Chapter 7. Hypothesis and Inference

> It is the mark of a truly intelligent person to be moved by statistics.
>
> George Bernard Shaw

What will we do with all this statistics and probability theory? The *science* part of data science frequently involves forming and testing *hypotheses* about our data and the processes that generate it.

Statistical Hypothesis Testing

Often, as data scientists, we'll want to test whether a certain hypothesis is likely to be true. For our purposes, hypotheses are assertions like "this coin is fair" or "data scientists prefer Python to R" or "people are more likely to navigate away from the page without ever reading the content if we pop up an irritating interstitial advertisement with a tiny, hard-to-find close button" that can be translated into statistics about data. Under various assumptions, those statistics can be thought of as observations of random variables from known distributions, which allows us to make statements about how likely those assumptions are to hold.

In the classical setup, we have a *null hypothesis* H_0 that represents some default position, and some alternative hypothesis H_1 that we'd like to compare it with. We use statistics to decide whether we can reject H_0 as false or not. This will probably make more sense with an example.

Example: Flipping a Coin

Imagine we have a coin and we want to test whether it's fair. We'll make the assumption that the coin has some probability p of landing heads, and so our null hypothesis is that the coin is fair — that is, that $p = 0.5$. We'll test this against the alternative hypothesis $p \neq 0.5$.

In particular, our test will involve flipping the coin some number n times and counting the number of heads X. Each coin flip is a Bernoulli trial, which means that X is a Binomial(n,p) random variable, which (as we saw in Chapter 6) we can approximate using the normal distribution:

```
def normal_approximation_to_binomial(n, p):
    """finds mu and sigma corresponding to a Binomial(n, p)"""
    mu = p * n
    sigma = math.sqrt(p * (1 - p) * n)
    return mu, sigma
```

Whenever a random variable follows a normal distribution, we can use `normal_cdf` to figure out the probability that its realized value lies within (or outside) a particular interval:

```
# the normal cdf _is_ the probability the variable is below a threshold
normal_probability_below = normal_cdf

# it's above the threshold if it's not below the threshold
def normal_probability_above(lo, mu=0, sigma=1):
    return 1 - normal_cdf(lo, mu, sigma)

# it's between if it's less than hi, but not less than lo
def normal_probability_between(lo, hi, mu=0, sigma=1):
    return normal_cdf(hi, mu, sigma) - normal_cdf(lo, mu, sigma)

# it's outside if it's not between
def normal_probability_outside(lo, hi, mu=0, sigma=1):
    return 1 - normal_probability_between(lo, hi, mu, sigma)
```

We can also do the reverse — find either the nontail region or the (symmetric) interval around the mean that accounts for a certain level of likelihood. For example, if we want to find an interval centered at the mean and containing 60% probability, then we find the cutoffs where the upper and lower tails each contain 20% of the probability (leaving 60%):

```
def normal_upper_bound(probability, mu=0, sigma=1):
    """returns the z for which P(Z <= z) = probability"""
    return inverse_normal_cdf(probability, mu, sigma)

def normal_lower_bound(probability, mu=0, sigma=1):
    """returns the z for which P(Z >= z) = probability"""
    return inverse_normal_cdf(1 - probability, mu, sigma)

def normal_two_sided_bounds(probability, mu=0, sigma=1):
    """returns the symmetric (about the mean) bounds
    that contain the specified probability"""
    tail_probability = (1 - probability) / 2

    # upper bound should have tail_probability above it
    upper_bound = normal_lower_bound(tail_probability, mu, sigma)
```

```
    # lower bound should have tail_probability below it
    lower_bound = normal_upper_bound(tail_probability, mu, sigma)

    return lower_bound, upper_bound
```

In particular, let's say that we choose to flip the coin $n = 1000$ times. If our hypothesis of fairness is true, *X* should be distributed approximately normally with mean 50 and standard deviation 15.8:

```
mu_0, sigma_0 = normal_approximation_to_binomial(1000, 0.5)
```

We need to make a decision about *significance* — how willing we are to make a *type 1 error* ("false positive"), in which we reject H_0 even though it's true. For reasons lost to the annals of history, this willingness is often set at 5% or 1%. Let's choose 5%.

Consider the test that rejects H_0 if *X* falls outside the bounds given by:

```
normal_two_sided_bounds(0.95, mu_0, sigma_0)   # (469, 531)
```

Assuming *p* really equals 0.5 (i.e., H_0 is true), there is just a 5% chance we observe an *X* that lies outside this interval, which is the exact significance we wanted. Said differently, if H_0 is true, then, approximately 19 times out of 20, this test will give the correct result.

We are also often interested in the *power* of a test, which is the probability of not making a *type 2 error*, in which we fail to reject H_0 even though it's false. In order to measure this, we have to specify what exactly H_0 being false *means*. (Knowing merely that *p* is *not* 0.5 doesn't give you a ton of information about the distribution of *X*.) In particular, let's check what happens if *p* is really 0.55, so that the coin is slightly biased toward heads.

In that case, we can calculate the power of the test with:

```
# 95% bounds based on assumption p is 0.5
lo, hi = normal_two_sided_bounds(0.95, mu_0, sigma_0)

# actual mu and sigma based on p = 0.55
mu_1, sigma_1 = normal_approximation_to_binomial(1000, 0.55)

# a type 2 error means we fail to reject the null hypothesis
# which will happen when X is still in our original interval
type_2_probability = normal_probability_between(lo, hi, mu_1, sigma_1)
power = 1 - type_2_probability      # 0.887
```

Imagine instead that our null hypothesis was that the coin is not biased toward heads, or that $p \leq 0.5$. In that case we want a *one-sided test* that rejects the null hypothesis when *X* is much larger than 50 but not when *X* is smaller than 50. So a 5%-significance test involves using `normal_probability_below` to find the cutoff below which 95% of the probability lies:

```
hi = normal_upper_bound(0.95, mu_0, sigma_0)
# is 526 (< 531, since we need more probability in the upper tail)

type_2_probability = normal_probability_below(hi, mu_1, sigma_1)
```

```
power = 1 - type_2_probability      # 0.936
```

This is a more powerful test, since it no longer rejects H_0 when X is below 469 (which is very unlikely to happen if H_1 is true) and instead rejects H_0 when X is between 526 and 531 (which is somewhat likely to happen if H_1 is true). === p-values

An alternative way of thinking about the preceding test involves *p-values*. Instead of choosing bounds based on some probability cutoff, we compute the probability — assuming H_0 is true — that we would see a value at least as extreme as the one we actually observed.

For our two-sided test of whether the coin is fair, we compute:

```
def two_sided_p_value(x, mu=0, sigma=1):
    if x >= mu:
        # if x is greater than the mean, the tail is what's greater than x
        return 2 * normal_probability_above(x, mu, sigma)
    else:
        # if x is less than the mean, the tail is what's less than x
        return 2 * normal_probability_below(x, mu, sigma)
```

If we were to see 530 heads, we would compute:

```
two_sided_p_value(529.5, mu_0, sigma_0)   # 0.062
```

> **NOTE**
>
> Why did we use 529.5 instead of 530? This is what's called a *continuity correction*. It reflects the fact that `normal_probability_between(529.5, 530.5, mu_0, sigma_0)` is a better estimate of the probability of seeing 530 heads than `normal_probability_between(530, 531, mu_0, sigma_0)` is.
>
> Correspondingly, `normal_probability_above(529.5, mu_0, sigma_0)` is a better estimate of the probability of seeing at least 530 heads. You may have noticed that we also used this in the code that produced Figure 6-4.

One way to convince yourself that this is a sensible estimate is with a simulation:

```
extreme_value_count = 0
for _ in range(100000):
    num_heads = sum(1 if random.random() < 0.5 else 0     # count # of heads
                    for _ in range(1000))                 # in 1000 flips
    if num_heads >= 530 or num_heads <= 470:              # and count how often
        extreme_value_count += 1                          # the # is 'extreme'

print extreme_value_count / 100000   # 0.062
```

Since the *p*-value is greater than our 5% significance, we don't reject the null. If we instead saw 532 heads, the *p*-value would be:

```
two_sided_p_value(531.5, mu_0, sigma_0)   # 0.0463
```

which is smaller than the 5% significance, which means we would reject the null. It's the exact same test as before. It's just a different way of approaching the statistics.

Similarly, we would have:

```
upper_p_value = normal_probability_above
lower_p_value = normal_probability_below
```

For our one-sided test, if we saw 525 heads we would compute:

```
upper_p_value(524.5, mu_0, sigma_0) # 0.061
```

which means we wouldn't reject the null. If we saw 527 heads, the computation would be:

```
upper_p_value(526.5, mu_0, sigma_0) # 0.047
```

and we would reject the null.

WARNING

Make sure your data is roughly normally distributed before using `normal_probability_above` to compute p-values. The annals of bad data science are filled with examples of people opining that the chance of some observed event occurring at random is one in a million, when what they really mean is "the chance, assuming the data is distributed normally," which is pretty meaningless if the data isn't.

There are various statistical tests for normality, but even plotting the data is a good start.

Confidence Intervals

We've been testing hypotheses about the value of the heads probability *p*, which is a *parameter* of the unknown "heads" distribution. When this is the case, a third approach is to construct a *confidence interval* around the observed value of the parameter.

For example, we can estimate the probability of the unfair coin by looking at the average value of the Bernoulli variables corresponding to each flip — 1 if heads, 0 if tails. If we observe 525 heads out of 1,000 flips, then we estimate *p* equals 0.525.

How *confident* can we be about this estimate? Well, if we knew the exact value of *p*, the central limit theorem (recall "The Central Limit Theorem") tells us that the average of those Bernoulli variables should be approximately normal, with mean *p* and standard deviation:

```
math.sqrt(p * (1 - p) / 1000)
```

Here we don't know *p*, so instead we use our estimate:

```
p_hat = 525 / 1000
mu = p_hat
sigma = math.sqrt(p_hat * (1 - p_hat) / 1000)   # 0.0158
```

This is not entirely justified, but people seem to do it anyway. Using the normal approximation, we conclude that we are "95% confident" that the following interval contains the true parameter *p*:

```
normal_two_sided_bounds(0.95, mu, sigma)        # [0.4940, 0.5560]
```

> **NOTE**
>
> This is a statement about the *interval*, not about *p*. You should understand it as the assertion that if you were to repeat the experiment many times, 95% of the time the "true" parameter (which is the same every time) would lie within the observed confidence interval (which might be different every time).

In particular, we do not conclude that the coin is unfair, since 0.5 falls within our confidence interval.

If instead we'd seen 540 heads, then we'd have:

```
p_hat = 540 / 1000
mu = p_hat
sigma = math.sqrt(p_hat * (1 - p_hat) / 1000) # 0.0158
normal_two_sided_bounds(0.95, mu, sigma) # [0.5091, 0.5709]
```

Here, "fair coin" doesn't lie in the confidence interval. (The "fair coin" hypothesis doesn't pass a test that you'd expect it to pass 95% of the time if it were true.)

P-hacking

A procedure that erroneously rejects the null hypothesis only 5% of the time will — by definition — 5% of the time erroneously reject the null hypothesis:

```
def run_experiment():
    """flip a fair coin 1000 times, True = heads, False = tails"""
    return [random.random() < 0.5 for _ in range(1000)]

def reject_fairness(experiment):
    """using the 5% significance levels"""
    num_heads = len([flip for flip in experiment if flip])
    return num_heads < 469 or num_heads > 531

random.seed(0)
experiments = [run_experiment() for _ in range(1000)]
num_rejections = len([experiment
                      for experiment in experiments
                      if reject_fairness(experiment)])

print num_rejections   # 46
```

What this means is that if you're setting out to find "significant" results, you usually can. Test enough hypotheses against your data set, and one of them will almost certainly appear significant. Remove the right outliers, and you can probably get your *p* value below 0.05. (We did something vaguely similar in "Correlation"; did you notice?)

This is sometimes called P-hacking and is in some ways a consequence of the "inference from *p*-values framework." A good article criticizing this approach is "The Earth Is Round."

If you want to do good *science*, you should determine your hypotheses before looking at the data, you should clean your data without the hypotheses in mind, and you should keep in mind that *p*-values are not substitutes for common sense. (An alternative approach is "Bayesian Inference".)

Example: Running an A/B Test

One of your primary responsibilities at DataSciencester is experience optimization, which is a euphemism for trying to get people to click on advertisements. One of your advertisers has developed a new energy drink targeted at data scientists, and the VP of Advertisements wants your help choosing between advertisement A (“tastes great!”) and advertisement B (“less bias!”).

Being a *scientist*, you decide to run an *experiment* by randomly showing site visitors one of the two advertisements and tracking how many people click on each one.

If 990 out of 1,000 A-viewers click their ad while only 10 out of 1,000 B-viewers click their ad, you can be pretty confident that A is the better ad. But what if the differences are not so stark? Here’s where you’d use statistical inference.

Let’s say that N_A people see ad A, and that n_A of them click it. We can think of each ad view as a Bernoulli trial where p_A is the probability that someone clicks ad A. Then (if N_A is large, which it is here) we know that n_A / N_A is approximately a normal random variable with mean p_A and standard deviation $\sigma_A = \sqrt{p_A(1 - p_A) / N_A}$.

Similarly, n_B / N_B is approximately a normal random variable with mean p_B and standard deviation $\sigma_B = \sqrt{p_B(1 - p_B) / N_B}$:

```
def estimated_parameters(N, n):
    p = n / N
    sigma = math.sqrt(p * (1 - p) / N)
    return p, sigma
```

If we assume those two normals are independent (which seems reasonable, since the individual Bernoulli trials ought to be), then their difference should also be normal with mean $p_B - p_A$ and standard deviation $\sqrt{\sigma_A^2 + \sigma_B^2}$.

> **NOTE**
>
> This is sort of cheating. The math only works out exactly like this if you *know* the standard deviations. Here we’re estimating them from the data, which means that we really should be using a *t*-distribution. But for large enough data sets, it’s close enough that it doesn’t make much of a difference.

This means we can test the *null hypothesis* that p_A and p_B are the same (that is, that $p_A - p_B$ is zero), using the statistic:

```
def a_b_test_statistic(N_A, n_A, N_B, n_B):
    p_A, sigma_A = estimated_parameters(N_A, n_A)
    p_B, sigma_B = estimated_parameters(N_B, n_B)
    return (p_B - p_A) / math.sqrt(sigma_A ** 2 + sigma_B ** 2)
```

which should approximately be a standard normal.

For example, if “tastes great” gets 200 clicks out of 1,000 views and “less bias” gets 180 clicks out of 1,000 views, the statistic equals:

```
z = a_b_test_statistic(1000, 200, 1000, 180)    # -1.14
```

The probability of seeing such a large difference if the means were actually equal would be:

```
two_sided_p_value(z)                            # 0.254
```

which is large enough that you can't conclude there's much of a difference. On the other hand, if "less bias" only got 150 clicks, we'd have:

```
z = a_b_test_statistic(1000, 200, 1000, 150)    # -2.94
two_sided_p_value(z)                            # 0.003
```

which means there's only a 0.003 probability you'd see such a large difference if the ads were equally effective.