement of a population parameter with corresponding variance, and now we want to test if the difference to another sample is significant.

If our intrinsic parameter of interest is denoted $\hat{\theta}$ which has known variance $\text{var}(\hat{\theta})$ and our new estimate is θ_0, then we can specify the Wald test in 2 ways. In one we compute

$$W^2 = \frac{(\hat{\theta} - \theta_0)^2}{\text{var}(\hat{\theta})},$$

and in this case we expect W^2 to have a χ^2 distribution, which we use to produce our p-values. In the second way we compute

$$W = \frac{\hat{\theta} - \theta_0}{\text{se}(\hat{\theta})},$$

where $\text{se}(\hat{\theta})$ is the standard error given by $\text{se}(\hat{\theta}) = \sqrt{\text{var}(\hat{\theta})}$ and in this case we expect W to have a Normal distribution, which we use to produce out p-values. $\text{var}(\hat{\theta})$ has to be estimated for the Wald test (usually seen a weakness of it). It can actually be derived from the Fisher Information I_n (discussed in detail later in this course), in which case it becomes $\text{var}(\hat{\theta}) = 1/I_n$. We might use the Wald test when we have a known population mean and a known expected variance given the size of our proposed sample, say heights of adult males when out sample is of size 10. The F-test we discussed earlier for discriminating between model complexity is actually an extension of the Wald test.

Other Frequentist Tests

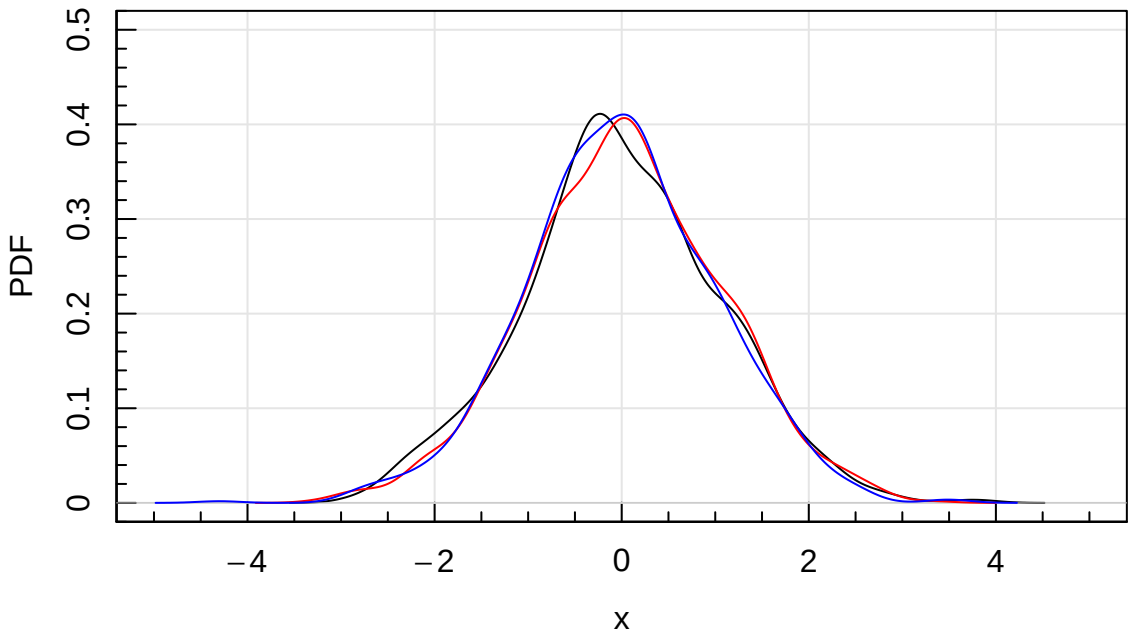
There are many tests that are similar to the above, in that they measure some characteristic of the data observed and compare to an analytic solution to determine how *significant* the observation is. The F-test and t-test are attractive examples since they make use of a fairly standard statistical distributions (the t and F distribution) directly. Many other tests are a bit more abstract regarding the quantity they compare against.

A common theme in all of the below tests is that they quantify some aspect of the data and assess whether this is "extreme" or not. In this sense it is perfectly possible to create you own metric of extremity, e.g. we could consider the sum of random Normal samples scaled by $\sqrt{(N)}$:

```
extreme_10 = foreach(1:1e3,.combine='c')%do%{sum(rnorm(10))/sqrt(10)}
extreme_100 = foreach(1:1e3,.combine='c')%do%{sum(rnorm(100))/sqrt(100)}
extreme_1000 = foreach(1:1e3,.combine='c')%do%{sum(rnorm(1000))/sqrt(1000)}
```

And now we can show our expected PDFs:

```
magplot(density(extreme_10), xlim=c(-5,5), type='l', xlab='x', ylab='PDF', ylim=c(0,0.5))
lines(density(extreme_100), col='red')
lines(density(extreme_1000), col='blue')
```



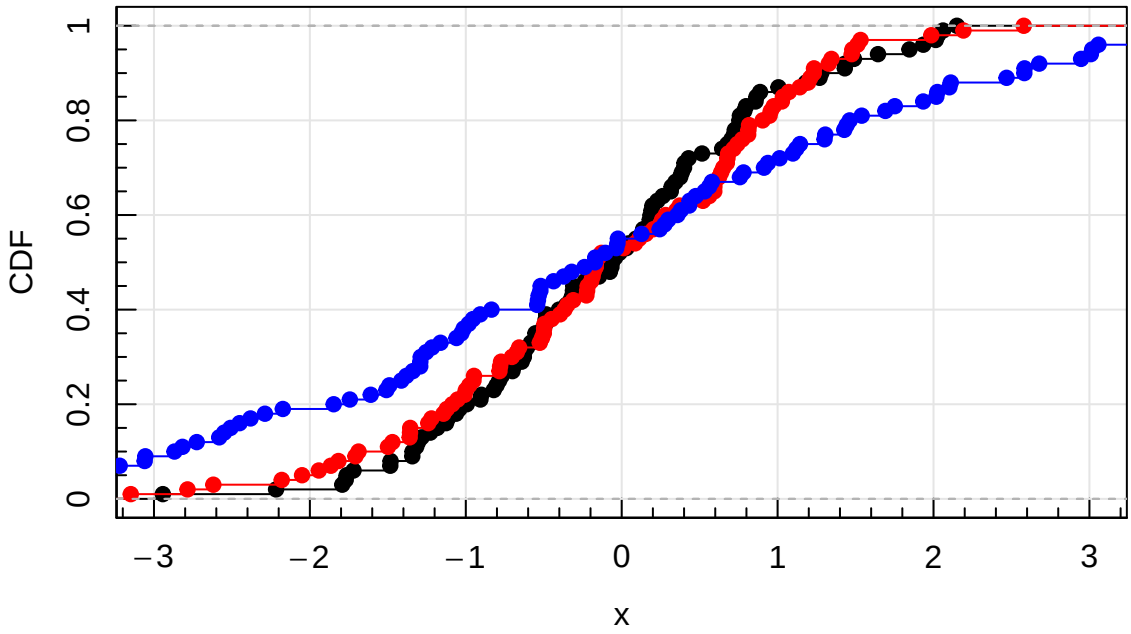
It seems we have just re-discovered the central limit theorem! Now we will look at some other classical tests.

The Kolmogorov-Smirnov Test

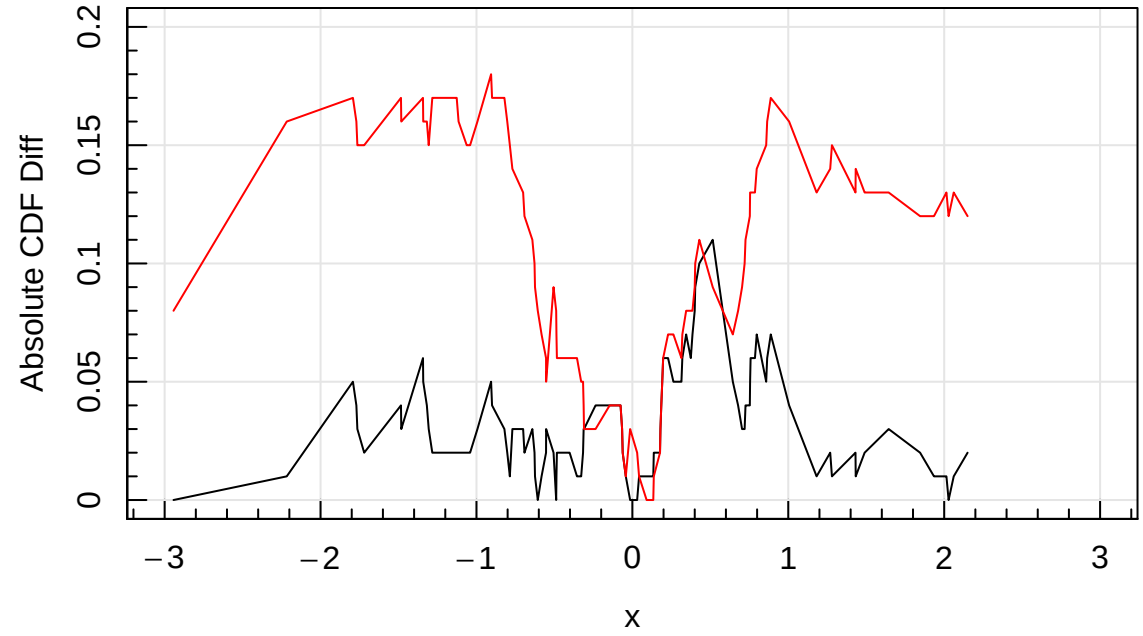
A very popular test (mostly due to its simplicity and adaptability) is the Kolmogorov-Smirnov Test. Here the quantity being assessed is the maximum amount of disagreement between the CDF of two sample distributions, or potentially one sample distribution and a target analytic distribution (e.g. the Normal). Consider this simple example:

```
set.seed(666)
norm_ks1 = rnorm(1e2)
norm_ks2 = rnorm(1e2)
norm_ks2_mod = rnorm(1e2,sd=2)
```

```
magplot(ecdf(norm_ks1), xlim=c(-3,3), ylim=c(0,1), xlab='x', ylab='CDF')
lines(ecdf(norm_ks2), col='red')
lines(ecdf(norm_ks2_mod), col='blue')
```




```
xcomp = sort(norm_ks1)
magplot(xcomp, abs(ecdf(norm_ks1)(xcomp)-ecdf(norm_ks2)(xcomp)),
        xlim=c(-3,3), ylim=c(0,0.2), xlab='x', ylab='Absolute CDF Diff', type='l', grid=TRUE)
lines(xcomp, abs(ecdf(norm_ks1)(xcomp)-ecdf(norm_ks2_mod)(xcomp)), col='red')
```



The red line above clearly has a much larger maximum disagreement between the CDFs, which makes sense since it was generated with a different Normal distribution ($\sigma = 2$ rather than $\sigma = 1$). The KS-test measures this maximum difference (~ 0.11 and ~ 0.18 in this case) and calculates how likely it is a difference this big *or larger* could happen by chance under the assumption that the two distributions actually share the same parent distribution.

Whilst that might sound like a mouthful, it is actually a very simple test, and it is entirely agnostic to the distribution/s that actually generated the data. It is only able to answer the narrow question as to whether we can reject the Null hypothesis that the distribution is shared at some level of significance. In **R** the KS test comes with the base **stats** package.

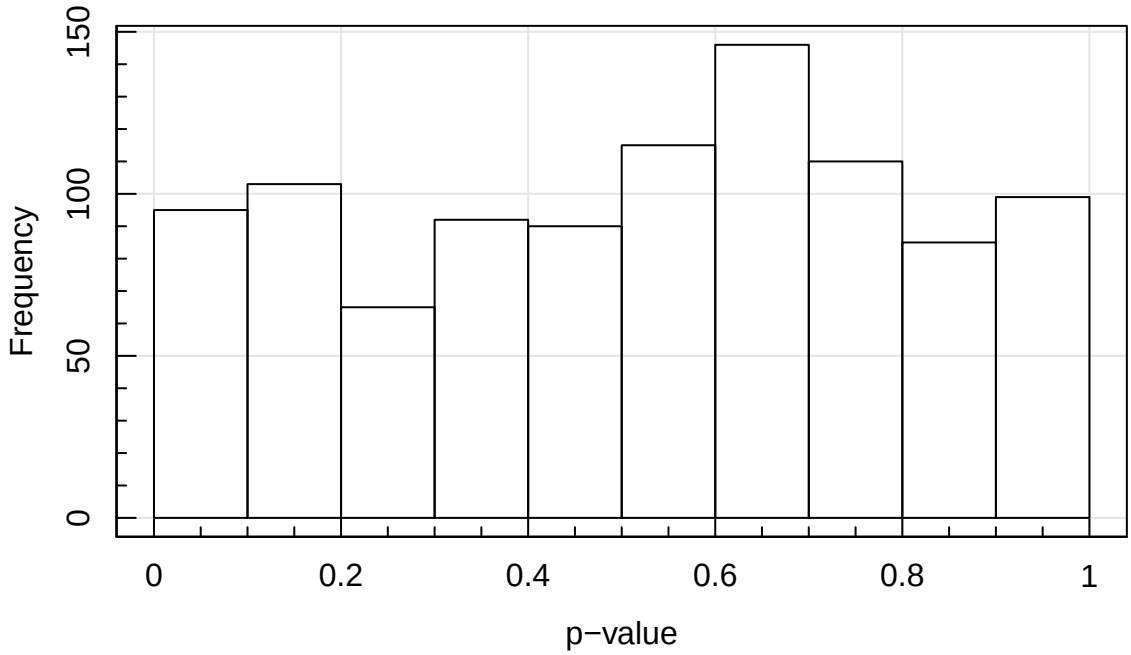
A key insight to make is that if we choose our significance level to be, e.g., 5% then by definition 5% of the time we will be rejecting the Null incorrectly. We can simulate this very easily:

```
ks_sim = foreach(i=1:1e3, .combine='c')%do%{ks.test(rnorm(1e3), rnorm(1e3))$p.value}
```

And check the distribution of p-values computed by the KS test:

```
maghist(ks_sim, xlab='p-value', ylab='Frequency')
```

```
## Summary of used sample:
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## 0.0004218 0.2877198 0.5360544 0.5155460 0.7590978 0.9987080
## Pop Std Dev: 0.28408
## MAD: 0.33068
## Half 16-84 Quan (1s): 0.33194
## Half 02-98 Quan (2s): 0.48152
## Using 1000 out of 1000
```

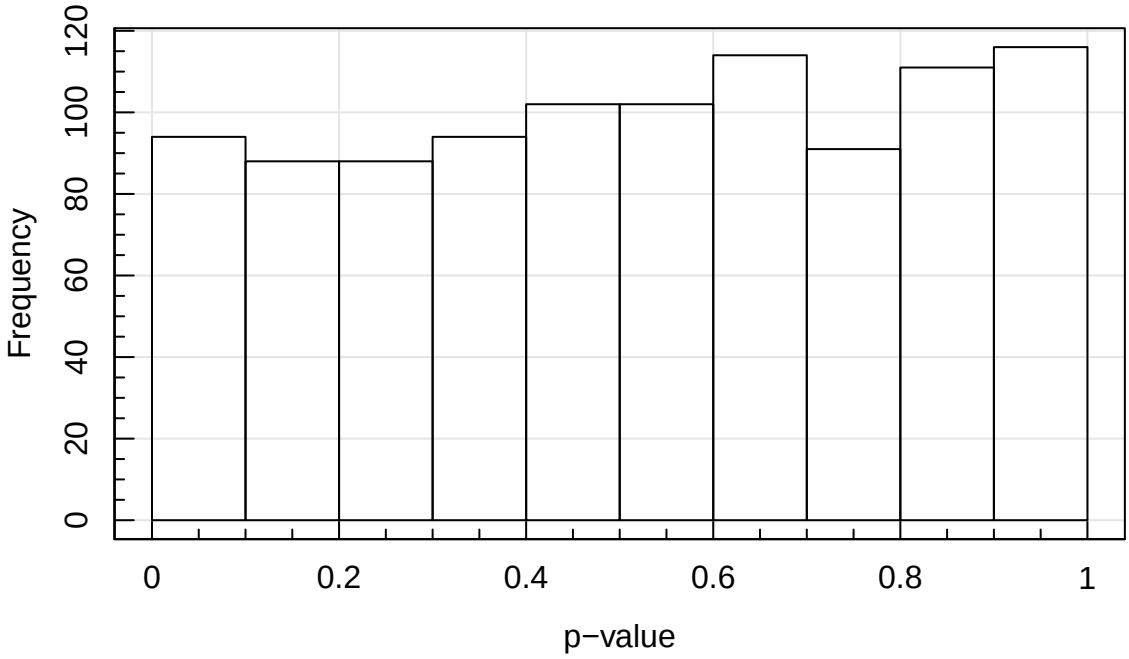


The above distribution should be Uniform between 0 and 1 (and visually approximately is). In fact, to complete the virtuous cycle we could even test whether the distribution of p-values is itself Uniform using a KS test (that is an exercise for the keen student).

We can actually use the KS-test for a known target distribution directly, i.e.:

```
ks_sim2 = foreach(i=1:1e3, .combine='c')%do%{ks.test(rnorm(1e3), 'pnorm', mean=0, sd=1)$p.value}
maghist(ks_sim2, xlab='p-value', ylab='Frequency')
```

```
## Summary of used sample:
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## 0.0002606 0.2861731 0.5349757 0.5218881 0.7756494 0.9998857
## Pop Std Dev: 0.28936
## MAD: 0.36371
## Half 16-84 Quan (1s): 0.34213
## Half 02-98 Quan (2s): 0.48061
## Using 1000 out of 1000
```



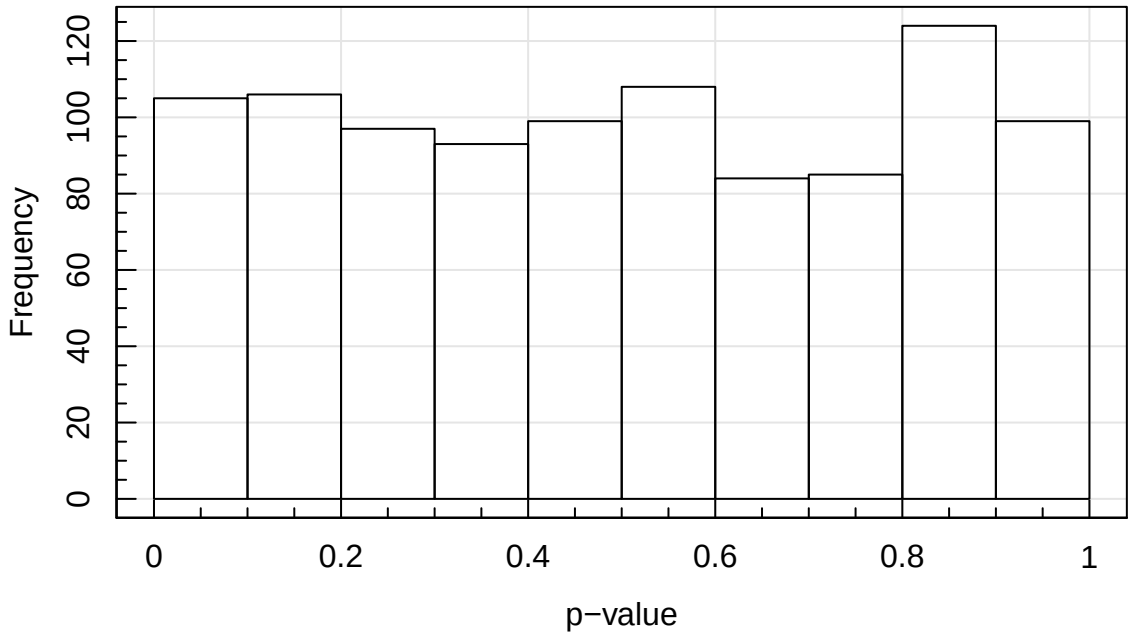
The caveat here is that it is not enough just to know the type of distribution (in this case Normal), but we also need to know the actual parameters of the distribution (a Normal with mean 0 and sd 1).

The Anderson-Darling Test

This parameter knowledge weakness of the KS-test is overcome with the Anderson-Darling test, often called the AD-test. In the background the AD-test also makes use of the comparison of CDFs, but it puts more weight on the tails, and is usually considered to be more sensitive. It also does not require explicit knowledge of the distribution parameters in question.

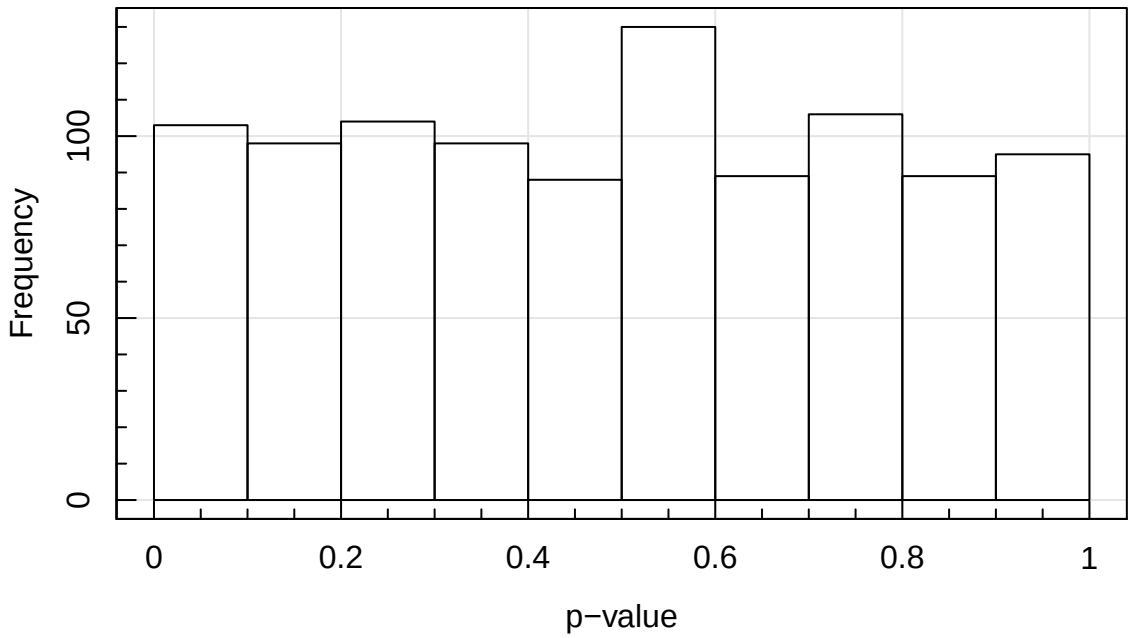
```
library(nortest)
ad_sim = foreach(i=1:1e3, .combine='c')%do%{ad.test(rnorm(1e3))$p.value}
ad_sim2 = foreach(i=1:1e3, .combine='c')%do%{ad.test(rnorm(1e3, mean=3, sd=10))$p.value}
maghist(ad_sim, xlab='p-value', ylab='Frequency')
```

```
## Summary of used sample:
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## 0.0001929 0.2363567 0.5001471 0.4995592 0.7692991 0.9985639
## Pop Std Dev: 0.29223
## MAD: 0.3933
## Half 16-84 Quan (1s): 0.35234
## Half 02-98 Quan (2s): 0.47349
## Using 1000 out of 1000
```




```
maghist(ad_sim2, xlab='p-value', ylab='Frequency')
```

```
## Summary of used sample:
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## 0.0003491 0.2442956 0.5032617 0.4930458 0.7323018 0.9970487
## Pop Std Dev: 0.28542
## MAD: 0.36361
## Half 16-84 Quan (1s): 0.343
## Half 02-98 Quan (2s): 0.47352
## Using 1000 out of 1000
```



This is a very useful feature of the AD-test, and in practice it makes it much more useful than the KS-test for the distributions for which 'critical values' have been computed. That is the downside, since the 'critical values' (the significance of deviance in its test statistic) varies for different distributions and needs

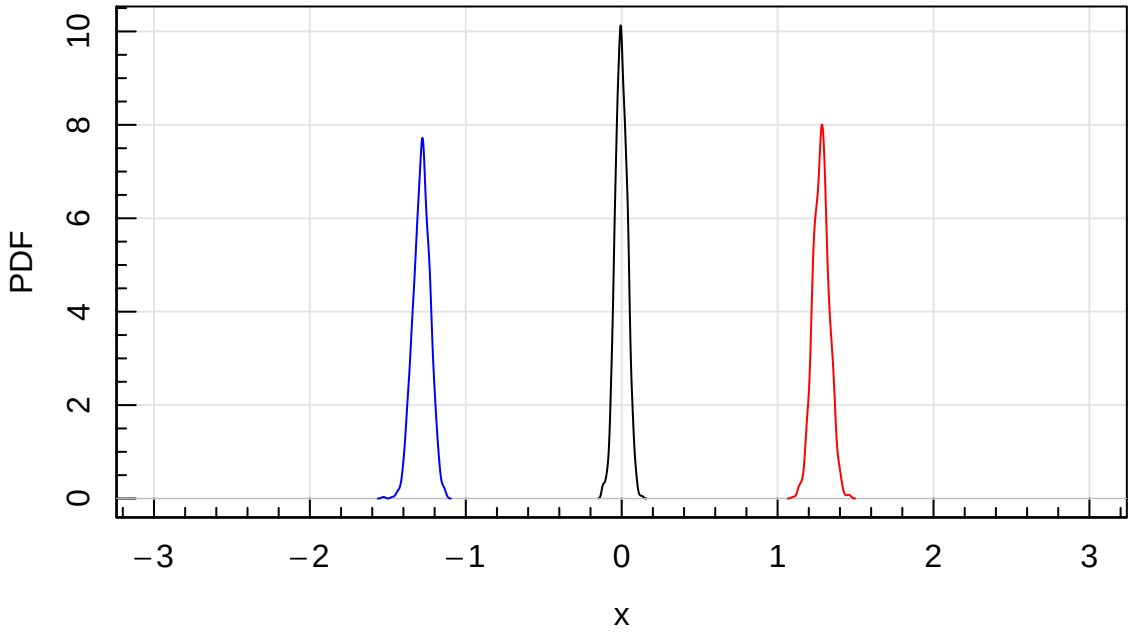
to be computed by somebody (and hopefully not you!). It also means it cannot be used to generically (blindly) compare two target distributions to test the hypothesis that they share a parent distribution.

Shapiro-Wilk Test

A different approach is used for the Shapiro-Wilk test (SW-test), which looks at the likelihood of every value being generated for a Normal distribution. The positive here is that it is usually regarded (and measured to be) the most powerful test of Normality, with an excellent ratio of true-positive to false-positive. However, an obvious limitation is that it can only be applied to testing for Normality (where KS-tests can be used for anything, and AD-tests have been developed for a fairly large suite of distributions).

In brief, the SW-test evaluates every sample based on its order statistic (where it appears in a sorted sequence), and compares this to the expected distribution of values for every order location. This is known analytically for the Normal distribution, but we can Monte-Carlo it approximately easily:

```
order100 = foreach(i=1:1e3, .combine='c')%do%{sort(rnorm(1e3))[100]}
order500 = foreach(i=1:1e3, .combine='c')%do%{sort(rnorm(1e3))[500]}
order900 = foreach(i=1:1e3, .combine='c')%do%{sort(rnorm(1e3))[900]}

magplot(density(order500), xlim=c(-3,3), type='l', xlab='x', ylab='PDF')
lines(density(order100), col='blue')
lines(density(order900), col='red')
```

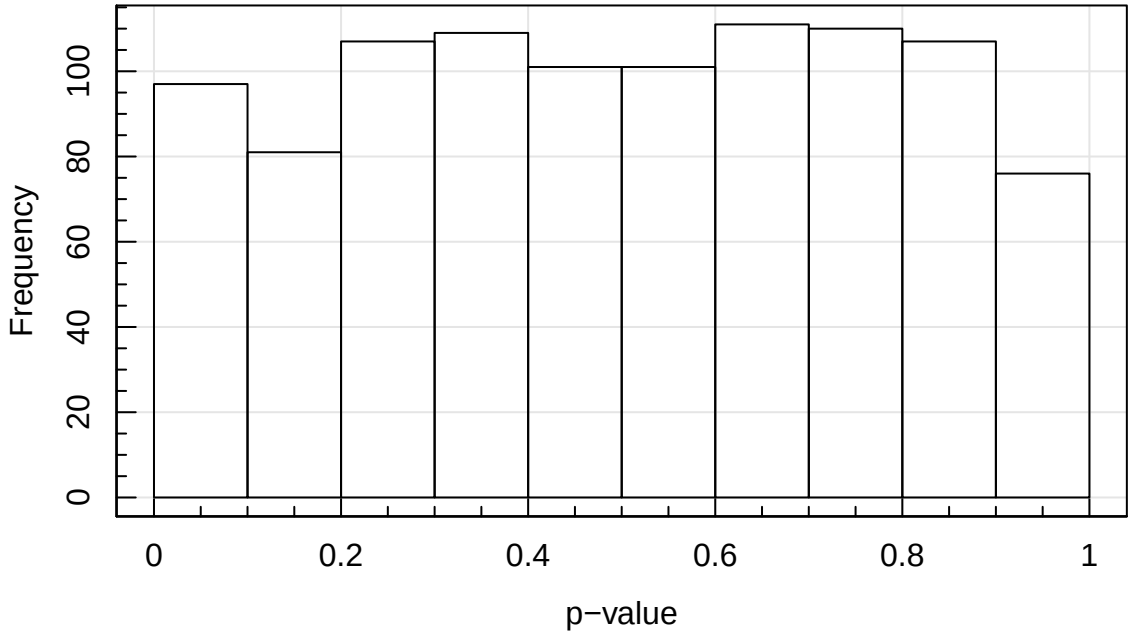


So we would expect the 100^{th} sample to be distributed close to -1.3 (qnorm(0.1)), the 500^{th} close to 0 (qnorm(0.5)) and the 900^{th} close to 1.3 (qnorm(0.9). Whilst the distributions might be non-intuitive (they are very spiky), the requirement for symmetry should be obvious. We can do a simulation similar to before:

```
sw_sim = foreach(i=1:1e3, .combine='c')%do%{shapiro.test(rnorm(1e3))$p.value}
sw_sim2 = foreach(i=1:1e3, .combine='c')%do%{shapiro.test(rnorm(1e3, mean=3, sd=10))$p.value}
maghist(sw_sim, xlab='p-value', ylab='Frequency')
```

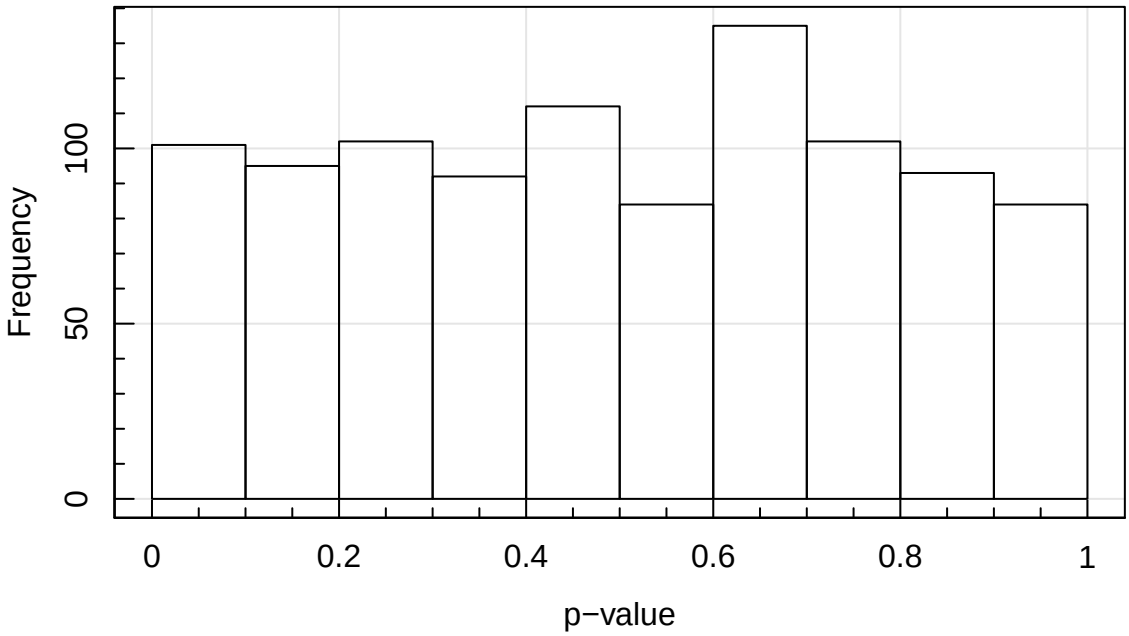
```
## Summary of used sample:

##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## 0.0004065 0.2623096 0.5056278 0.5007915 0.7440604 0.9983137

## Pop Std Dev: 0.28115
```

```
## MAD: 0.35641
## Half 16-84 Quan (1s): 0.32346
## Half 02-98 Quan (2s): 0.47184
## Using 1000 out of 1000
```




```
maghist(sw_sim2, xlab='p-value', ylab='Frequency')
```

```
## Summary of used sample:
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## 0.0003313 0.2509986 0.4993398 0.4980084 0.7304234 0.9976261
## Pop Std Dev: 0.28258
## MAD: 0.35267
## Half 16-84 Quan (1s): 0.33339
## Half 02-98 Quan (2s): 0.47428
## Using 1000 out of 1000
```



Again, we meet the expectation that the p-value distributions are themselves Uniformly distributed.

Is This Normal?

Generally the order of Normality test preferences goes SW-test (best), AD-test, KS-test (worst), however, this is also inversely ordered with how generically adaptable these tests are. As we have discussed, KS-tests can be applied to almost anything (assuming exact parameters are being tested), and are very easy to implement from scratch. AD-tests can be used for a fairly wide range of distributions, and within these families it is not necessary to know the parameters being tested. SW-tests can only be used to test for Normality.

You tend to see the KS-test used most widely, probably because it is simple to understand and implement. But in this era of well maintained statistics software (be it **R** or **Python** packages and modules), there is no excuse for not simply using the best tool for the job.

In the next part of this section we will make a clear distinction between the number of objects in a given sample (N) and the number of times we sample the data (n).

Jackknife Resampling

Jackknifing is the conceptually and computationally simplest approach to estimating model parameter errors from data. In its basic form you sample the data leaving out one data point, doing so n times. The two key parameters estimated from the output are then the mean and the so-called 'Jackknife Standard Error' defined as:

$$SE_j = \sqrt{\frac{n-1}{n}\sum_{i=1}^{n}(\hat{\theta}_m - \hat{\theta}_i)^2}$$

where $\hat{\theta}_m$ is the the mean of the sample statistics $\hat{\theta}_m = \frac{1}{n}\sum_{i=1}^{n}\hat{\theta}_i$, and $\hat{\theta}_i$ is the sample statistic of the i^{th} sample.

```
R1e3 = rnorm(1e3)
jack_mean = foreach(i = 1:1e3, .combine='c')%do%{
  sample = R1e3[-i]
```

```
  mean(sample)
}
```

We can now compute our Jackknife summary properties:

```
mean(jack_mean)
```

```
## [1] -0.003248956
```

```
sqrt((1e3 - 1)/1e3*sum((jack_mean - mean(jack_mean))^2))
```

```
## [1] 0.03121723
```

Computationally this is actually the same as:

```
sqrt(var(jack_mean)*(1e3 - 1)^2/1000)
```

```
## [1] 0.03121723
```

The extra complexity of a factor 999 is due to **R** computing unbiased sample variance by default, which is a normal sample variance with an extra factor of $n/(n-1)$.

We know from central limit theory that the standard error on the mean for 1,000 samples of the Normal distribution will be given by $\sqrt{1/1000} = 0.0316$, so our estimate is clearly very close. Hopefully it is clear why this is useful: you can generically estimate the errors in an experiment without doing the experiment lots of times, but still extracting the best single possible measurement possible.

The powerful aspect of the above is that this can be applied to a large variety of different estimators, i.e. the same approach could be used to calculate 1σ like interval limits for variances, medians, quantile ranges and other outcomes. However, some of these are known to become biased. Luckily that can be assessed by the jackknifing alone with its bias estimator:

$$B_j = (n-1)(\hat{\theta}_m - \hat{\theta})$$

where $\hat{\theta}$ is the target statistic applied to the full data. For the mean this happens to always be 0:

```
(1e3 - 1)*(mean(jack_mean) - mean(R1e3))
```

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

However if our target statistic was instead the "unbiased" standard deviation then we would find a bias:

```
jack_sd=foreach(i = 1:1e3, .combine='c')%do%{
  sample = R1e3[-i]
  sd(sample)
}
```

```
(1e3 - 1)*(mean(jack_sd) - sd(R1e3))
```

```
## [1] -0.0002186609
```

The above is the amount that the full sample statistic would typically under-represent the true value of the parent population. Naturally one can adjust any sample statistic for this bias, and the process is the justification for a wide variety of 'unbiased' estimators. E.g. to calculate the population variance from a sample of size N we would compute $Var_{pop} \sim Var_{samp}(N)/(N-1)$.

Jackknife Subsampling

A different method that is sometimes called "jackknifing" (especially in astronomy) is really sub-sampling. In simple terms this would be where we divide a large sample up into distinct regions with non-overlapping subsets and estimate uncertainties etc using those separate regions. For a simple example we can again use the Normal distribution:

```
jack_mean2 = foreach(i = 1:100, .combine='c')%do%{
  sample = R1e3[1:10+(i-1)*10]
  mean(sample)
}
```

The mean of the above will once again be the estimated population mean, but the error on this is now given by:

$$SE_{js} = \frac{\text{StdDev}(\hat{\theta}_i)}{\sqrt{n-1}}$$

Which we can compute easily:

```
mean(jack_mean2)
```

```
## [1] -0.003248956
```

```
sd(jack_mean2)/sqrt(100 - 1)
```

```
## [1] 0.03378532
```

We could also have divided out sample into 10 sub-samples each with 100 observations, in which case we would compute:

```
jack_mean3 = foreach(i = 1:10, .combine='c')%do%{
  sample = R1e3[1:100+(i-1)*100]
  mean(sample)
}
mean(jack_mean3)
```

```
## [1] -0.003248956
```

```
sd(jack_mean3)/sqrt(10-1)
```

```
## [1] 0.02680897
```

The error here is very similar to the first method. Note that both end up being a bit lower than what we computed with full jackknifing, and the latter should be preferred since we have ultimately sampled our data in more ways (1,000, rather than 10 or 100). This method of sub-sampling does sometimes make it easier to isolate "bad" observations, since they will only contribute to at most 1 sub-sample, rather than $n-1$ jackknife samples.

In general proper leave-one-out jackknifing is a better technique to use, but there are occasions where it makes sense to sub-sample your data. Say you can compute a correlation statistic in a contiguous volume of space, it might not make physical sense to "leave-one-out" when computing this. Instead you will do better to split your large volume into distinct sub-volumes are then compute your statistics (say a fit to the Schechter luminosity function) for each separate sub-volume of data, and then use the above to correctly compute the uncertainty from this process.

Bootstrap Resampling

Above we have looked at computational cheap, but surprisingly flexible and powerful, jackknife sampling. In some form this has been around since the 1940s, and served as a tractable route to estimating parameters with suitable confidence intervals. A step beyond this is bootstrap sampling, which allows for much deeper information extraction at the cost of computer resources.

The basic strategy behind bootstrap sampling is superficially similar to the above except instead of doing n samples of $n-1$ without replacement we do a very large number of n samples *with* replacement. Even for moderate n, the number of unique extractions possible for bootstrapping grows very quickly with n. From combination theory it can be shown that the number of distinct samples is given by:

$$\left(\left(\frac{n}{n}\right)\right) = \frac{(2n-1)!}{n!(n-1)!}$$

E.g. for only 10 samples we could do up to factorial(10*2 - 1)/factorial(10)/factorial(10 - 1)= 9.2378×10^4 extractions.

The main difference to jackknife re-sampling is that the standard error now looks different:

$$SE_b = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{\theta}_m - \hat{\theta}_i)^2}$$

where $\hat{\theta}_i$ is now the sample statistic for the i^{th} replacement sampling of the data.

To achieve good convergence we would generally want the number of extractions to be large, where the only limit is usually the computational resources available. In any case we can repeat the above experiment easily.

```
boot_mean = foreach(i = 1:1e3, .combine='c')%do%{
  sample = R1e3[sample(1e3,1e3,replace=TRUE)]
  mean(sample)
}
```

We can now compute our bootstrap summary properties:

```
mean(boot_mean)
```

```
## [1] -0.002931905
```

```
sqrt(1/1e3*sum((boot_mean-mean(boot_mean))^2))
```

```
## [1] 0.03144188
```

These are much like the values we estimated via the jackknife sampling, however it is now possible to do better by increasing the number of re-samples:

```
boot_mean = foreach(i = 1:1e4, .combine='c')%do%{
  sample = R1e3[sample(1e3,1e3,replace=TRUE)]
  mean(sample)
}
```

We can now compute our bootstrap summary properties:

```
mean(boot_mean)
```

```
## [1] -0.00296931
```

```
sqrt(1/1e4*sum((boot_mean-mean(boot_mean))^2))
```

```
## [1] 0.0311597
```

where the target for the SE is expected to be 0.0316 as before.

Similar to before, we can compute an expected bootstrap bias, but it is a bit different to the jackknife:

$$B_b = \hat{\theta}_m - \hat{\theta}$$

so in our case this is

```
mean(R1e3) - mean(boot_mean)
```

```
## [1] -0.0002796461
```

This is similar to what we found using jackknife re-samples.

Cross Validation and Leave One Out

Moving a step beyond bootstrapping is cross-validation (CV). The variant know as "leave one out" (LOO) is analogous to jackknifing, and "leave N out" (LNO) is more akin to bootstrapping. In both cases the idea is to model some aspect of the data on a subset, but compute validation statistics on the other part of the sample (hence the term "cross validation" ").

A simple analogy would be training and testing machine learning. You might attempt to optimise a neural network for image recognition using a subset of the data, but test it against the other part. And just like bootstrapping, you are not restricted in how many times you can do this (theoretically), instead you sample the data repeatedly and keep re-fitting and re-testing the network. Doing this exhaustively is very expensive, but practically people might do it 100s or 1,000s of times to calculate statistics on the quality of the network. CV is considered one of the most pragmatically rigorous approach to assessing models and data, with many other approaches being at best approximations of it.

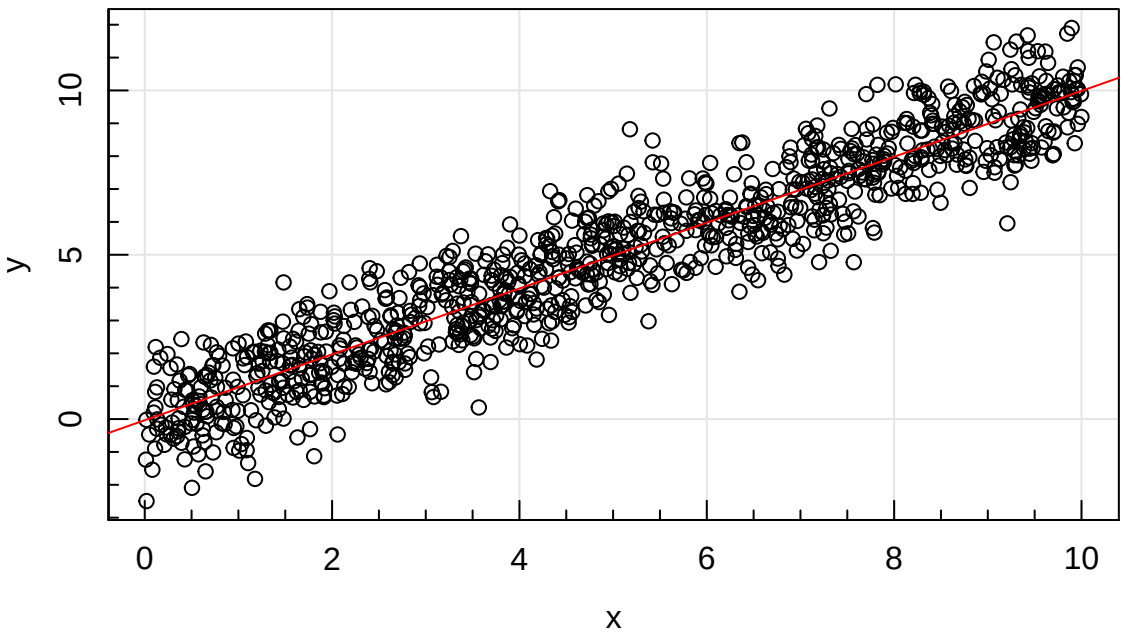
Creating Mock Data

To investigate how cross validation works in practice we should first make some fake data that we want to fit with a simple linear regression.

```
set.seed(1)
mock_lin = data.frame(x=runif(1e3,0,10))
mock_lin$y = mock_lin$x + rnorm(1e3)
```

We should see some linearly correlated data with some scatter along the y-axis:

```
magplot(mock_lin, xlab='x', ylab='y', grid=TRUE)
abline(lm(y~x, data=mock_lin), col='red')
```



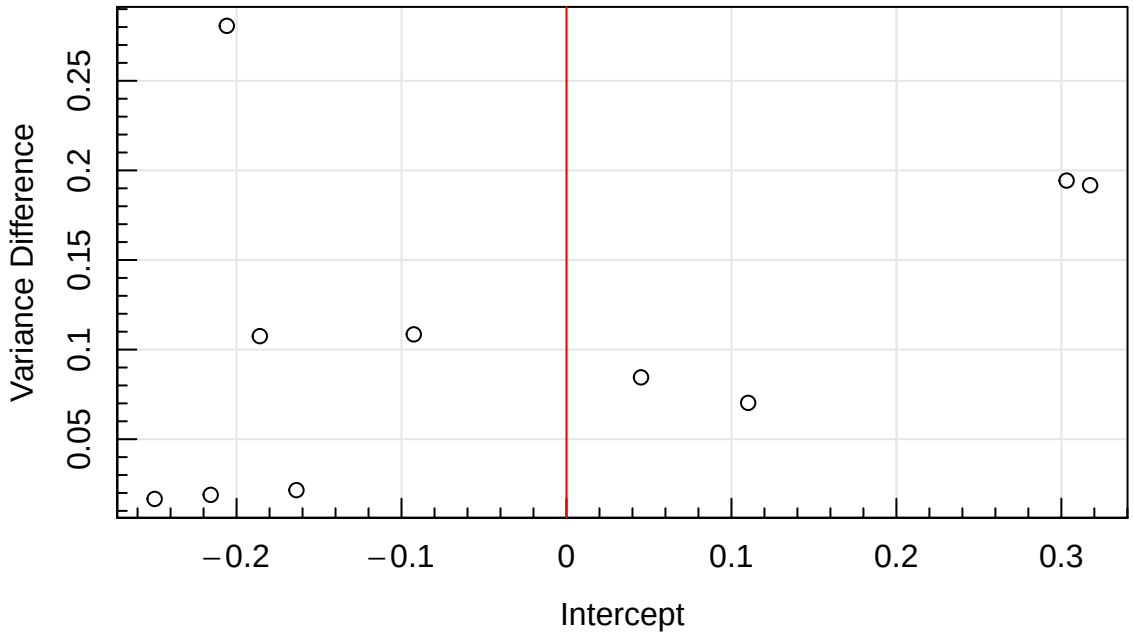
To see how we might cross validate a linear regression model we will refit the model 10 times to 10 different unique subsets of the data:

```
mock_lin$sub=1:10
lin_cv = foreach(i=1:10, .combine='rbind')%do%{
  fit_lm = lm(y~x, data=mock_lin[mock_lin$sub==i,])
  var_train = var(predict(fit_lm)-mock_lin[mock_lin$sub==i,'y'])
  var_validation = var(predict(fit_lm, newdata=mock_lin[mock_lin$sub!=i,c('x','y')]) - mock_lin[mock
  c(fit_lm$coefficients, var_train=var_train, var_validation=var_validation)
}
lin_cv = cbind(lin_cv, var_diff=abs(lin_cv[,'var_train']-lin_cv[,'var_validation']))
lin_cv
```

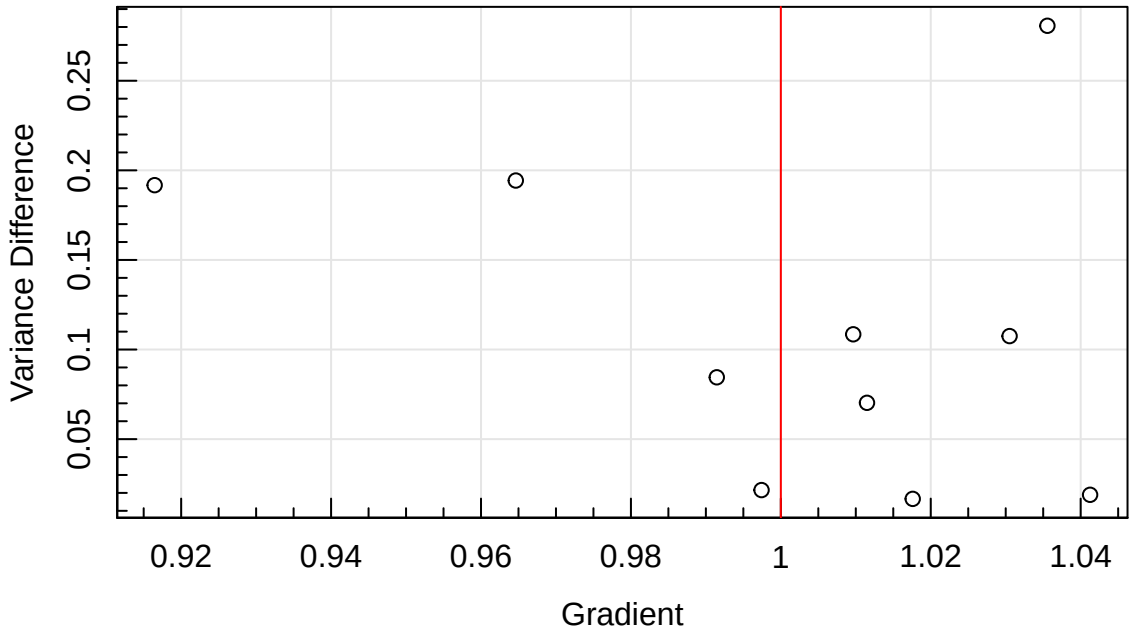
```
##            (Intercept)         x var_train var_validation   var_diff
## result.1  -0.16370477 0.9974249 1.0472249       1.068808 0.02158324
## result.2   0.11014013 1.0114922 1.0023769       1.072656 0.07027942
## result.3   0.04518510 0.9914553 1.1460556       1.061542 0.08451348
## result.4   0.31741851 0.9164624 0.9568061       1.148529 0.19172260
## result.5   0.30317528 0.9646436 1.2532640       1.058939 0.19432490
## result.6  -0.24966249 1.0175924 1.0839706       1.067299 0.01667109
## result.7  -0.09256287 1.0096551 0.9717455       1.080251 0.10850598
## result.8  -0.21567291 1.0412563 1.0988548       1.079904 0.01895072
## result.9  -0.18589879 1.0305144 1.1726090       1.065096 0.10751285
## result.10 -0.20591313 1.0355577 0.8255296       1.106207 0.28067758
```

The basic approach was to 'train' (in this case just do a linear regression fit) the data against 10% of the total data, and test the prediction against the remaining 90%. We do this 10 times for each distinct 10% sub-sample. We choose some aspect of the data- in this case the variance of the predictor residuals compared to the intrinsic y values. The best cross validation result is likely to be the one that has the least difference in this variance (not simply the one with the smallest trained variance, which we might naively pick otherwise), and it is pretty clear that the fits with the biggest differences between the training results and the validation set are poor fits, with coefficients pretty far away from the known inputs (0 for the intercept and 1 for the gradient in this case). As you can see above, the result with the smallest variance difference is not actually the one with parameters nearest to the truth, but they are very good.

```
magplot(lin_cv[,'(Intercept)'], lin_cv[,"var_diff"], xlab='Intercept', ylab='Variance Difference')
abline(v=0, col='red')
```




```
magplot(lin_cv[,'x'], lin_cv[,"var_diff"], xlab='Gradient', ylab='Variance Difference')
abline(v=1, col='red')
```



A Final Note

Confusingly in statistics and astronomy the terms jackknife, bootstrap and cross-validation are used somewhat interchangeably and even incorrectly. In the simplest terms:

- Jackknife and bootstrap techniques are used to estimate the errors for model parameters.
- Cross-validation is a method to assess the validity of the model itself.

Formally the latter is a hard problem in statistics, and strictly beyond the domain of Bayesian modelling, which ultimately is instead focussed on parameter inference *not* model selection (although various slightly ad-hoc methods exist to help with the latter).

Cross validation is also an important tool to prevent over-training and over-fitting in model analysis, which is a common curse in machine learning. A classic (and perhaps apocryphal) example of this is machine learning being used in the Cold War to identify Soviet versus NATO tanks. This was trained with an early (and now recognised as far too simple) form of neural network. The end result was that it worked perfectly after training, but poorly in practice. It turned out all it had learnt was that Soviet tanks tended to be on snow (images were predominantly white) and NATO tanks on earth (images were predominantly brown). This was merely a reflection of the training images, and almost no information about the tanks themselves existed in the network. Modern cross validation techniques might help protect against this, but clearly you also need to take care to choose a representative training set in the first place.

Resampling Exercises

Here you should experiment with different re-sampling techniques, and build confidence in how they behave in different regimes.

Modify the code above that made the object *mock_lin* to create different sizes of mock data (e.g. 100, 10^4, 10^6), and then try to assess the uncertainty in the intercept and gradient parameters using standard jackknife, sub-sampling and bootstrapping. Compare the estimated errors to your ground truth- which estimators converge best for a given amount of CPU time?

Whilst doing this, you can also use cross-validation to sanity check the quality of your fits.