CHAPTER 9
General Statistics

Any significant application of R includes statistics or models or graphics. This chapter addresses the statistics. Some recipes simply describe how to calculate a statistic, such as relative frequency. Most recipes involve statistical tests or confidence intervals. The statistical tests let you choose between two competing hypotheses; that paradigm is described next. Confidence intervals reflect the likely range of a population parameter and are calculated based on your data sample.

Null Hypotheses, Alternative Hypotheses, and p-Values

Many of the statistical tests in this chapter use a time-tested paradigm of statistical inference. In the paradigm, we have one or two data samples. We also have two competing hypotheses, either of which could reasonably be true.

One hypothesis, called the *null hypothesis*, is that *nothing happened*: the mean was unchanged; the treatment had no effect; you got the expected answer; the model did not improve; and so forth.

The other hypothesis, called the *alternative hypothesis*, is that *something happened*: the mean rose; the treatment improved the patients' health; you got an unexpected answer; the model fit better; and so forth.

We want to determine which hypothesis is more likely in light of the data. Here's how we do this:

1. To begin, we assume that the null hypothesis is true.
2. We calculate a test statistic. It could be something simple, such as the mean of the sample, or it could be quite complex. The critical requirement is that we must know the statistic's distribution. We might know the distribution of the sample mean, for example, by invoking the Central Limit Theorem.

3. From the statistic and its distribution we can calculate a p-value, the probability of a test statistic value as extreme or more extreme than the one we observed, while assuming that the null hypothesis is true.
4. If the p-value is too small, we have strong evidence against the null hypothesis. This is called *rejecting* the null hypothesis.
5. If the p-value is not small, then we have no such evidence. This is called *failing to reject* the null hypothesis.

There is one necessary decision here: when is a p-value "too small"?

In this book, we follow the common convention that we reject the null hypothesis when $p < 0.05$ and fail to reject it when $p > 0.05$. In statistical terminology, we choose a significance level of $\alpha = 0.05$ to define the border between strong evidence and insufficient evidence against the null hypothesis.

But the real answer is, "It depends." Your chosen significance level depends on your problem domain. The conventional limit of $p < 0.05$ works for many problems. In our work, the data is especially noisy and so we are often satisfied with $p < 0.10$. For someone working in high-risk areas, $p < 0.01$ or $p < 0.001$ might be necessary.

In the recipes, we mention which tests include a p-value so that you can compare the p-value against your chosen significance level of α. We worded the recipes to help you interpret the comparison. Here is the wording from Recipe 9.4, a test for the independence of two factors:

> Conventionally, a p-value of less than 0.05 indicates that the variables are likely not independent, whereas a p-value exceeding 0.05 fails to provide any such evidence.

This is a compact way of saying:

- The null hypothesis is that the variables are independent.
- The alternative hypothesis is that the variables are not independent.
- For $\alpha = 0.05$, if $p < 0.05$ then we reject the null hypothesis, giving strong evidence that the variables are not independent; if $p > 0.05$, we fail to reject the null hypothesis.
- You are free to choose your own α, of course, in which case your decision to reject or fail to reject might be different.

Remember, the recipe states the *informal interpretation* of the test results, not the rigorous mathematical interpretation. We use colloquial language in the hope that it will guide you toward a practical understanding and application of the test. If the precise semantics of hypothesis testing are critical for your work, we urge you to consult the

reference cited under See Also or one of the other fine textbooks on mathematical statistics.

Confidence Intervals

Hypothesis testing is a well-understood mathematical procedure, but it can be frustrating. First, the semantics are tricky. The test does not reach a definite, useful conclusion. You might get strong evidence against the null hypothesis, but that's all you'll get. Second, it does not give you a number, only evidence.

If you want numbers then use confidence intervals, which bound the estimate of a population parameter at a given level of confidence. Recipes in this chapter can calculate confidence intervals for means, medians, and proportions of a population.

For example, Recipe 9.9 calculates a 95% confidence interval for the population mean based on sample data. The interval is $97.16 < \mu < 103.98$, which means there is a 95% probability that the population's mean, μ, is between 97.16 and 103.98.

See Also

Statistical terminology and conventions can vary. This book generally follows the conventions of *Mathematical Statistics with Applications*, 6th ed., by Dennis Wackerly et al. (Duxbury Press). We recommend this book also for learning more about the statistical tests described in this chapter.

9.1 Summarizing Your Data

Problem

You want a basic statistical summary of your data.

Solution

The `summary` function gives some useful statistics for vectors, matrices, factors, and data frames:

```
summary(vec)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>     0.0     0.5     1.0     1.6     1.9    33.0
```

Discussion

The Solution exhibits the summary of a vector. The `1st Qu.` and `3rd Qu.` are the first and third quartile, respectively. Having both the median and mean is useful because you can quickly detect skew. The output in the Solution, for example, shows a mean

that is larger than the median; this indicates a possible skew to the right, as one would expect from a lognormal distribution.

The summary of a matrix works column by column. Here we see the summary of a matrix, `mat`, with three columns named `Samp1`, `Samp2`, and `Samp3`:

```
summary(mat)
#>      Samp1            Samp2             Samp3
#>  Min.   :  1.0   Min.   :-2.943   Min.   : 0.04
#>  1st Qu.: 25.8   1st Qu.:-0.774   1st Qu.: 0.39
#>  Median : 50.5   Median :-0.052   Median : 0.85
#>  Mean   : 50.5   Mean   :-0.067   Mean   : 1.60
#>  3rd Qu.: 75.2   3rd Qu.: 0.684   3rd Qu.: 2.12
#>  Max.   :100.0   Max.   : 2.150   Max.   :13.18
```

The summary of a factor gives counts:

```
summary(fac)
#> Maybe    No   Yes
#>    38    32    30
```

The summary of a character vector is pretty useless, giving just the vector length:

```
summary(char)
#>    Length     Class      Mode
#>       100 character character
```

The summary of a data frame incorporates all these features. It works column by column, giving an appropriate summary according to the column type. Numeric values receive a statistical summary and factors are counted (character strings are not summarized):

```
suburbs <- read_csv("./data/suburbs.txt")
summary(suburbs)
#>      city              county            state
#>  Length:17          Length:17          Length:17
#>  Class :character   Class :character   Class :character
#>  Mode  :character   Mode  :character   Mode  :character
#>
#>
#>
#>       pop
#>  Min.   :   5428
#>  1st Qu.:  72616
#>  Median :  83048
#>  Mean   : 249770
#>  3rd Qu.: 102746
#>  Max.   :2853114
```

The "summary" of a list is pretty funky: you get the data type of each list member. Here is a `summary` of a list of vectors:

```
summary(vec_list)
#>   Length Class  Mode
#> x 100    -none- numeric
#> y 100    -none- numeric
#> z 100    -none- character
```

To summarize the data inside a list of vectors, map `summary` to each list element:

```
library(purrr)
map(vec_list, summary)
#> $x
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>  -2.572  -0.686  -0.084  -0.043   0.660   2.413
#>
#> $y
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>  -1.752  -0.589   0.045   0.079   0.769   2.293
#>
#> $z
#>    Length     Class      Mode
#>       100 character character
```

Unfortunately, the `summary` function does not compute any measure of variability, such as standard deviation or median absolute deviation. This is a serious shortcoming, so we usually call `sd` or `mad` (mean absolute deviation) right after calling `summary`.

See Also

See Recipe 2.6 and Recipe 6.1.

9.2 Calculating Relative Frequencies

Problem

You want to count the relative frequency of certain observations in your sample.

Solution

Identify the interesting observations by using a logical expression; then use the `mean` function to calculate the fraction of observations it identifies. For example, given a vector *x*, you can find the relative frequency of positive values in this way:

```
mean(x > 3)
#> [1] 0.12
```

Discussion

A logical expression, such as $x > 3$, produces a vector of logical values (`TRUE` and `FALSE`), one for each element of *x*. The `mean` function converts those values to `1`s and

0s, respectively, and computes the average. This gives the fraction of values that are TRUE—in other words, the relative frequency of the interesting values. In the Solution, for example, that's the relative frequency of values greater than 3.

The concept here is pretty simple. The tricky part is dreaming up a suitable logical expression. Here are some examples:

`mean(lab == "NJ")`
: Fraction of `lab` values that are New Jersey

`mean(after > before)`
: Fraction of observations for which the effect increases

`mean(abs(x-mean(x)) > 2*sd(x))`
: Fraction of observations that exceed two standard deviations from the mean

`mean(diff(ts) > 0)`
: Fraction of observations in a time series that are larger than the previous observation

9.3 Tabulating Factors and Creating Contingency Tables

Problem

You want to tabulate one factor or build a contingency table from multiple factors.

Solution

The `table` function produces counts of one factor:

```
table(f1)
#> f1
#>  a  b  c  d  e
#> 14 23 24 21 18
```

It can also produce contingency tables (cross-tabulations) from two or more factors:

```
table(f1, f2)
#>    f2
#> f1  f  g  h
#>   a  6  4  4
#>   b  7  9  7
#>   c  4 11  9
#>   d  7  8  6
#>   e  5 10  3
```

`table` works for characters, too, not only factors:

```
t1 <- sample(letters[9:11], 100, replace = TRUE)
table(t1)
```

```
#> t1
#>  i  j  k
#> 20 40 40
```

Discussion

The `table` function counts the levels of one factor or characters, such as these counts of `initial` and `outcome` (which are factors):

```
set.seed(42)
initial <- factor(sample(c("Yes", "No", "Maybe"), 100, replace = TRUE))
outcome <- factor(sample(c("Pass", "Fail"), 100, replace = TRUE))

table(initial)
#> initial
#> Maybe    No   Yes
#>    39    31    30

table(outcome)
#> outcome
#> Fail Pass
#>   56   44
```

The greater power of `table` is in producing contingency tables, also known as cross-tabulations. Each cell in a contingency table counts how many times that row/column combination occurred:

```
table(initial, outcome)
#>        outcome
#> initial Fail Pass
#>   Maybe   23   16
#>   No      20   11
#>   Yes     13   17
```

This table shows that the combination of `initial = Yes` and `outcome = Fail` occurred 13 times, the combination of `initial = Yes` and `outcome = Pass` occurred 17 times, and so forth.

See Also

The `xtabs` function can also produce a contingency table. It has a formula interface, which some people prefer.

9.4 Testing Categorical Variables for Independence

Problem

You have two categorical variables that are represented by factors. You want to test them for independence using the chi-squared test.

Solution

Use the `table` function to produce a contingency table from the two factors. Then use the `summary` function to perform a chi-squared test of the contingency table. In this example we have two vectors of factor values, which we created in the prior recipe:

```
summary(table(initial, outcome))
#> Number of cases in table: 100
#> Number of factors: 2
#> Test for independence of all factors:
#>  Chisq = 3, df = 2, p-value = 0.2
```

The output includes a *p*-value. Conventionally, a *p*-value of less than 0.05 indicates that the variables are likely not independent, whereas a *p*-value exceeding 0.05 fails to provide any such evidence.

Discussion

This example performs a chi-squared test on the contingency table from Recipe 9.3, and yields a *p*-value of 0.2:

```
summary(table(initial, outcome))
#> Number of cases in table: 100
#> Number of factors: 2
#> Test for independence of all factors:
#>  Chisq = 3, df = 2, p-value = 0.2
```

The large *p*-value indicates that the two factors, `initial` and `outcome`, are probably independent. Practically speaking, we conclude there is no connection between the variables. This makes sense, as this example data was created by simply drawing random data using the `sample` function in the prior recipe.

See Also

The `chisq.test` function can also perform this test.

9.5 Calculating Quantiles (and Quartiles) of a Dataset

Problem

Given a fraction *f*, you want to know the corresponding quantile of your data. That is, you seek the observation *x* such that the fraction of observations below *x* is *f*.

Solution

Use the `quantile` function. The second argument is the fraction, *f*:

```
quantile(vec, 0.95)
#>  95%
#> 1.43
```

For quartiles, simply omit the second argument altogether:

```
quantile(vec)
#>      0%     25%     50%     75%    100%
#> -2.0247 -0.5915 -0.0693  0.4618  2.7019
```

Discussion

Suppose `vec` contains 1,000 observations between 0 and 1. The `quantile` function can tell you which observation delimits the lower 5% of the data:

```
vec <- runif(1000)
quantile(vec, .05)
#>     5%
#> 0.0451
```

The `quantile` documentation refers to the second argument as a "probability," which is natural when we think of probability as meaning relative frequency.

In true R style, the second argument can be a vector of probabilities; in this case, `quantile` returns a vector of corresponding quantiles, one for each probability:

```
quantile(vec, c(.05, .95))
#>     5%    95%
#> 0.0451 0.9363
```

That is a handy way to identify the middle 90% (in this case) of the observations.

If you omit the probabilities altogether, then R assumes you want the probabilities 0, 0.25, 0.50, 0.75, and 1.0—in other words, the quartiles:

```
quantile(vec)
#>       0%      25%      50%      75%     100%
#> 0.000405 0.235529 0.479543 0.737619 0.999379
```

Amazingly, the `quantile` function implements nine (yes, nine) different algorithms for computing quantiles. Study the help page before assuming that the default algorithm is the best one for you.

9.6 Inverting a Quantile

Problem

Given an observation x from your data, you want to know its corresponding quantile. That is, you want to know what fraction of the data is less than x.

Solution

Assuming your data is in a vector `vec`, compare the data against the observation and then use `mean` to compute the relative frequency of values less than *x*—say, 1.6 as per this example:

```
mean(vec < 1.6)
#> [1] 0.948
```

Discussion

The expression `vec < x` compares every element of `vec` against *x* and returns a vector of logical values, where the *n*th logical value is `TRUE` if `vec[`*n*`] < x`. The `mean` function converts those logical values to `0`s and `1`s: `0` for `FALSE` and `1` for `TRUE`. The average of all those `1`s and `0`s is the fraction of `vec` that is less than *x*, or the inverse quantile of *x*.

See Also

This is an application of the general approach described in Recipe 9.2.

9.7 Converting Data to z-Scores

Problem

You have a dataset, and you want to calculate the corresponding *z*-scores for all data elements. (This is sometimes called *normalizing* the data.)

Solution

Use the `scale` function:

```
scale(x)
#>          [,1]
#>  [1,]  0.8701
#>  [2,] -0.7133
#>  [3,] -1.0503
#>  [4,]  0.5790
#>  [5,] -0.6324
#>  [6,]  0.0991
#>  [7,]  2.1495
#>  [8,]  0.2481
#>  [9,] -0.8155
#> [10,] -0.7341
#> attr(,"scaled:center")
#> [1] 2.42
#> attr(,"scaled:scale")
#> [1] 2.11
```

This works for vectors, matrices, and data frames. In the case of a vector, `scale` returns the vector of normalized values. In the case of matrices and data frames, `scale` normalizes each column independently and returns columns of normalized values in a matrix.

Discussion

You might also want to normalize a single value y relative to a dataset x. You can do so by using vectorized operations as follows:

```
(y - mean(x)) / sd(x)
#> [1] -0.633
```

9.8 Testing the Mean of a Sample (t-Test)

Problem

You have a sample from a population. Given this sample, you want to know if the mean of the population could reasonably be a particular value m.

Solution

Apply the `t.test` function to the sample x with the argument `mu = m`:

```
t.test(x, mu = m)
```

The output includes a p-value. Conventionally, if $p < 0.05$ then the population mean is unlikely to be m, whereas $p > 0.05$ provides no such evidence.

If your sample size n is small, then the underlying population must be normally distributed in order to derive meaningful results from the t-test. A good rule of thumb is that "small" means $n < 30$.

Discussion

The t-test is a workhorse of statistics, and this is one of its basic uses: making inferences about a population mean from a sample. The following example simulates sampling from a normal population with mean $\mu = 100$. It uses the t-test to ask if the population mean could be 95, and `t.test` reports a p-value of 0.005:

```
x <- rnorm(75, mean = 100, sd = 15)
t.test(x, mu = 95)
#>
#>  One Sample t-test
#>
#> data:  x
#> t = 3, df = 70, p-value = 0.005
#> alternative hypothesis: true mean is not equal to 95
```

```
#> 95 percent confidence interval:
#>   96.5 103.0
#> sample estimates:
#> mean of x
#>      99.7
```

The *p*-value is small, so it's unlikely (based on the sample data) that 95 could be the mean of the population.

Informally, we could interpret the low *p*-value as follows. If the population mean were really 95, then the probability of observing our test statistic ($t = 2.8898$ or something more extreme) would be only 0.005. That is very improbable, yet that is the value we observed. Hence we conclude that the null hypothesis is wrong; therefore, the sample data does not support the claim that the population mean is 95.

In sharp contrast, testing for a mean of 100 gives a *p*-value of 0.9:

```
t.test(x, mu = 100)
#>
#>  One Sample t-test
#>
#> data:  x
#> t = -0.2, df = 70, p-value = 0.9
#> alternative hypothesis: true mean is not equal to 100
#> 95 percent confidence interval:
#>   96.5 103.0
#> sample estimates:
#> mean of x
#>      99.7
```

The large *p*-value indicates that the sample is consistent with assuming a population mean μ of 100. In statistical terms, the data does not provide evidence against the true mean being 100.

A common case is testing for a mean of zero. If you omit the `mu` argument, it defaults to `0`.

See Also

The `t.test` function is a many-splendored thing. See Recipe 9.9 and Recipe 9.15 for other uses.

9.9 Forming a Confidence Interval for a Mean

Problem

You have a sample from a population. Given that sample, you want to determine a confidence interval for the population's mean.

Solution

Apply the `t.test` function to your sample `x`:

```
t.test(x)
```

The output includes a confidence interval at the 95% confidence level. To see intervals at other levels, use the `conf.level` argument.

As in Recipe 9.8, if your sample size n is small, then the underlying population must be normally distributed for there to be a meaningful confidence interval. Again, a good rule of thumb is that "small" means $n < 30$.

Discussion

Applying the `t.test` function to a vector yields a lot of output. Buried in the output is a confidence interval:

```
t.test(x)
#>
#>  One Sample t-test
#>
#> data:  x
#> t = 50, df = 50, p-value <2e-16
#> alternative hypothesis: true mean is not equal to 0
#> 95 percent confidence interval:
#>   94.2 101.5
#> sample estimates:
#> mean of x
#>      97.9
```

In this example, the confidence interval is approximately $94.2 < \mu < 101.5$, which is sometimes written simply as (94.2, 101.5).

We can raise the confidence level to 99% by setting `conf.level=0.99`:

```
t.test(x, conf.level = 0.99)
#>
#>  One Sample t-test
#>
#> data:  x
#> t = 50, df = 50, p-value <2e-16
#> alternative hypothesis: true mean is not equal to 0
#> 99 percent confidence interval:
#>   92.9 102.8
#> sample estimates:
#> mean of x
#>      97.9
```

That change widens the confidence interval to $92.9 < \mu < 102.8$.

9.10 Forming a Confidence Interval for a Median

Problem

You have a data sample, and you want to know the confidence interval for the median.

Solution

Use the `wilcox.test` function, setting `conf.int=TRUE`:

```
wilcox.test(x, conf.int = TRUE)
```

The output will contain a confidence interval for the median.

Discussion

The procedure for calculating the confidence interval of a mean is well defined and widely known. The same is not true for the median, unfortunately. There are several procedures for calculating the median's confidence interval. None of them is "the" procedure, but the Wilcoxon signed rank test is pretty standard.

The `wilcox.test` function implements that procedure. Buried in the output is the 95% confidence interval, which is approximately (–0.102, 0.646) in this case:

```
wilcox.test(x, conf.int = TRUE)
#>
#>  Wilcoxon signed rank test
#>
#> data:  x
#> V = 200, p-value = 0.1
#> alternative hypothesis: true location is not equal to 0
#> 95 percent confidence interval:
#>  -0.102  0.646
#> sample estimates:
#> (pseudo)median
#>          0.311
```

You can change the confidence level by setting `conf.level`, such as `conf.level=0.99` or other such values.

The output also includes something called the *pseudomedian*, which is defined on the help page. Don't assume it equals the median; they are different:

```
median(x)
#> [1] 0.314
```

See Also

The bootstrap procedure is also useful for estimating the median's confidence interval; see Recipe 8.5 and Recipe 13.8.

9.11 Testing a Sample Proportion

Problem

You have a sample of values from a population consisting of successes and failures. You believe the true proportion of successes is p, and you want to test that hypothesis using the sample data.

Solution

Use the `prop.test` function. Suppose the sample size is n and the sample contains x successes:

```
prop.test(x, n, p)
```

The output includes a p-value. Conventionally, a p-value of less than 0.05 indicates that the true proportion is unlikely to be p, whereas a p-value exceeding 0.05 fails to provide such evidence.

Discussion

Suppose you encounter some loudmouthed fan of the Chicago Cubs early in the baseball season. The Cubs have played 20 games and won 11 of them, or 55% of their games. Based on that evidence, the fan is "very confident" that the Cubs will win more than half of their games this year. Should they be that confident?

The `prop.test` function can evaluate the fan's logic. Here, the number of observations is $n = 20$, the number of successes is $x = 11$, and p is the true probability of winning a game. We want to know whether it is reasonable to conclude, based on the data, that $p > 0.5$. Normally, `prop.test` would check for $p \neq 0.05$, but we can check for $p > 0.5$ instead by setting `alternative="greater"`:

```
prop.test(11, 20, 0.5, alternative = "greater")
#>
#>  1-sample proportions test with continuity correction
#>
#> data:  11 out of 20, null probability 0.5
#> X-squared = 0.05, df = 1, p-value = 0.4
#> alternative hypothesis: true p is greater than 0.5
#> 95 percent confidence interval:
#>  0.35 1.00
#> sample estimates:
```

```
#>    p
#> 0.55
```

The `prop.test` output shows a large p-value, 0.55, so we cannot reject the null hypothesis; that is, we cannot reasonably conclude that p is greater than 1/2. The Cubs fan is being overly confident based on too little data. No surprise there.

9.12 Forming a Confidence Interval for a Proportion

Problem

You have a sample of values from a population consisting of successes and failures. Based on the sample data, you want to form a confidence interval for the population's proportion of successes.

Solution

Use the `prop.test` function. Suppose the sample size is n and the sample contains x successes:

```
prop.test(x, n)
```

The function output includes the confidence interval for p.

Discussion

We subscribe to a stock market newsletter that is well written, but includes a section purporting to identify stocks that are likely to rise. It does this by looking for a certain pattern in the stock price. It recently reported, for example, that a certain stock was following the pattern. It also reported that the stock rose six times after the last nine times that pattern occurred. The writers concluded that the probability of the stock rising again was therefore 6/9, or 66.7%.

Using `prop.test`, we can obtain the confidence interval for the true proportion of times the stock rises after the pattern. Here, the number of observations is $n = 9$ and the number of successes is $x = 6$. The output shows a confidence interval of (0.309, 0.910) at the 95% confidence level:

```
prop.test(6, 9)
#> Warning in prop.test(6, 9): Chi-squared approximation may be incorrect
#>
#>  1-sample proportions test with continuity correction
#>
#> data:  6 out of 9, null probability 0.5
#> X-squared = 0.4, df = 1, p-value = 0.5
#> alternative hypothesis: true p is not equal to 0.5
#> 95 percent confidence interval:
#>  0.309 0.910
```

```
#> sample estimates:
#>     p
#> 0.667
```

The writers are pretty foolish to say the probability of the stock rising is 66.7%. They could be leading their readers into a very bad bet.

By default, `prop.test` calculates a confidence interval at the 95% confidence level. Use the `conf.level` argument for other confidence levels:

```
prop.test(x, n, p, conf.level = 0.99)   # 99% confidence level
```

See Also

See Recipe 9.11.

9.13 Testing for Normality

Problem

You want a statistical test to determine whether your data sample is from a normally distributed population.

Solution

Use the `shapiro.test` function:

```
shapiro.test(x)
```

The output includes a p-value. Conventionally, $p < 0.05$ indicates that the population is likely not normally distributed, whereas $p > 0.05$ provides no such evidence.

Discussion

This example reports a p-value of 0.4 for `x`:

```
shapiro.test(x)
#>
#>  Shapiro-Wilk normality test
#>
#> data:  x
#> W = 1, p-value = 0.4
```

The large p-value suggests the underlying population could be normally distributed. The next example reports a very small p-value for `y`, so it is unlikely that this sample came from a normal population:

```
shapiro.test(y)
#>
#>  Shapiro-Wilk normality test
```

```
#>
#> data:  y
#> W = 0.7, p-value = 7e-13
```

We have highlighted the Shapiro–Wilk test because it is a standard R function. You can also install the package `nortest`, which is dedicated entirely to tests for normality. This package includes the following tests:

- Anderson–Darling (`ad.test`)
- Cramer–von Mises (`cvm.test`)
- Lilliefors (`lillie.test`)
- Pearson chi-squared for the composite hypothesis of normality (`pearson.test`)
- Shapiro–Francia (`sf.test`)

The problem with all of these is their null hypothesis: they all assume that the population is normally distributed until proven otherwise. As a result, the population must be decidedly nonnormal before the test reports a small *p*-value and you can reject that null hypothesis. That makes the tests quite conservative, tending to err on the side of normality.

Instead of depending solely upon a statistical test, we suggest also using histograms (Recipe 10.19) and quantile-quantile plots (Recipe 10.21) to evaluate the normality of any data. Are the tails too fat? Is the peak too peaked? Your judgment is likely better than a single statistical test.

See Also

See Recipe 3.10 for how to install the `nortest` package.

9.14 Testing for Runs

Problem

Your data is a sequence of binary values: yes/no, 0/1, true/false, or other two-valued data. You want to know: is the sequence random?

Solution

The `tseries` package contains the `runs.test` function, which checks a sequence for randomness. The sequence should be a factor with two levels:

```
library(tseries)
runs.test(as.factor(s))
```

The `runs.test` function reports a *p*-value. Conventionally, a *p*-value of less than 0.05 indicates that the sequence is likely not random, whereas a *p*-value exceeding 0.05 provides no such evidence.

Discussion

A run is a subsequence composed of identical values, such as all `1`s or all `0`s. A random sequence should be properly jumbled up, without too many runs. It shouldn't contain too *few* runs, either—a sequence of perfectly alternating values (`0`, `1`, `0`, `1`, `0`, `1`, …) contains no runs, but would you say that it's random?

The `runs.test` function checks the number of runs in your sequence. If there are too many or too few, it reports a small *p*-value.

This first example generates a random sequence of `0`s and `1`s and then tests the sequence for runs. Not surprisingly, `runs.test` reports a large *p*-value, indicating the sequence is likely random:

```
s <- sample(c(0, 1), 100, replace = T)
runs.test(as.factor(s))
#>
#>  Runs Test
#>
#> data:  as.factor(s)
#> Standard Normal = 0.1, p-value = 0.9
#> alternative hypothesis: two.sided
```

This next sequence, however, consists of three runs and so the reported *p*-value is quite low:

```
s <- c(0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0)
runs.test(as.factor(s))
#>
#>  Runs Test
#>
#> data:  as.factor(s)
#> Standard Normal = -2, p-value = 0.02
#> alternative hypothesis: two.sided
```

See Also

See Recipe 5.4 and Recipe 8.6.

9.15 Comparing the Means of Two Samples

Problem

You have one sample each from two populations. You want to know if the two populations could have the same mean.

Solution

Perform a *t*-test by calling the `t.test` function:

```
t.test(x, y)
```

By default, `t.test` assumes that your observations are not paired. If the observations are paired (i.e., if each x_i is paired with one y_i), then specify `paired=TRUE`:

```
t.test(x, y, paired = TRUE)
```

In either case, `t.test` will compute a *p*-value. Conventionally, if $p < 0.05$ then the means are likely different, whereas $p > 0.05$ provides no such evidence:

- If either sample size is small, then the populations must be normally distributed. Here, "small" means fewer than 20 data points.
- If the two populations have the same variance, specify `var.equal=TRUE` to obtain a less conservative test.

Discussion

We often use the *t*-test to get a quick sense of the difference between two population means. It requires that the samples be large enough (i.e., both samples have 20 or more observations) or that the underlying populations be normally distributed. We don't take the "normally distributed" part too literally. Being bell-shaped and reasonably symmetrical should be good enough.

A key distinction here is whether or not your data contains paired observations, since the results may differ in the two cases. Suppose we want to know if drinking coffee in the morning improves scores on SATs. We could run the experiment two ways:

- Randomly select one group of people. Give them the SAT twice, once with morning coffee and once without morning coffee. For each person, we will have two SAT scores. These are paired observations.
- Randomly select two groups of people. One group has a cup of morning coffee and takes the SAT. The other group just takes the test. We have a score for each person, but the scores are not paired in any way.

Statistically, these experiments are quite different. In experiment 1, there are two observations for each person (caffeinated and not) and they are not statistically independent. In experiment 2, the observations are independent.

If you have paired observations (experiment 1) and erroneously analyze them as unpaired observations (experiment 2), then you could get this result with a *p*-value of 0.3:

```
load("./data/sat.rdata")
t.test(x, y)
#>
#>  Welch Two Sample t-test
#>
#> data:  x and y
#> t = -1, df = 200, p-value = 0.3
#> alternative hypothesis: true difference in means is not equal to 0
#> 95 percent confidence interval:
#>  -46.4  16.2
#> sample estimates:
#> mean of x mean of y
#>      1054      1069
```

The large *p*-value forces you to conclude there is no difference between the groups. Contrast that result with the one that follows from analyzing the same data but correctly identifying it as paired:

```
t.test(x, y, paired = TRUE)
#>
#>  Paired t-test
#>
#> data:  x and y
#> t = -20, df = 100, p-value <2e-16
#> alternative hypothesis: true difference in means is not equal to 0
#> 95 percent confidence interval:
#>  -16.8 -13.5
#> sample estimates:
#> mean of the differences
#>                   -15.1
```

The *p*-value plummets to 2e-16, and we reach the exactly opposite conclusion.

See Also

If the populations are not normally distributed (bell-shaped) and either sample is small, consider using the Wilcoxon–Mann–Whitney test described in Recipe 9.16.

9.16 Comparing the Locations of Two Samples Nonparametrically

Problem

You have samples from two populations. You don't know the distribution of the populations, but you know they have similar shapes. You want to know: is one population shifted to the left or right compared with the other?

Solution

You can use a nonparametric test, the Wilcoxon–Mann–Whitney test, which is implemented by the `wilcox.test` function. For paired observations (every x_i is paired with y_i), set `paired=TRUE`:

```
wilcox.test(x, y, paired = TRUE)
```

For unpaired observations, let `paired` default to `FALSE`:

```
wilcox.test(x, y)
```

The test output includes a p-value. Conventionally, a p-value of less than 0.05 indicates that the second population is likely shifted left or right with respect to the first population, whereas a p-value exceeding 0.05 provides no such evidence.

Discussion

When we stop making assumptions regarding the distributions of populations, we enter the world of nonparametric statistics. The Wilcoxon–Mann–Whitney test is nonparametric and so can be applied to more datasets than the t-test, which requires that the data be normally distributed (for small samples). This test's only assumption is that the two populations have the same shape.

In this recipe, we are asking: is the second population shifted left or right with respect to the first? This is similar to asking whether the average of the second population is smaller or larger than that of the first. However, the Wilcoxon–Mann–Whitney test answers a different question: it tells us whether the central locations of the two populations are significantly different or, equivalently, whether their relative frequencies are different.

Suppose we randomly select a group of employees and ask each one to complete the same task under two different circumstances: under favorable conditions and under unfavorable conditions, such as a noisy environment. We measure their completion times under both conditions, so we have two measurements for each employee. We want to know if the two times are significantly different, but we can't assume they are normally distributed.

The observations are paired, so we must set `paired=TRUE`:

```
load(file = "./data/workers.rdata")
wilcox.test(fav, unfav, paired = TRUE)
#>
#>  Wilcoxon signed rank test
#>
#> data:  fav and unfav
#> V = 10, p-value = 1e-04
#> alternative hypothesis: true location shift is not equal to 0
```

The *p*-value is essentially zero. Statistically speaking, we reject the assumption that the completion times were equal. Practically speaking, it's reasonable to conclude that the times were different.

In this example, setting `paired=TRUE` is critical. Treating the data as unpaired would be wrong because the observations are not independent, and this in turn would produce bogus results. Running the example with `paired=FALSE` produces a *p*-value of 0.1022, which leads to the wrong conclusion.

See Also

See Recipe 9.15 for the parametric test.

9.17 Testing a Correlation for Significance

Problem

You calculated the correlation between two variables, but you don't know if the correlation is statistically significant.

Solution

The `cor.test` function can calculate both the *p*-value and the confidence interval of the correlation. If the variables came from normally distributed populations then use the default measure of correlation, which is the Pearson method:

```
cor.test(x, y)
```

For nonnormal populations, use the Spearman method instead:

```
cor.test(x, y, method = "spearman")
```

The function returns several values, including the *p*-value from the test of significance. Conventionally, $p < 0.05$ indicates that the correlation is likely significant, whereas $p > 0.05$ indicates it is not.

Discussion

In our experience, people often fail to check a correlation for significance. In fact, many people are unaware that a correlation can be insignificant. They jam their data into a computer, calculate the correlation, and blindly believe the result. However, they should ask themselves: Was there enough data? Is the magnitude of the correlation large enough? Fortunately, the `cor.test` function answers those questions.

Suppose we have two vectors, `x` and `y`, with values from normal populations. We might be very pleased that their correlation is greater than 0.75:

```
cor(x, y)
#> [1] 0.751
```

But that is naïve. If we run `cor.test`, it reports a relatively large *p*-value of 0.09:

```
cor.test(x, y)
#>
#>  Pearson's product-moment correlation
#>
#> data:  x and y
#> t = 2, df = 4, p-value = 0.09
#> alternative hypothesis: true correlation is not equal to 0
#> 95 percent confidence interval:
#>  -0.155  0.971
#> sample estimates:
#>   cor
#> 0.751
```

The *p*-value is above the conventional threshold of 0.05, so we conclude that the correlation is unlikely to be significant.

You can also check the correlation by using the confidence interval. In this example, the confidence interval is (–0.155, 0.971). The interval contains zero and so it is possible that the correlation is zero, in which case there would be no correlation. Again, you could not be confident that the reported correlation is significant.

The `cor.test` output also includes the point estimate reported by `cor` (at the bottom, labeled "sample estimates"), saving you the additional step of running `cor`.

By default, `cor.test` calculates the Pearson correlation, which assumes that the underlying populations are normally distributed. The Spearman method makes no such assumption because it is nonparametric. Use `method="Spearman"` when working with nonnormal data.

See Also

See Recipe 2.6 for calculating simple correlations.

9.18 Testing Groups for Equal Proportions

Problem

You have samples from two or more groups. The groups' elements are binary-valued: either success or failure. You want to know if the groups have equal proportions of successes.

Solution

Use the `prop.test` function with two vector arguments:

```
ns <- c(48, 64)
nt <- c(100, 100)
prop.test(ns, nt)
#>
#>  2-sample test for equality of proportions with continuity
#>  correction
#>
#> data:  ns out of nt
#> X-squared = 5, df = 1, p-value = 0.03
#> alternative hypothesis: two.sided
#> 95 percent confidence interval:
#>  -0.3058 -0.0142
#> sample estimates:
#> prop 1 prop 2
#>   0.48   0.64
```

These are parallel vectors. The first vector, `ns`, gives the number of successes in each group. The second vector, `nt`, gives the size of the corresponding group (often called the *number of trials*).

The output includes a *p*-value. Conventionally, a *p*-value of less than 0.05 indicates that it is likely the groups' proportions are different, whereas a *p*-value exceeding 0.05 provides no such evidence.

Discussion

In Recipe 9.11, we tested a proportion based on one sample. Here, we have samples from multiple groups and want to compare the proportions in the underlying groups.

One of the authors recently taught statistics to 38 students and awarded a grade of A to 14 of them. A colleague taught the same class to 40 students and awarded an A to only 10. We wanted to know: was the author fostering grade inflation by awarding significantly more A grades than the other teacher did?

We used `prop.test`. "Success" means awarding an A, so the vector of successes contains two elements, the number awarded by the author and the number awarded by the colleague:

```
successes <- c(14, 10)
```

The number of trials is the number of students in the corresponding class:

```
trials <- c(38, 40)
```

The `prop.test` output yields a *p*-value of 0.4:

```
prop.test(successes, trials)
#>
#>  2-sample test for equality of proportions with continuity
#>  correction
#>
#> data:  successes out of trials
#> X-squared = 0.8, df = 1, p-value = 0.4
#> alternative hypothesis: two.sided
#> 95 percent confidence interval:
#>  -0.111  0.348
#> sample estimates:
#> prop 1 prop 2
#>  0.368  0.250
```

The relatively large *p*-value means that we cannot reject the null hypothesis: the evidence does not suggest any difference between the teachers' grading.

See Also

See Recipe 9.11.

9.19 Performing Pairwise Comparisons Between Group Means

Problem

You have several samples, and you want to perform a pairwise comparison between the sample means. That is, you want to compare the mean of every sample against the mean of every other sample.

Solution

Place all data into one vector and create a parallel factor to identify the groups. Use `pairwise.t.test` to perform the pairwise comparison of means:

```
pairwise.t.test(x, f)   # x is the data, f is the grouping factor
```

The output contains a table of p-values, one for each pair of groups. Conventionally, if $p < 0.05$ then the two groups likely have different means, whereas $p > 0.05$ provides no such evidence.

Discussion

This is more complicated than Recipe 9.15, where we compared the means of two samples. Here we have several samples and want to compare the mean of every sample against the mean of every other sample.

Statistically speaking, pairwise comparisons are tricky. It is not the same as simply performing a t-test on every possible pair. The p-values must be adjusted, as otherwise you will get an overly optimistic result. The help pages for `pairwise.t.test` and `p.adjust` describe the adjustment algorithms available in R. Anyone doing serious pairwise comparisons is urged to review the help pages and consult a good textbook on the subject.

Suppose we are using a larger sample of the data from Recipe 5.5, where we combined data for freshmen, sophomores, and juniors into a data frame called `comb`. The data frame has two columns: the data in a column called `values`, and the grouping factor in a column called `ind`. We can use `pairwise.t.test` to perform pairwise comparisons between the groups:

```
pairwise.t.test(comb$values, comb$ind)
#>
#>  Pairwise comparisons using t-tests with pooled SD
#>
#> data:  comb$values and comb$ind
#>
#>      fresh soph
#> soph 0.001 -
#> jrs  3e-04 0.592
#>
#> P value adjustment method: holm
```

Notice the table of p-values. The comparisons of juniors versus freshmen and of sophomores versus freshmen produced small p-values: 0.001 and 0.0003, respectively. We can conclude there are significant differences between those groups. However, the comparison of sophomores versus juniors produced a (relatively) large p-value of 0.592, so they are not significantly different.

See Also

See Recipe 5.5 and Recipe 9.15.

9.20 Testing Two Samples for the Same Distribution

Problem

You have two samples, and you are wondering: did they come from the same distribution?

Solution

The Kolmogorov–Smirnov test compares two samples and tests them for being drawn from the same distribution. The `ks.test` function implements that test:

```
ks.test(x, y)
```

The output includes a *p*-value. Conventionally, a *p*-value of less than 0.05 indicates that the two samples (*x* and *y*) were drawn from different distributions, whereas a *p*-value exceeding 0.05 provides no such evidence.

Discussion

The Kolmogorov–Smirnov test is wonderful for two reasons. First, it is a nonparametric test and so you needn't make any assumptions regarding the underlying distributions: it works for all distributions. Second, it checks the location, dispersion, and shape of the populations, based on the samples. If these characteristics disagree then the test will detect that, allowing you to conclude that the underlying distributions are different.

Suppose we suspect that the vectors `x` and `y` come from differing distributions. Here, `ks.test` reports a *p*-value of 0.04:

```
ks.test(x, y)
#>
#>  Two-sample Kolmogorov-Smirnov test
#>
#> data:  x and y
#> D = 0.2, p-value = 0.04
#> alternative hypothesis: two-sided
```

From the small *p*-value we can conclude that the samples are from different distributions. However, when we test `x` against another sample, `z`, the *p*-value is much larger (0.6); this suggests that `x` and `z` could have the same underlying distribution:

```
z <- rnorm(100, mean = 4, sd = 6)
ks.test(x, z)
#>
#>  Two-sample Kolmogorov-Smirnov test
#>
#> data:  x and z
```

```
#> D = 0.1, p-value = 0.6
#> alternative hypothesis: two-sided
```