# Hypothesis and Inference

## Lecture 7

Centre for Data Science
Institute of Technical Education and Research
Siksha 'O' Anusandhan (Deemed to be University)
Bhubaneswar, Odisha, 751030

# Overview

# Statistical Hypothesis Testing

- **Hypotheses** are assertions like "this coin is fair" or "data scientists prefer Python to R" that can be translated into statistics about data.
- Those statistics can be thought of as observations of random variables from known distributions.
- It allows us to make statements about how likely those assumptions are to hold.
- In the classical setup, we have
  - **Null hypothesis**, $H_0$, represents some default position.
  - **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.

# 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.
  - Our null hypothesis is that the coin is fair that is, that $p = 0.5$.
- In particular, our test will involve flipping the coin some number, n, times and counting the number of heads, X.

# Example: Flipping a Coin

- Each coin flip is a Bernoulli trial, which means that X is a Binomial(n,p) random variable, which we can approximate using the normal distribution.

```
from typing import Tuple
import math
def normal_approximation_to_binomial(n: int, p: float) -> Tuple[float, float]:
    """Returns mu and sigma corresponding to a Binomial(n, p)"""
    mu = p * n
    sigma = math.sqrt(p * (1 - p) * n)
    return mu, sigma
```

# Example: Flipping a Coin

- 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:

```
from scratch.probability import normal_cdf
# 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: float,mu: float = 0,
    sigma: float = 1) -> float:
    """The probability that an N(mu, sigma) is greater
    than lo."""
    return 1 - normal_cdf(lo, mu, sigma)
```

# Example: Flipping a Coin

```
# It's between if it's less than hi, but not less than lo
def normal_probability_between(lo: float,hi: float,mu: float
    = 0,sigma: float = 1) -> float:
    """The probability that an N(mu, sigma) is between lo and
     hi."""
    return normal_cdf(hi, mu, sigma) - normal_cdf(lo, mu,
    sigma)
# It's outside if it's not between
def normal_probability_outside(lo: float,hi: float,mu: float
    = 0,sigma: float = 1) -> float:
    """The probability that an N(mu, sigma) is not between lo
     and hi."""
    return 1 - normal_probability_between(lo, hi, mu, sigma)
```

# Example: Flipping a Coin

- 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%).

```
from scratch.probability import inverse_normal_cdf
def normal_upper_bound(probability: float,mu: float = 0,
    sigma: float = 1) -> float:
    """Returns the z for which P(Z <= z) = probability"""
    return inverse_normal_cdf(probability, mu, sigma)
def normal_lower_bound(probability: float,mu: float = 0,
    sigma: float = 1) -> float:
    """Returns the z for which P(Z >= z) = probability"""
    return inverse_normal_cdf(1 - probability, mu, sigma)
```

# Example: Flipping a Coin

```
1 def normal_two_sided_bounds(probability: float ,mu: float = 0,
    sigma: float = 1) -> Tuple[float , float]:
2     """Returns the symmetric (about the mean) bounds that
    contain the specified probability"""
3     tail_probability = (1 - probability) / 2
4     # upper bound should have tail_probability above it
5     upper_bound = normal_lower_bound(tail_probability , mu,
    sigma)
6     # lower bound should have tail_probability below it
7     lower_bound = normal_upper_bound(tail_probability , mu,
    sigma)
8     return lower_bound , upper_bound
```

# Example: Flipping a Coin

- Let's say that we choose to flip the coin n = 1,000 times.
- If our hypothesis of fairness is true, X should be distributed approximately normally with mean 500 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 *type 1 error*, we reject $H_0$ even though it's true.

# Example: Flipping a Coin

- Consider the test that rejects $H_0$ if X falls outside the bounds given by

```
lower_bound, upper_bound = normal_two_sided_bounds(0.95,
    mu_0, sigma_0)
```

- Assuming p really equals 0.5 (i.e., H0 is true), there is just a 5% chance we observe an X that lies outside this interval.
- If $H_0$ is true, then, approximately 19 times out of 20, this test will give the correct result.

# Example: Flipping a Coin

- We are also often interested in the power of a test, which is the probability of not making a *type 2 error* ("false negative").
- In *type 2 error*, we fail to reject $H_0$ even though it's false.
- In particular, let's check what happens if p is really 0.55, so that the coin is slightly biased toward heads.

```
# 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
```

# 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: float, mu: float = 0, sigma:
    float = 1) -> float:
    if x >= mu:
        # x is greater than the mean, so the tail is
    everything greater than x
        return 2 * normal_probability_above(x, mu, sigma)
    else:
        # x is less than the mean, so the tail is
    everything less than x
        return 2 * normal_probability_below(x, mu, sigma)
```

# p- Values

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

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

**Note**

Why did we use a value of 529.5 rather than using 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.

# p- Values

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

```
import random
extreme_value_count = 0
for _ in range(1000):
    num_heads = sum(1 if random.random() < 0.5 else 0 for
     _ in range(1000))
    if num_heads >= 530 or num_heads <= 470:
        extreme_value_count += 1
    # p-value was 0.062 => ~62 extreme values out of 1000
assert 59 < extreme_value_count < 65, f"{
    extreme_value_count}"
```

- 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
```

# p- Values

- The p-Value computed above 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.
- For our one-sided test, if we saw 525 heads we would compute:

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

  - It 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
```

  - It means we would reject the null.

# 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.
- **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?

# Confidence Intervals

- 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 would lie within the observed confidence interval.

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

# Confidence Intervals

- 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.

# 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.
- Test enough hypotheses against your dataset, and one of them will almost certainly appear significant.
- Remove the right outliers, and you can probably get your p-value below 0.05.
- This is sometimes called **p-hacking**. and in some ways a consequence of the "inference from p-values framework."

# p-Hacking

```python
from typing import List
def run_experiment() -> List[bool]:
    """Flips a fair coin 1000 times, True = heads, False = tails"""
    return [random.random() < 0.5 for _ in range(1000)]
def reject_fairness(experiment: List[bool]) -> bool:
    """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)])
assert num_rejections == 46
```

# Example: Running an AIB Test

- **Assume** 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!").
- 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**?

# Example: Running an AIB Test

- 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.
- 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 $\sigma_B = \sqrt{p_B(1-p_B)/N_B}$.

```
1 def estimated_parameters(N: int, n: int) -> Tuple[float,
      float]:
2 p = n / N
3 sigma = math.sqrt(p * (1 - p) / N)
4 return p, sigma
```

# Example: Running an AIB Test

- If we assume those two normals are independent then their difference should also be normal with mean $p_A - p_B$ and standard deviation $\sqrt{\sigma_A^2 + \sigma_B^2}$.
- This means we can test the null hypothesis that $p_A$ and $p_B$ are the same( i.e. $p_A - p_B = 0$).

```
def a_b_test_statistic(N_A: int, n_A: int, N_B: int, n_B: int) -> float:
    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)
```

# Example: Running an AIB Test

- For example, if "tastes great" gets 200 clicks out of 1,000 views and —"ess 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 we 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 we'd see such a large difference if the ads were equally effective.

# Bayesian Inference

- An alternative approach to inference involves treating the unknown parameters themselves as random variables.
- The analyst starts with a prior distribution for the parameters and then uses the observed data and Bayes's theorem to get an updated posterior distribution for the parameters.
- Rather than making probability judgments about the tests, you make probability judgments about the parameters.

# Bayesian Inference

- For example, when the unknown parameter is a probability, we often use a prior from the Beta distribution, which puts all its probability between 0 and 1:

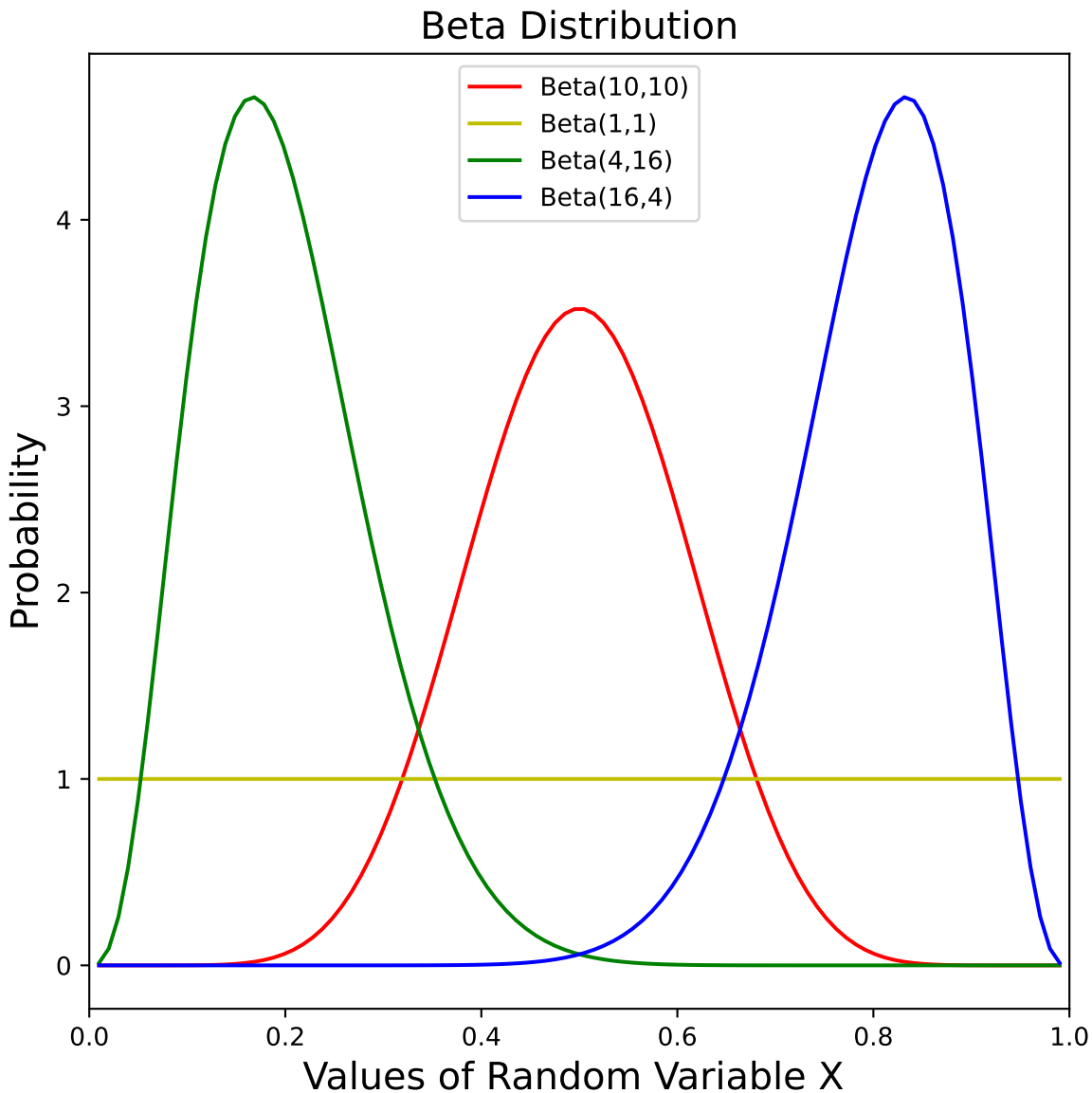
```
1 def B(alpha: float, beta: float) -> float:
2     """A normalizing constant so that the total
      probability is 1"""
3     return math.gamma(alpha) * math.gamma(beta) / math.
      gamma(alpha + beta)
4 def beta_pdf(x: float, alpha: float, beta: float) ->
      float:
5     if x <= 0 or x >= 1: # no weight outside of [0, 1]
6         return 0
7     return x ** (alpha - 1) * (1 - x) ** (beta - 1) / B(
      alpha, beta)
```

- Generally speaking, this distribution centers its weight at: $alpha/(alpha + beta)$ and the larger alpha and beta are, the "tighter" the distribution is.

# Bayesian Inference

- **For example**, if alpha and beta are both 1, it's just the uniform distribution.
- If alpha is much larger than beta, most of the weight is near 1.
- if alpha is much smaller than beta, most of the weight is near 0.

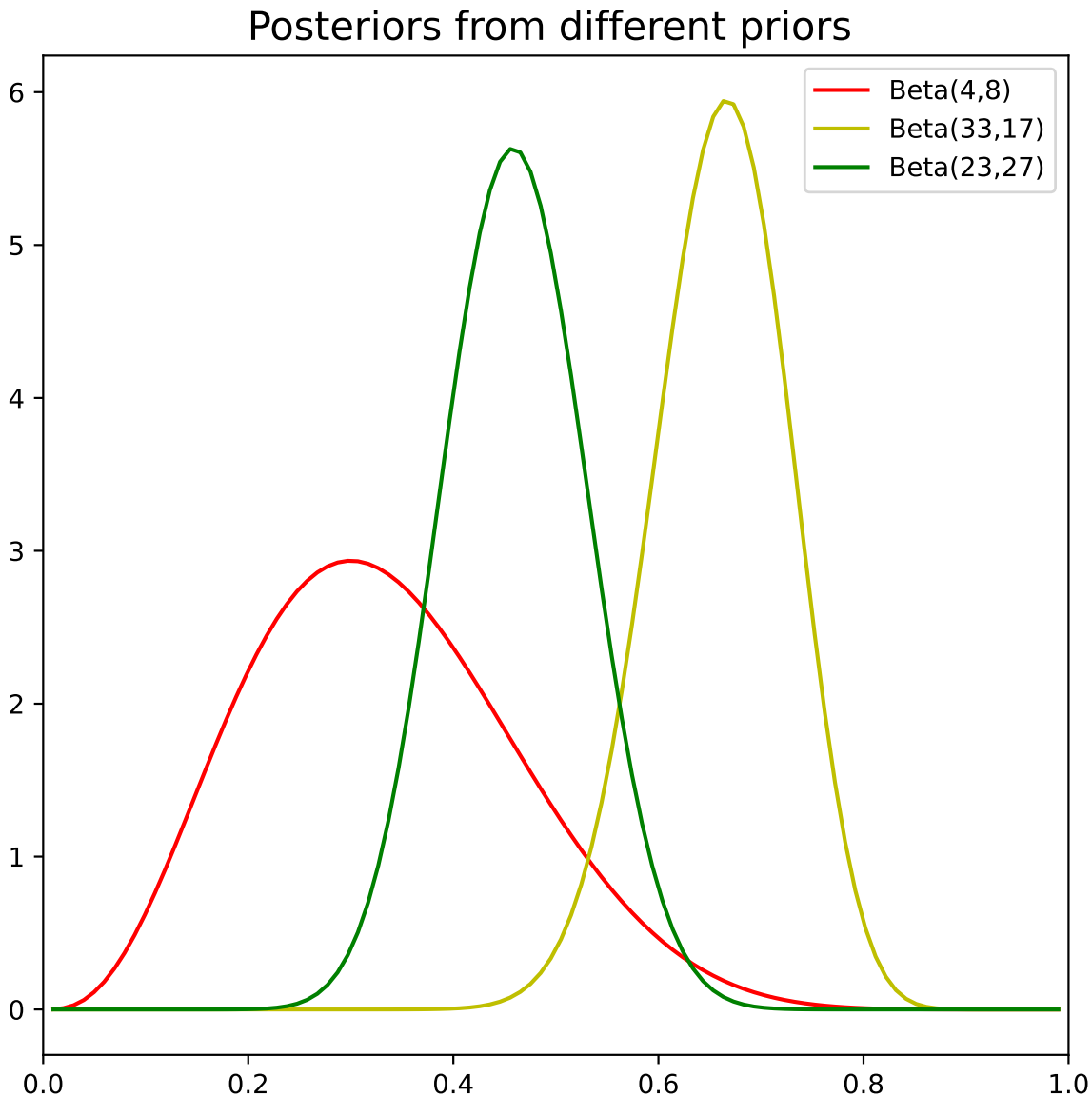
# Bayesian Inference



# Bayesian Inference

- Maybe we don't want to take a stand on whether the coin is fair, and we choose *alpha* and *beta* to both equal 1.
- Then we flip our coin a bunch of times and see *h* heads and *t* tails.
- Bayes's theorem tells us that the posterior distribution for p is again a Beta distribution, but with parameters *alpha* + *h* and *beta* + *t*.
- Let's say you flip the coin 10 times and see only 3 heads.
  Your posterior distribution would be a Beta(4, 8), centered around 0.33. $\{(4,8) = (1+3, 1+7)\}$
- If you started with a Beta(20, 20), your posterior distribution would be a Beta(23, 27), centered around 0.46. $\{(23,27) = (20+3, 20+7)\}$
- If you started with a Beta(30, 10), your posterior distribution would be a Beta(33, 17), centered around 0.66. $\{(33,17) = (30+3, 10+7)\}$

# Bayesian Inference



# Bayesian Inference

- If you flipped the coin more and more times, the prior would matter less and less until eventually you'd have (nearly) the same posterior distribution no matter which prior you started with.
- Using Bayesian inference to test hypotheses is considered somewhat controversial in part because the mathematics can get somewhat complicated and in part because of the subjective nature of choosing a prior.

# References

[1] Joel Grus. Data Science from Scratch. First Principles with Python. Second Edition. O'REILLY, May 2019.

# Thank You