Example: Running an A/B Test

Python code snippet to Running an A/B Test

```python
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)


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

```

✓ 0.4s

```
-1.1403464899034472
```

Example: Running an A/B Test

Example 1: Not a Big Difference

"Tastes great" gets 200 clicks out of 1000 views → p_A = 0.20

"Less bias" gets 180 clicks out of 1000 views → p_B = 0.18

```
z = a_b_test_statistic(1000, 200, 1000, 180)  # ≈ -1.14
two_sided_p_value(z)  # ≈ 0.254
```

Interpretation:

A z-score of -1.14 means the observed difference is only 1.14 standard deviations below the mean (i.e., not very surprising).

A p-value of 0.254 = 25.4% chance of seeing such a difference due to random chance, even if both ads are equally effective.

Conclusion: The difference is not statistically significant.

Example: Running an A/B Test

Example 2: Likely a Real Difference

"Tastes great" gets 200 clicks out of 1000 views → p_A = 0.20“

Less bias" gets 150 clicks out of 1000 views → p_B = 0.15

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

Interpretation: A z-score of -2.94 means the difference is almost 3 standard deviations away.

A p-value of 0.003 = 0.3% chance this could happen just by random chance.

Conclusion: Statistically significant difference.

Bayesian Inference

- Statements about our *tests we made were like:*
- "*there's only a 5% chance you'd observe such an extreme statistic if our null hypothesis were true.*"
- An alternative approach to inference involves treating the unknown parameters as random variables.
- **The analyst (that's you) 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.**

Bayesian Inference

- The **Beta distribution** is often used as a **prior probability distribution** in **Bayesian statistics —** especially when you're modeling the probability of **binary outcomes**, like success/failure or heads/tails.
- The Beta distribution is a way to model beliefs about a probability (like the chance of a coin landing heads):
  - alpha = how many "successes" (e.g., heads) you believe in.
  - beta = how many "failures" (e.g., tails) you believe in.
  - Beta(1, 1) prior → this is uniform across [0, 1], meaning you have no preference or knowledge.
  - Beta(5, 2) → this encodes your belief that the probability of heads is likely around 70%.

Bayesian Inference

The Beta function, is used to normalize the Beta distribution. Makes sure the total area under the curve equals 1 (which is a requirement for any probability distribution).

$$B(\alpha, \beta) = \frac{\Gamma(\alpha) \cdot \Gamma(\beta)}{\Gamma(\alpha + \beta)}$$

Where:

- `Γ(n)` is the **Gamma function**, which generalizes factorial:

$$\Gamma(n) = (n-1)!$$

(only for positive integers)

Calculate `B(2, 3)`

$$B(2,3) = \frac{\Gamma(2) \cdot \Gamma(3)}{\Gamma(5)} = \frac{1! \cdot 2!}{4!} = \frac{1 \cdot 2}{24} = \frac{2}{24} = 0.0833$$

Bayesian Inference

Python code snippet for Beta Function

```
import math
def B(alpha, beta):
    return math.gamma(alpha) * math.gamma(beta) / math.gamma(alpha + beta)
```

Bayesian Inference

- The PDF gives the shape of the distribution —
  - It tells you how likely different values of the parameter (e.g., a coin's probability of heads) are.
- For the posterior, the PDF tells you:
  - "Given the data I've seen, how likely is each possible value of the parameter?"

Bayesian Inference

It's a **continuous probability distribution** used to model probabilities themselves. Represent uncertainty about a proportion, like:

$$f(x;\alpha,\beta) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)} \quad \text{for } x \in [0,1]$$

Symbol	Meaning
x	The value you're evaluating (a probability between 0 and 1)
α	Shape parameter: related to prior "successes"
β	Shape parameter: related to prior "failures"
$B(\alpha,\beta)$	**Beta function** (a normalization constant so the total area under the curve = 1)
$x^{\alpha-1}(1-x)^{\beta-1}$	Shapes the distribution based on your belief about probability

Interpretation:

This formula gives the **height** (density) of the Beta distribution at any point $x \in [0,1]$.

The values of α and β **control the shape** of the distribution:

- High α: pushes the peak toward **1**
- High β: pushes the peak toward **0**
- Equal $\alpha = \beta$: makes it symmetric

Bayesian Inference

It's a **continuous probability distribution** used to model probabilities themselves. Represent uncertainty about a proportion, like:

$$f(x;\alpha,\beta) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)} \quad \text{for } x \in [0,1]$$

Python code snippet for Continuous Probability Distribution in Bayesian Inference

```
def beta_pdf(x, alpha, beta):
    if x < 0 or x > 1: # no weight outside of [0, 1]
    return 0
    return x ** (alpha - 1) * (1 - x) ** (beta - 1) / B(alpha, beta)
```

Bayesian Inference

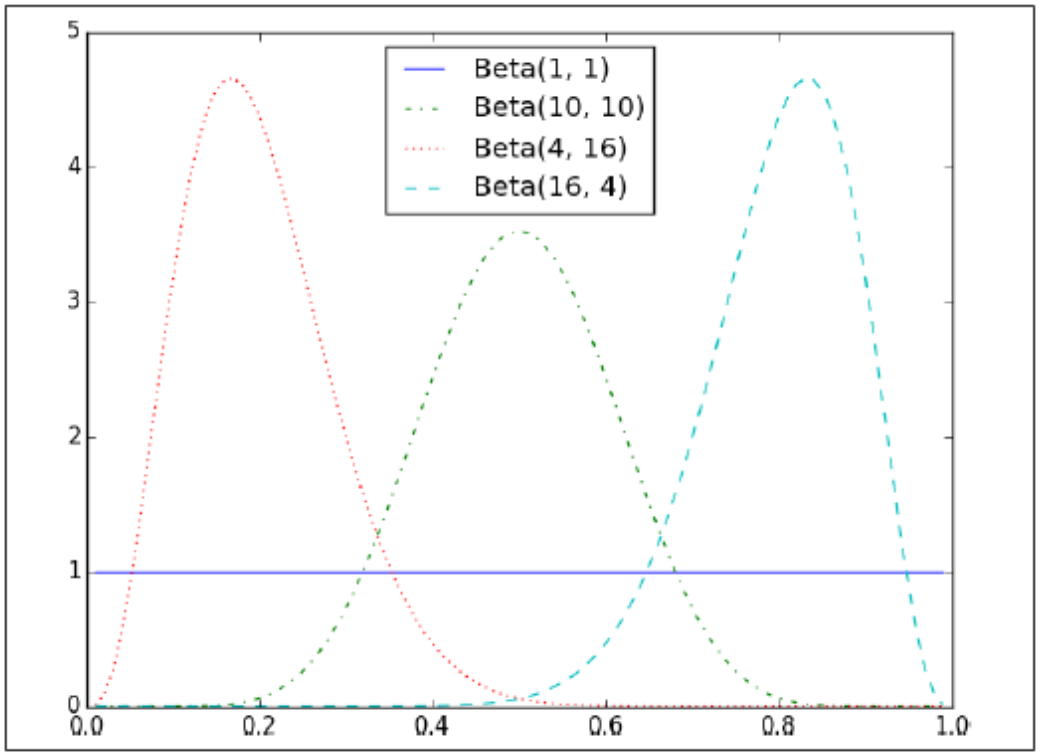


Figure 7-1. Example Beta distributions

- If alpha and beta are both 1, it's just the uniform distribution (centered at 0.5, very dispersed).
- If alpha is much larger than beta, most of the weight is near 1.
- And if alpha is much smaller than beta, most of the weight is near zero

Bayesian Inference

Bayes's Theorem Tells us that the posterior distribution for p is again a Beta distribution but with parameters alpha + h and beta + t.

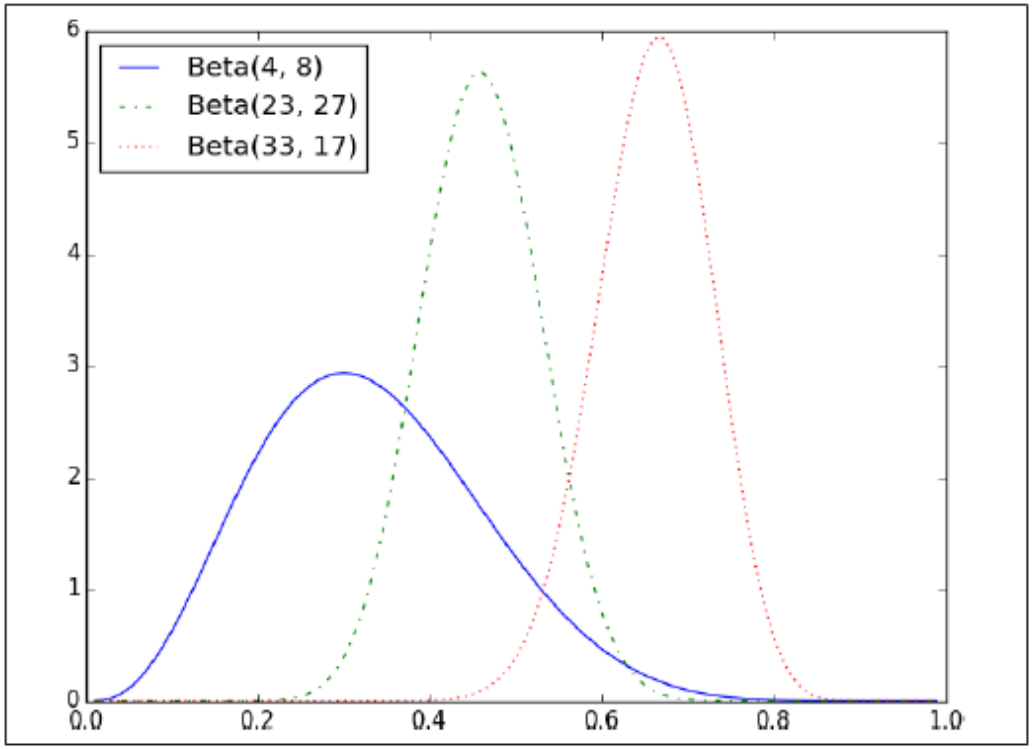


Figure 7-2. Posteriors arising from different priors

- Flip the coin 10 times and see only 3 heads.
- If you started with the uniform Beta(1,1) prior your posterior distribution would be a Beta(4, 8), centered around 0.33 your best guess is something pretty close to the observed probability.
- If you started with a Beta(20, 20) (expressing the belief that the coin was roughly fair), your posterior distribution would be a Beta(23, 27), centered around 0.46, indicating a revised belief that maybe the coin is slightly biased toward tails.
- And if you started with a Beta(30, 10) (expressing a belief that the coin was biased to flip 75% heads), your posterior distribution would be a Beta(33, 17), centered around 0.66

Gradient Descent

- Frequently when doing data science, we try to the find the **best model** for a certain situation.
- And usually "best" will mean something like "**minimizes the error of its predictions**" **or** "**maximizes the likelihood of the data.**"
- In other words, it will represent the solution to some sort of **optimization problem**.
- This means we'll need to solve a number of optimization problems. And need to solve them from scratch.
- Our approach will be a technique called ***gradient descent***, which lends itself pretty well to a from scratch treatment.

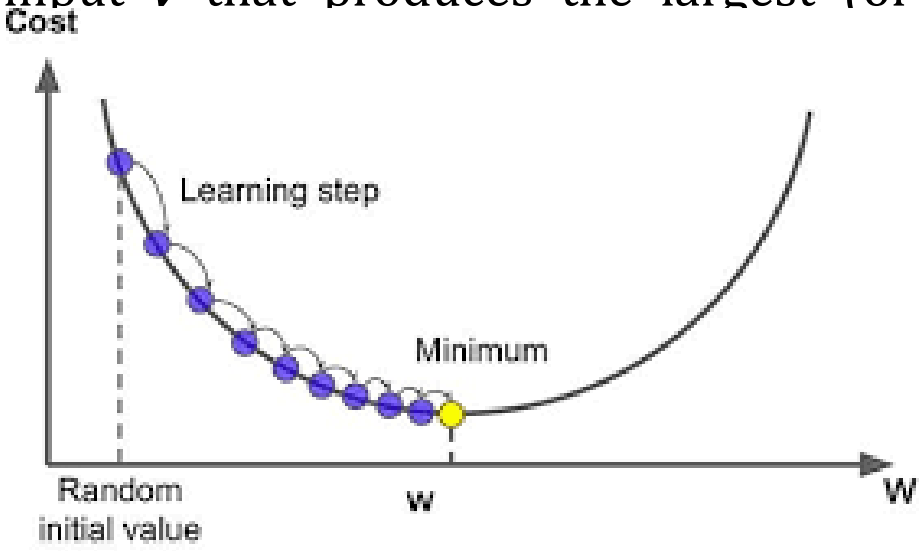
- Gradient descent is an optimization algorithm used in machine learning to find the minimum of a function, typically a cost or loss function, by iteratively adjusting the model's parameters.
- It works by moving in the direction of the steepest descent, akin to rolling downhill on a hill, until a minimum is reached.
- It's an iterative process where the algorithm repeatedly adjusts the parameters of a model in the direction that decreases the cost function, aiming to find the optimal parameter values.

The Idea Behind Gradient Descent

- Suppose we have some function ***f*** that takes as input a vector of real numbers and outputs a single real number. One simple such function is:

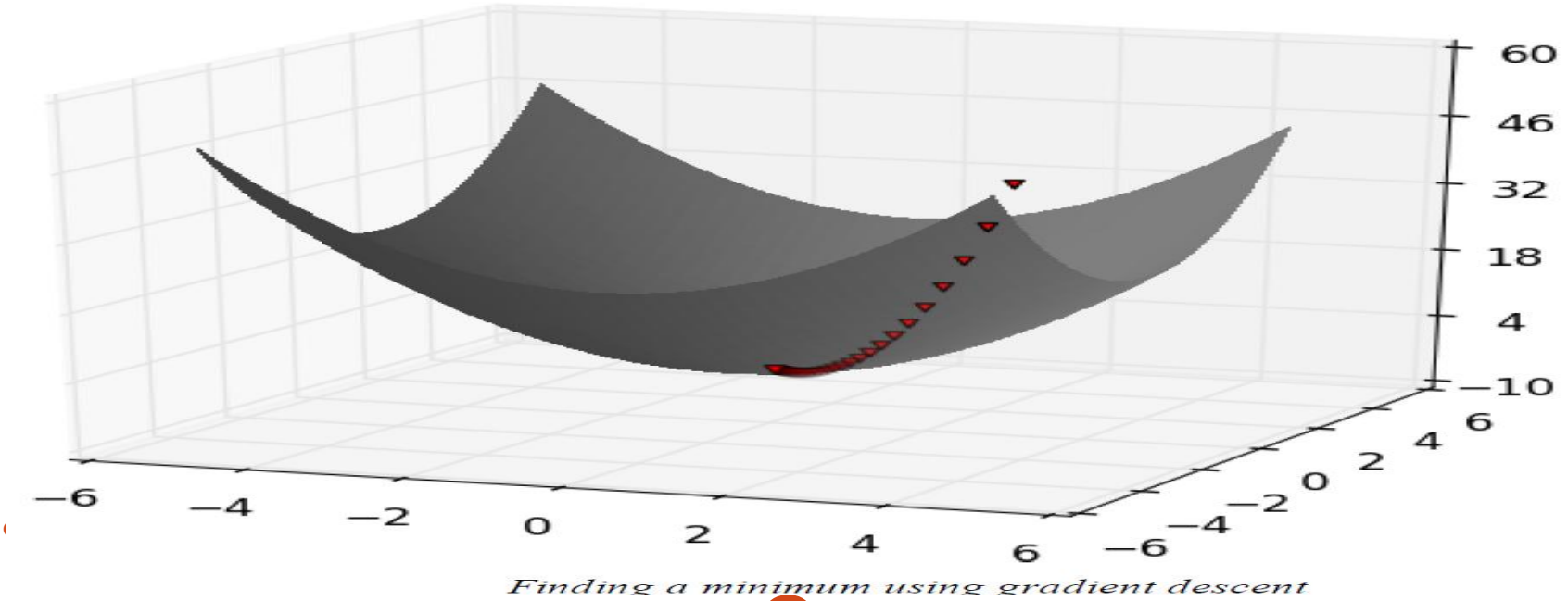
```
def sum_of_squares(v):
"""computes the sum of squared elements in v"""
return sum(v_i ** 2 for v_i in v)
```

- We'll frequently need to maximize or minimize such functions.
- That is, we need to find the input ***v*** that produces the largest (or smallest) possible value.



The Idea Behind Gradient Descent

- For functions like ours, the gradient (if you remember your calculus, this is the vector of partial derivatives) gives the input direction in which the function most quickly increases.
- Accordingly, one approach to maximizing a function is to pick a random starting point, compute the gradient, take a small step in the direction of the gradient (i.e., the direction that causes the function to increase the most), and repeat with the new starting point.
- Similarly, you can try to minimize a function by taking small steps in the opposite direction, as shown in Figure



Finding a minimum using gradient descent

Estimating the Gradient

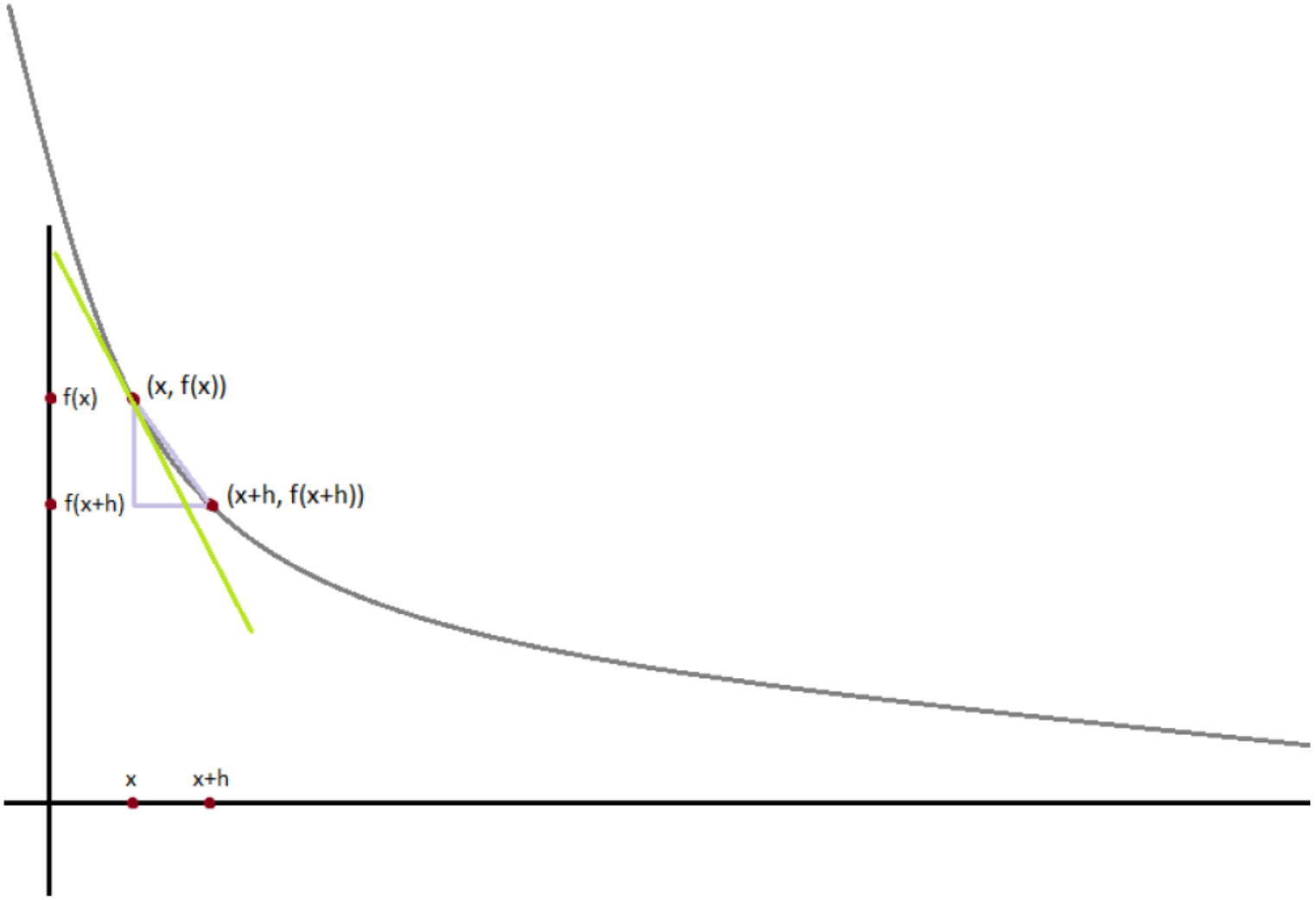
- If ***f*** is a function of one variable, its derivative at a point ***x*** measures how ***f(x)*** changes when we make a very small change to ***x***.
- The derivative is defined as the limit of the difference quotients:

```
def square(x):
        return x * x
def derivative(x):
        return 2 * x
def difference_quotient(f, x, h):
        return (f(x + h) - f(x)) / h
```

as h approaches zero.

- The derivative is the slope of the tangent line at (x, f(x)), while the difference quotient is the slope of the not-quite-tangent line that runs through (x + h, f(x + h)).
- As h gets smaller and smaller, the not-quite-tangent line gets closer and closer to the tangent line

Estimating the Gradient

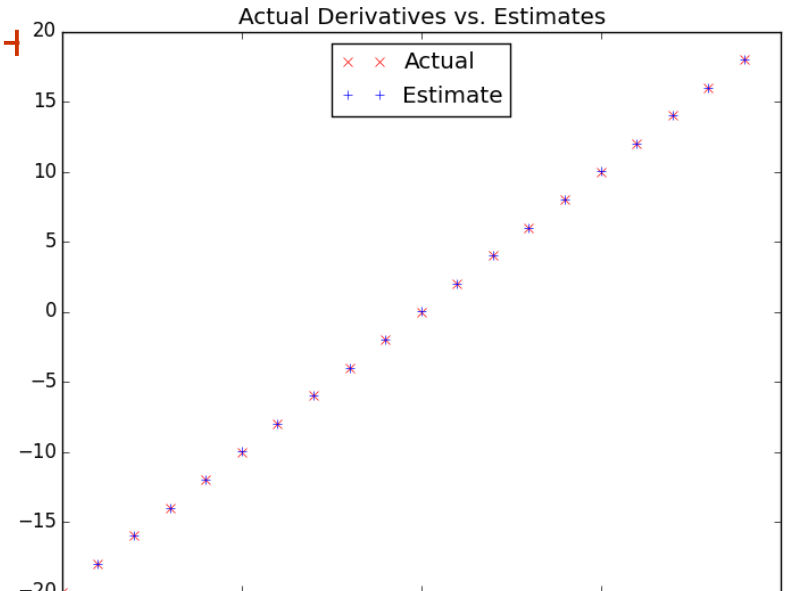


Approximating a derivative with a difference quotient

Estimating the Gradient

Python code snippet for estimating gradient for X^2 and plot the graph for Actual Derivatives Vs Estimates

```
derivative_estimate = partial(difference_quotient, square, h=0.00001)
# plot to show they're basically the same
import matplotlib.pyplot as plt
x = range(-10,10)
plt.title("Actual Derivatives vs. Estimates")
plt.plot(x, map(derivative, x), 'rx', label='Actual') # red x
plt.plot(x, map(derivative_estimate, x), 'b+
plt.legend(loc=9)
plt.show()
```



Estimating the Gradient

- When ***f*** is a function of many variables, it has multiple partial derivatives, each indicating how ***f*** changes when we make small changes in just one of the input variables.

```
Function: f(x, y) = x^2 + y^2
At point x = 3.0, y = 4.0
True ∂f/∂x: 6.0
Estimated ∂f/∂x: 6.00001
True ∂f/∂y: 8.0
Estimated ∂f/∂y: 8.00001
```

```
def partial_differe                                    :
    """compute the ith partial difference quotient of f at v"""
    w = [v_j + (h if j == i else 0) for j, v_j in enumerate(v)]
    return (f(w) - f(v)) / h
def estimate_gradient(f, v, h=0.00001):
  return [partial_difference_quotient(f, v, i, h) for i, _ in enumerate(v)]
```

- We calculate its ***i***th partial derivative by treating it as a function of just its ***i***th variable, holding the other variables fixed:

Using Gradient Descent

- It's easy to see that the ***sum_of_squares*** function is smallest when its input ***v*** is a vector of zeros.
- But imagine we didn't know that.
- Let's use gradients to find the minimum among all three-dimensional vectors.
- We'll just pick a random starting point and then take tiny steps in the opposite direction of the gradient until we reach a point where the gradient is very small:

Using Gradient Descent

Python Program to find the minimum among all three-dimensional vectors using gradients

```
import random
from scratch.linear_algebra import distance, add, scalar_multiply
def gradient_step(v: Vector, gradient: Vector, step_size: float) -> Vector:
    """Moves `step_size` in the `gradient` direction from `v`"""
    assert len(v) == len(gradient)
    step = scalar_multiply(step_size, gradient)
    return add(v, step)
def sum_of_squares_gradient(v: Vector) -> Vector:
     return [2 * v_i for v_i in v]
# pick a random starting point
v = [random.uniform(-10, 10) for i in range(3)]
for epoch in range(1000):
    grad = sum_of_squares_gradient(v) # compute the gradient at v
    v = gradient_step(v, grad, -0.01) # take a negative gradient step
    print(epoch, v)
assert distance(v, [0, 0, 0]) < 0.001 # v should be close to 0
```

Choosing the right step size

- Although the rationale for moving against the gradient is clear, how far to move is not.
- Indeed, choosing the right step size is more of an art than a science.
- Popular **options** include:
  - Using a fixed step size
  - Gradually shrinking the step size over time
  - At each step, choosing the step size that minimizes the value of the objective function
- The last approach sounds great but is, in practice, a costly computation.
- To keep things simple, we'll mostly just use **a fixed step size.**
- The step size that "works" depends on the problem—too small, or too big, **will need to experiment**.

Choosing the right step size

- We'll be using gradient descent to fit parameterized models to data.
- In the usual case, we'll have some dataset and some (hypothesized) model for the data that depends (in a differentiable way) on one or more parameters.
- We'll also have a loss function that measures how well the model fits our data. (Smaller is better.)
- If we think of our data as being fixed, then our loss function tells us how good or bad any particular model parameters are.
- This means we can use gradient descent to find the model parameters that make the loss as small as possible.

Using Gradient Descent to Fit Models

- Let's think about what that gradient means.
- Imagine for some x our prediction is too large. In that case the error is positive.
- The second gradient term, 2 * error, is positive, which reflects the fact that small increases in the intercept will make the (already too large) prediction even larger, which will cause the squared error (for this x) to get even bigger.
- The first gradient term, 2 * error * x, has the same sign as x.
- Sure enough, if x is positive, small increases in the slope will again make the prediction (and hence the error) larger.

Using Gradient Descent to Fit Models

- If x is negative, though, small increases in the slope will make the prediction (and hence the error) smaller.
- Now, that computation was for a single data point. For the whole dataset we'll look at the mean squared error.
- And the gradient of the mean squared error is just the mean of the individual gradients.
- So, here's what we're going to do:
    1. Start with a random value for theta.
    2. Compute the mean of the gradients.
    3. Adjust theta in that direction.
    4. Repeat.

Using Gradient Descent to Fit Models

- Let's look at a simple example:

Python Program to demonstrate use of gradient descent is used to fit parameterized model Y=20X+5

```
# x ranges from -50 to 49, y is always 20 * x + 5
inputs = [(x, 20 * x + 5) for x in range(-50, 50)]
def linear_gradient(x: float, y: float, theta: Vector) -> Vector:
    slope, intercept = theta
    predicted = slope * x + intercept # The prediction of the model.
    error = (predicted - y) # error is (predicted - actual).
    squared_error = error ** 2 # We'll minimize squared error
    grad = [2 * error * x, 2 * error] # using its gradient.
    return grad
```

Using Gradient Descent to Fit Models

After a lot of epochs (what we call each pass through the dataset), we should learn something like the correct parameters:

```
 from scratch.linear_algebra import vector_mean
# Start with random values for slope and intercept
theta = [random.uniform(-1, 1), random.uniform(-1, 1)]
learning_rate = 0.001
for epoch in range(5000):
      # Compute the mean of the gradients
      grad = vector_mean([linear_gradient(x, y, theta) for x, y in inputs])
      # Take a step in that direction
      theta = gradient_step(theta, grad, -learning_rate)
      print(epoch, theta)
slope, intercept = theta
assert 19.9 < slope < 20.1, "slope should be about 20"
assert 4.9 < intercept < 5.1, "intercept should be about 5"
```

Minibatch Gradient Descent

- One drawback of the preceding approach is that we had to evaluate the gradients on the entire dataset before we could take a gradient step and update our parameters.
- In this case it was fine, because our dataset was only 100 pairs and the gradient computation was cheap.
- Your models, however, will frequently have large datasets and expensive gradient computations.
- In that case you'll want to take gradient steps more often.
- We can do this using a technique called ***minibatch gradient descent***, in which we compute the gradient (and take a gradient step) based on a "minibatch" sampled from the larger dataset:

Minibatch Gradient Descent

Python Code Snippet to demonstrate use of Minibatch gradient descent to fit parameterized model Y=20X+5

```
from typing import TypeVar, List, Iterator
T = TypeVar('T') # this allows us to type "generic" functions
def minibatches(dataset: List[T], batch_size: int, shuffle: bool = True) ->
Iterator[List[T]]:
    """Generates `batch_size`-sized minibatches from the dataset"""
    # start indexes 0, batch_size, 2 * batch_size, ...
    batch_starts = [start for start in range(0, len(dataset),batch_size)]
    if shuffle: random.shuffle(batch_starts) # shuffle the batches
    for start in batch_starts:
        end = start + batch_size
        yield dataset[start:end]
```

Minibatch Gradient Descent

```
theta = [random.uniform(-1, 1), random.uniform(-1, 1)]
for epoch in range(1000):
    for batch in minibatches(inputs, batch_size=20):
        grad = vector_mean([linear_gradient(x, y, theta) for x, y in
        batch])
        theta = gradient_step(theta, grad, -learning_rate)
        print(epoch, theta)
slope, intercept = theta
assert 19.9 < slope < 20.1, "slope should be about 20"
assert 4.9 < intercept < 5.1, "intercept should be about 5"
```

- **Stochastic Gradient Descent**
- Using the batch approach, each gradient step requires us to make a prediction and compute the gradient for the **whole data set**, which makes each step take a long time.
- Now, usually these error functions are additive, which means that the predictive error on the whole data set is simply the sum of the predictive errors for each data point.
- When this is the case, we can instead apply a technique called stochastic gradient descent, which computes the gradient (and takes a step) for only one point at a time.
- It cycles over our data repeatedly until it reaches a stopping point.

Stochastic Gradient Descent

Stochastic Gradient Descent

Python Code Snippet to demonstrate use of Stochastic gradient descent to fit parameterized model Y=20X+5

```
theta = [random.uniform(-1, 1), random.uniform(-1, 1)]
for epoch in range(100):
    for x, y in inputs:
        grad = linear_gradient(x, y, theta)
        theta = gradient_step(theta, grad, -learning_rate)
    print(epoch, theta)
slope, intercept = theta
assert 19.9 < slope < 20.1, "slope should be about 20"
assert 4.9 < intercept < 5.1, "intercept should be about 5"
```