Part VI

Information

CHAPTER

FORTYONE

JOB SEARCH VII: SEARCH WITH LEARNING

Contents

In addition to what's in Anaconda, this lecture deploys the libraries:

```
!conda install -y quantecon
!pip install interpolation
```

41.1 Overview

In this lecture, we consider an extension of the previously studied job search model of McCall [McC70].

We'll build on a model of Bayesian learning discussed in this lecture on the topic of exchangeability and its relationship to the concept of IID (identically and independently distributed) random variables and to Bayesian updating.

In the McCall model, an unemployed worker decides when to accept a permanent job at a specific fixed wage, given

- his or her discount factor
- the level of unemployment compensation

- the distribution from which wage offers are drawn

In the version considered below, the wage distribution is unknown and must be learned.

- The following is based on the presentation in [LS18], section 6.6.

Let's start with some imports

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
from numba import njit, prange, vectorize
from interpolation import mlinterp, interp
from math import gamma
import numpy as np
from matplotlib import cm
import scipy.optimize as op
from scipy.stats import cumfreq, beta
```

41.1.1 Model Features

- Infinite horizon dynamic programming with two states and one binary control.
- Bayesian updating to learn the unknown distribution.

41.2 Model

Let's first review the basic McCall model [McC70] and then add the variation we want to consider.

41.2.1 The Basic McCall Model

Recall that, *in the baseline model*, an unemployed worker is presented in each period with a permanent job offer at wage W_t.

At time t, our worker either

1. accepts the offer and works permanently at constant wage W_t
2. rejects the offer, receives unemployment compensation c and reconsiders next period

The wage sequence W_t is IID and generated from known density q.

The worker aims to maximize the expected discounted sum of earnings $\mathbb{E} \sum_{t=0}^{\infty} \beta^t y_t$ The function V satisfies the recursion

$$v(w) = \max \left\{ \frac{w}{1-\beta}, \, c + \beta \int v(w') q(w') dw' \right\} \tag{1}$$

The optimal policy has the form $\mathbf{1}\{w \geq \bar{w}\}$, where $\bar{w}$ is a constant called the *reservation wage*.

41.2.2 Offer Distribution Unknown

Now let's extend the model by considering the variation presented in [LS18], section 6.6.

The model is as above, apart from the fact that

- the density q is unknown
- the worker learns about q by starting with a prior and updating based on wage offers that he/she observes

The worker knows there are two possible distributions F and G — with densities f and g.

At the start of time, "nature" selects q to be either f or g — the wage distribution from which the entire sequence W_t will be drawn.

This choice is not observed by the worker, who puts prior probability π_0 on f being chosen.

Update rule: worker's time t estimate of the distribution is $\pi_t f + (1 - \pi_t)g$, where π_t updates via

$$\pi_{t+1} = \frac{\pi_t f(w_{t+1})}{\pi_t f(w_{t+1}) + (1 - \pi_t) g(w_{t+1})} \tag{2}$$

This last expression follows from Bayes' rule, which tells us that

$$\mathbb{P}\{q = f \,|\, W = w\} = \frac{\mathbb{P}\{W = w \,|\, q = f\}\mathbb{P}\{q = f\}}{\mathbb{P}\{W = w\}} \quad \text{and} \quad \mathbb{P}\{W = w\} = \sum_{\omega \in \{f, g\}} \mathbb{P}\{W = w \,|\, q = \omega\}\mathbb{P}\{q = \omega\}$$

The fact that (2) is recursive allows us to progress to a recursive solution method.

Letting

$$q_\pi(w) := \pi f(w) + (1 - \pi) g(w) \quad \text{and} \quad \kappa(w, \pi) := \frac{\pi f(w)}{\pi f(w) + (1 - \pi) g(w)}$$

we can express the value function for the unemployed worker recursively as follows

$$v(w, \pi) = \max \left\{ \frac{w}{1 - \beta}, \, c + \beta \int v(w', \pi') \, q_\pi(w') \, dw' \right\} \quad \text{where} \quad \pi' = \kappa(w', \pi) \tag{3}$$

Notice that the current guess π is a state variable, since it affects the worker's perception of probabilities for future rewards.

41.2.3 Parameterization

Following section 6.6 of [LS18], our baseline parameterization will be

- f is Beta$(1, 1)$
- g is Beta$(3, 1.2)$
- $\beta = 0.95$ and $c = 0.3$

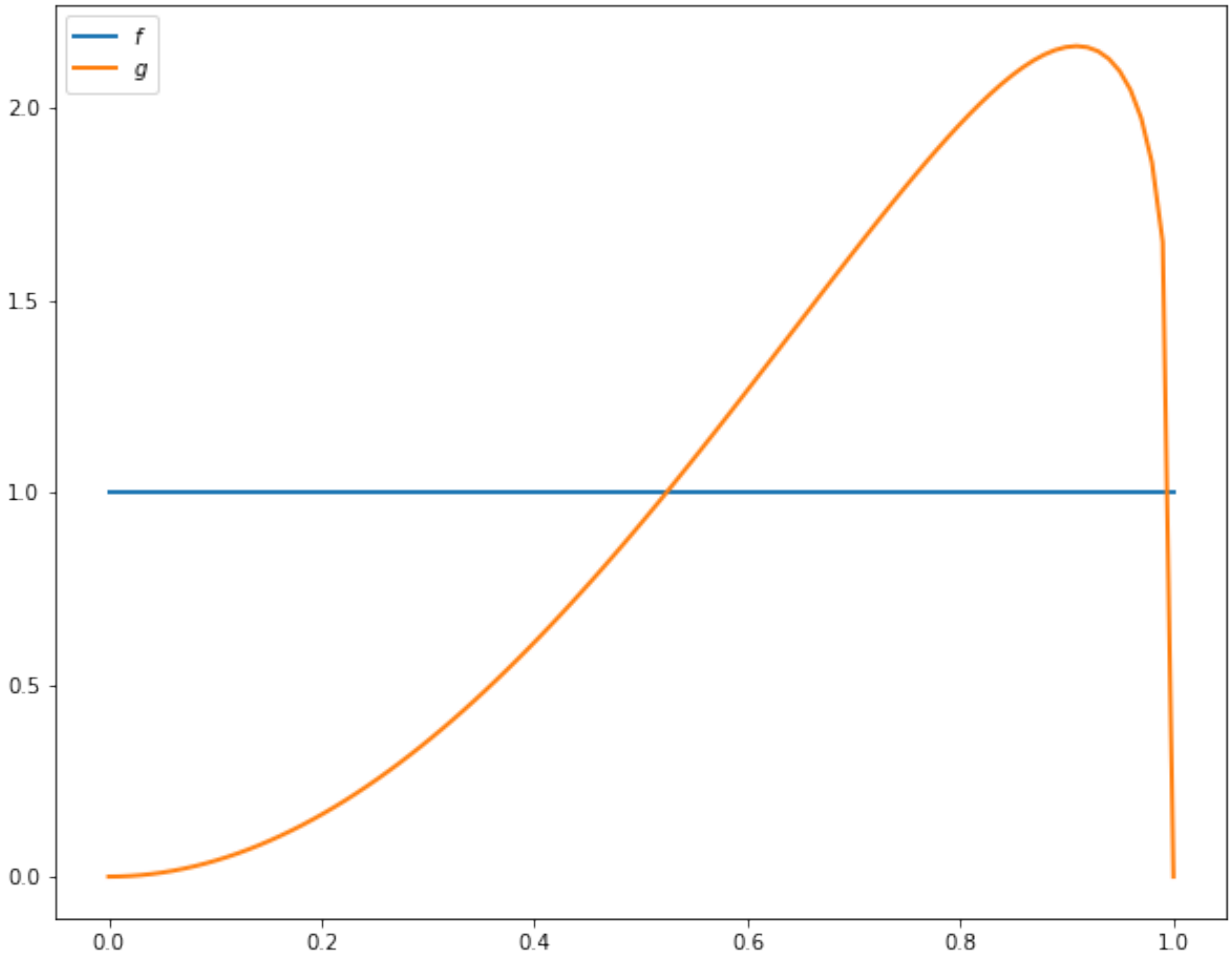
The densities f and g have the following shape

```
@vectorize
def p(x, a, b):
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * x**(a-1) * (1 - x)**(b-1)


x_grid = np.linspace(0, 1, 100)
f = lambda x: p(x, 1, 1)
```

(continues on next page)

(continued from previous page)

```
g = lambda x: p(x, 3, 1.2)

fig, ax = plt.subplots(figsize=(10, 8))
ax.plot(x_grid, f(x_grid), label='$f$', lw=2)
ax.plot(x_grid, g(x_grid), label='$g$', lw=2)

ax.legend()
plt.show()
```



41.2.4 Looking Forward

What kind of optimal policy might result from (3) and the parameterization specified above?

Intuitively, if we accept at w_a and $w_a \leq w_b$, then — all other things being given — we should also accept at w_b.

This suggests a policy of accepting whenever w exceeds some threshold value $\bar{w}$.

But $\bar{w}$ should depend on π — in fact, it should be decreasing in π because

- f is a less attractive offer distribution than g
- larger π means more weight on f and less on g

Thus larger π depresses the worker's assessment of her future prospects, and relatively low current offers become more attractive.

Summary: We conjecture that the optimal policy is of the form $\mathbb{1}w \geq \bar{w}(\pi)$ for some decreasing function $\bar{w}$.

41.3 Take 1: Solution by VFI

Let's set about solving the model and see how our results match with our intuition.

We begin by solving via value function iteration (VFI), which is natural but ultimately turns out to be second best.

The class `SearchProblem` is used to store parameters and methods needed to compute optimal actions.

```
class SearchProblem:
    """
    A class to store a given parameterization of the "offer distribution
    unknown" model.

    """

    def __init__(self,
                 β=0.95,            # Discount factor
                 c=0.3,             # Unemployment compensation
                 F_a=1,
                 F_b=1,
                 G_a=3,
                 G_b=1.2,
                 w_max=1,           # Maximum wage possible
                 w_grid_size=100,
                 π_grid_size=100,
                 mc_size=500):

        self.β, self.c, self.w_max = β, c, w_max

        self.f = njit(lambda x: p(x, F_a, F_b))
        self.g = njit(lambda x: p(x, G_a, G_b))

        self.π_min, self.π_max = 1e-3, 1-1e-3    # Avoids instability
        self.w_grid = np.linspace(0, w_max, w_grid_size)
        self.π_grid = np.linspace(self.π_min, self.π_max, π_grid_size)

        self.mc_size = mc_size

        self.w_f = np.random.beta(F_a, F_b, mc_size)
        self.w_g = np.random.beta(G_a, G_b, mc_size)
```

The following function takes an instance of this class and returns jitted versions of the Bellman operator `T`, and a `get_greedy()` function to compute the approximate optimal policy from a guess `v` of the value function

```
def operator_factory(sp, parallel_flag=True):

    f, g = sp.f, sp.g
    w_f, w_g = sp.w_f, sp.w_g
    β, c = sp.β, sp.c
    mc_size = sp.mc_size
    w_grid, π_grid = sp.w_grid, sp.π_grid
```

(continues on next page)

(continued from previous page)

```
    @njit
    def v_func(x, y, v):
        return mlinterp((w_grid, π_grid), v, (x, y))

    @njit
    def κ(w, π):
        """
        Updates π using Bayes' rule and the current wage observation w.
        """
        pf, pg = π * f(w), (1 - π) * g(w)
        π_new = pf / (pf + pg)

        return π_new

    @njit(parallel=parallel_flag)
    def T(v):
        """
        The Bellman operator.

        """
        v_new = np.empty_like(v)

        for i in prange(len(w_grid)):
            for j in prange(len(π_grid)):
                w = w_grid[i]
                π = π_grid[j]

                v_1 = w / (1 - β)

                integral_f, integral_g = 0, 0
                for m in prange(mc_size):
                    integral_f += v_func(w_f[m], κ(w_f[m], π), v)
                    integral_g += v_func(w_g[m], κ(w_g[m], π), v)
                integral = (π * integral_f + (1 - π) * integral_g) / mc_size

                v_2 = c + β * integral
                v_new[i, j] = max(v_1, v_2)

        return v_new

    @njit(parallel=parallel_flag)
    def get_greedy(v):
        """"
        Compute optimal actions taking v as the value function.

        """
        σ = np.empty_like(v)

        for i in prange(len(w_grid)):
            for j in prange(len(π_grid)):
                w = w_grid[i]
                π = π_grid[j]

                v_1 = w / (1 - β)

                integral_f, integral_g = 0, 0
                for m in prange(mc_size):
```

(continues on next page)

(continued from previous page)

```
                    integral_f += v_func(w_f[m], κ(w_f[m], π), v)
                    integral_g += v_func(w_g[m], κ(w_g[m], π), v)
                integral = (π * integral_f + (1 - π) * integral_g) / mc_size

                v_2 = c + β * integral

                σ[i, j] = v_1 > v_2  # Evaluates to 1 or 0

        return σ

    return T, get_greedy
```

We will omit a detailed discussion of the code because there is a more efficient solution method that we will use later.

To solve the model we will use the following function that iterates using T to find a fixed point

```
def solve_model(sp,
                use_parallel=True,
                tol=1e-4,
                max_iter=1000,
                verbose=True,
                print_skip=5):

    """
    Solves for the value function

    * sp is an instance of SearchProblem
    """

    T, _ = operator_factory(sp, use_parallel)

    # Set up loop
    i = 0
    error = tol + 1
    m, n = len(sp.w_grid), len(sp.π_grid)

    # Initialize v
    v = np.zeros((m, n)) + sp.c / (1 - sp.β)

    while i < max_iter and error > tol:
        v_new = T(v)
        error = np.max(np.abs(v - v_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        v = v_new

    if i == max_iter:
        print("Failed to converge!")

    if verbose and i < max_iter:
        print(f"\nConverged in {i} iterations.")


    return v_new
```

Let's look at solutions computed from value function iteration

```
sp = SearchProblem()
v_star = solve_model(sp)
fig, ax = plt.subplots(figsize=(6, 6))
ax.contourf(sp.π_grid, sp.w_grid, v_star, 12, alpha=0.6, cmap=cm.jet)
cs = ax.contour(sp.π_grid, sp.w_grid, v_star, 12, colors="black")
ax.clabel(cs, inline=1, fontsize=10)
ax.set(xlabel='$\pi$', ylabel='$w$')

plt.show()
```

```
Error at iteration 5 is 0.628771132601825.
```

```
Error at iteration 10 is 0.09448480297581341.
```

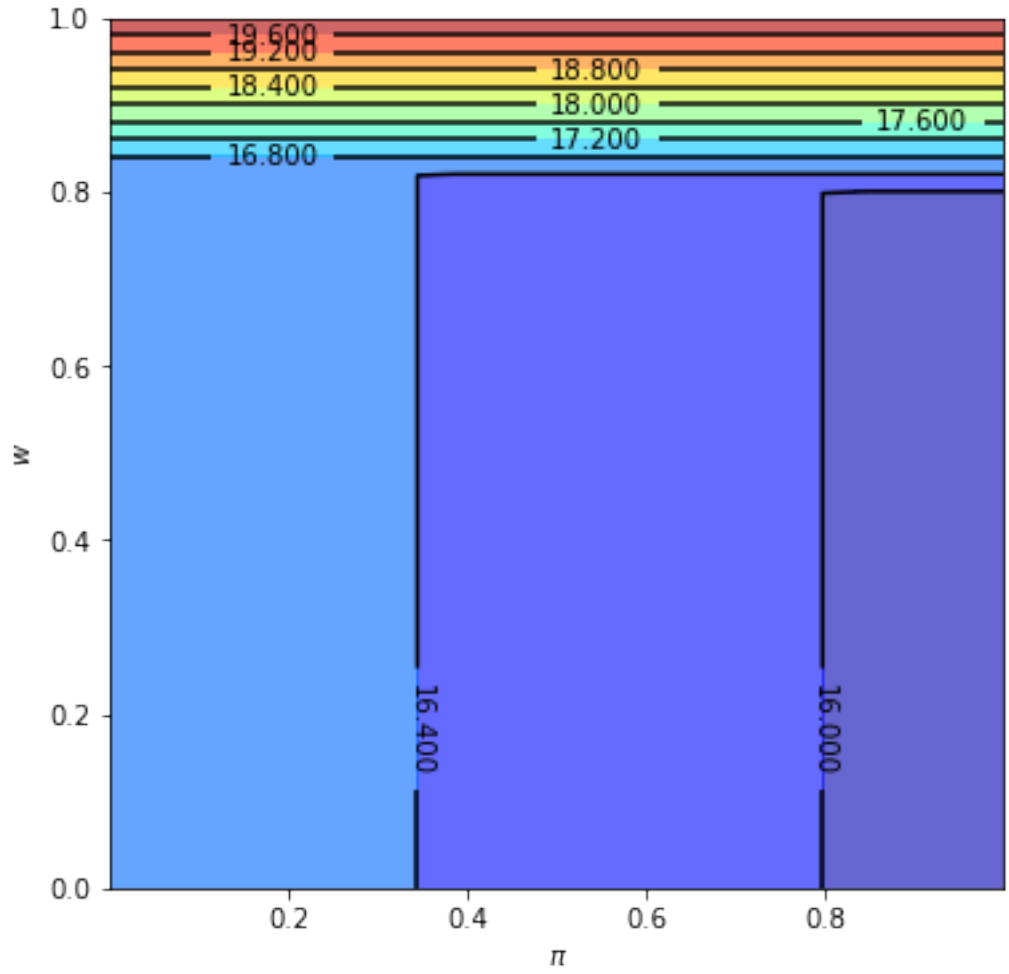
```
Error at iteration 15 is 0.018582313759575086.
```

```
Error at iteration 20 is 0.0040371680976569735.
```

```
Error at iteration 25 is 0.0008885268254843481.
```

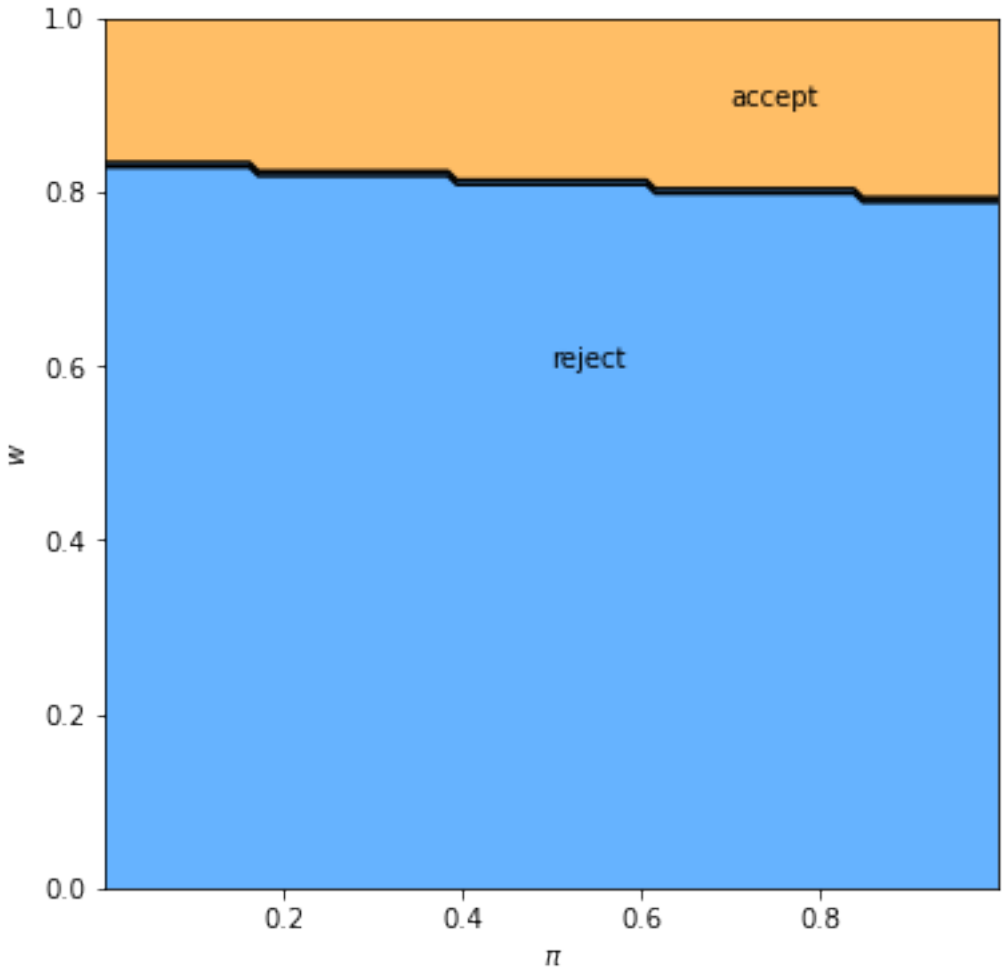
```
Error at iteration 30 is 0.00019558229431737573.
```

```
Converged in 33 iterations.
```



We will also plot the optimal policy

```
T, get_greedy = operator_factory(sp)
σ_star = get_greedy(v_star)

fig, ax = plt.subplots(figsize=(6, 6))
ax.contourf(sp.π_grid, sp.w_grid, σ_star, 1, alpha=0.6, cmap=cm.jet)
ax.contour(sp.π_grid, sp.w_grid, σ_star, 1, colors="black")
ax.set(xlabel='$\pi$', ylabel='$w$')

ax.text(0.5, 0.6, 'reject')
ax.text(0.7, 0.9, 'accept')

plt.show()
```



The results fit well with our intuition from section *looking forward*.

- The black line in the figure above corresponds to the function $\bar{w}(\pi)$ introduced there.
- It is decreasing as expected.

41.4 Take 2: A More Efficient Method

Let's consider another method to solve for the optimal policy.

We will use iteration with an operator that has the same contraction rate as the Bellman operator, but

- one dimensional rather than two dimensional
- no maximization step

As a consequence, the algorithm is orders of magnitude faster than VFI.

This section illustrates the point that when it comes to programming, a bit of mathematical analysis goes a long way.

41.5 Another Functional Equation

To begin, note that when $w = \bar{w}(\pi)$, the worker is indifferent between accepting and rejecting.

Hence the two choices on the right-hand side of (3) have equal value:

$$\frac{\bar{w}(\pi)}{1-\beta} = c + \beta \int v(w', \pi') \, q_\pi(w') \, dw' \tag{4}$$

Together, (3) and (4) give

$$v(w, \pi) = \max \left\{ \frac{w}{1-\beta}, \, \frac{\bar{w}(\pi)}{1-\beta} \right\} \tag{5}$$

Combining (4) and (5), we obtain

$$\frac{\bar{w}(\pi)}{1-\beta} = c + \beta \int \max \left\{ \frac{w'}{1-\beta}, \, \frac{\bar{w}(\pi')}{1-\beta} \right\} q_\pi(w') \, dw'$$

Multiplying by $1-\beta$, substituting in $\pi' = \kappa(w', \pi)$ and using $\circ$ for composition of functions yields

$$\bar{w}(\pi) = (1-\beta)c + \beta \int \max \left\{ w', \bar{w} \circ \kappa(w', \pi) \right\} \, q_\pi(w') \, dw' \tag{6}$$

Equation (6) can be understood as a functional equation, where $\bar{w}$ is the unknown function.

- Let's call it the *reservation wage functional equation* (RWFE).
- The solution $\bar{w}$ to the RWFE is the object that we wish to compute.

41.6 Solving the RWFE

To solve the RWFE, we will first show that its solution is the fixed point of a contraction mapping.

To this end, let

- $b[0, 1]$ be the bounded real-valued functions on $[0, 1]$
- $\|\omega\| := \sup_{x \in [0,1]} |\omega(x)|$

Consider the operator Q mapping $\omega \in b[0, 1]$ into $Q\omega \in b[0, 1]$ via

$$(Q\omega)(\pi) = (1-\beta)c + \beta \int \max \left\{ w', \omega \circ \kappa(w', \pi) \right\} \, q_\pi(w') \, dw' \tag{7}$$

Comparing (6) and (7), we see that the set of fixed points of Q exactly coincides with the set of solutions to the RWFE.

- If $Q\bar{w} = \bar{w}$ then $\bar{w}$ solves (6) and vice versa.

Moreover, for any $\omega, \omega' \in b[0, 1]$, basic algebra and the triangle inequality for integrals tells us that

$$|(Q\omega)(\pi) - (Q\omega')(\pi)| \leq \beta \int \left| \max \left\{ w', \omega \circ \kappa(w', \pi) \right\} - \max \left\{ w', \omega' \circ \kappa(w', \pi) \right\} \right| \, q_\pi(w') \, dw' \tag{8}$$

Working case by case, it is easy to check that for real numbers a, b, c we always have

$$|\max\{a, b\} - \max\{a, c\}| \leq |b - c| \tag{9}$$

Combining (8) and (9) yields

$$|(Q\omega)(\pi) - (Q\omega')(\pi)| \leq \beta \int |\omega \circ \kappa(w', \pi) - \omega' \circ \kappa(w', \pi)| \, q_\pi(w') \, dw' \leq \beta \|\omega - \omega'\| \tag{10}$$

Taking the supremum over π now gives us

$$\|Q\omega - Q\omega'\| \leq \beta\|\omega - \omega'\| \tag{11}$$

In other words, Q is a contraction of modulus β on the complete metric space $(b[0,1], \|\cdot\|)$.

Hence

- A unique solution $\bar{w}$ to the RWFE exists in $b[0,1]$.
- $Q^k\omega \to \bar{w}$ uniformly as $k \to \infty$, for any $\omega \in b[0,1]$.

41.7 Implementation

The following function takes an instance of `SearchProblem` and returns the operator `Q`

```
def Q_factory(sp, parallel_flag=True):

    f, g = sp.f, sp.g
    w_f, w_g = sp.w_f, sp.w_g
    β, c = sp.β, sp.c
    mc_size = sp.mc_size
    w_grid, π_grid = sp.w_grid, sp.π_grid

    @njit
    def ω_func(p, ω):
        return interp(π_grid, ω, p)

    @njit
    def κ(w, π):
        """
        Updates π using Bayes' rule and the current wage observation w.
        """
        pf, pg = π * f(w), (1 - π) * g(w)
        π_new = pf / (pf + pg)

        return π_new

    @njit(parallel=parallel_flag)
    def Q(ω):
        """

        Updates the reservation wage function guess ω via the operator
        Q.

        """
        ω_new = np.empty_like(ω)

        for i in prange(len(π_grid)):
            π = π_grid[i]
            integral_f, integral_g = 0, 0

            for m in prange(mc_size):
                integral_f += max(w_f[m], ω_func(κ(w_f[m], π), ω))
                integral_g += max(w_g[m], ω_func(κ(w_g[m], π), ω))
            integral = (π * integral_f + (1 - π) * integral_g) / mc_size
```

(continues on next page)

(continued from previous page)

```
            ω_new[i] = (1 - β) * c + β * integral

        return ω_new

    return Q
```

In the next exercise, you are asked to compute an approximation to $\bar{w}$.

41.8 Exercises

41.8.1 Exercise 1

Use the default parameters and `Q_factory` to compute an optimal policy.

Your result should coincide closely with the figure for the optimal policy *shown above*.

Try experimenting with different parameters, and confirm that the change in the optimal policy coincides with your intuition.

41.9 Solutions

41.9.1 Exercise 1

This code solves the "Offer Distribution Unknown" model by iterating on a guess of the reservation wage function.

You should find that the run time is shorter than that of the value function approach.

Similar to above, we set up a function to iterate with `Q` to find the fixed point

```
def solve_wbar(sp,
               use_parallel=True,
               tol=1e-4,
               max_iter=1000,
               verbose=True,
               print_skip=5):

    Q = Q_factory(sp, use_parallel)

    # Set up loop
    i = 0
    error = tol + 1
    m, n = len(sp.w_grid), len(sp.π_grid)

    # Initialize w
    w = np.ones_like(sp.π_grid)

    while i < max_iter and error > tol:
        w_new = Q(w)
        error = np.max(np.abs(w - w_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
```

(continues on next page)

(continued from previous page)

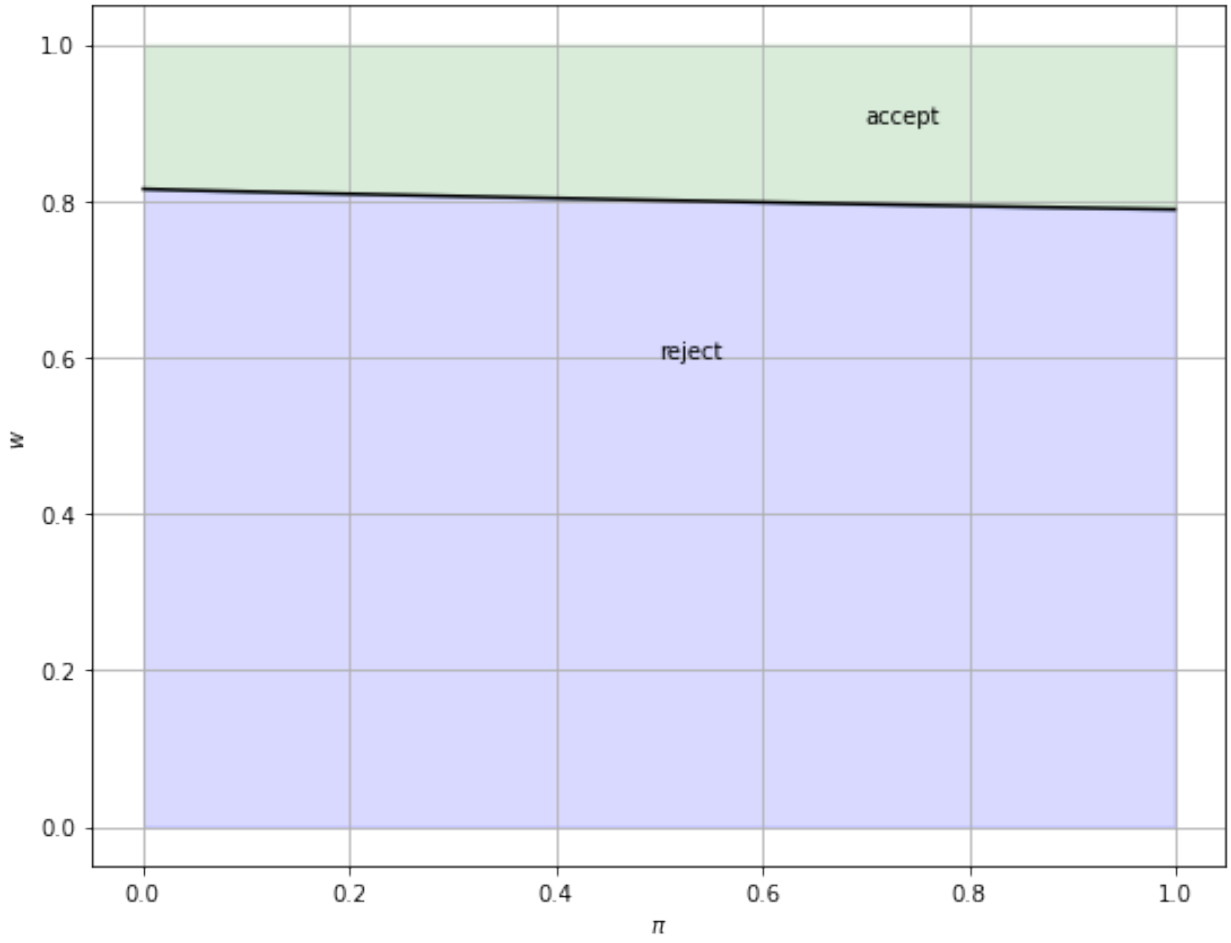
```
        w = w_new

    if i == max_iter:
        print("Failed to converge!")

    if verbose and i < max_iter:
        print(f"\nConverged in {i} iterations.")

    return w_new
```

The solution can be plotted as follows

```
sp = SearchProblem()
w_bar = solve_wbar(sp)

fig, ax = plt.subplots(figsize=(9, 7))

ax.plot(sp.π_grid, w_bar, color='k')
ax.fill_between(sp.π_grid, 0, w_bar, color='blue', alpha=0.15)
ax.fill_between(sp.π_grid, w_bar, sp.w_max, color='green', alpha=0.15)
ax.text(0.5, 0.6, 'reject')
ax.text(0.7, 0.9, 'accept')
ax.set(xlabel='$\pi$', ylabel='$w$')
ax.grid()
plt.show()
```

```
Error at iteration 5 is 0.02049948818856684.
Error at iteration 10 is 0.005736330570978998.
Error at iteration 15 is 0.0012388225586256185.
```

```
Error at iteration 20 is 0.000245237590392966.

Converged in 23 iterations.
```



41.10 Appendix A

The next piece of code generates a fun simulation to see what the effect of a change in the underlying distribution on the unemployment rate is.

At a point in the simulation, the distribution becomes significantly worse.

It takes a while for agents to learn this, and in the meantime, they are too optimistic and turn down too many jobs.

As a result, the unemployment rate spikes

```
F_a, F_b, G_a, G_b = 1, 1, 3, 1.2

sp = SearchProblem(F_a=F_a, F_b=F_b, G_a=G_a, G_b=G_b)
f, g = sp.f, sp.g

# Solve for reservation wage
w_bar = solve_wbar(sp, verbose=False)

# Interpolate reservation wage function
π_grid = sp.π_grid
w_func = njit(lambda x: interp(π_grid, w_bar, x))
```

(continues on next page)

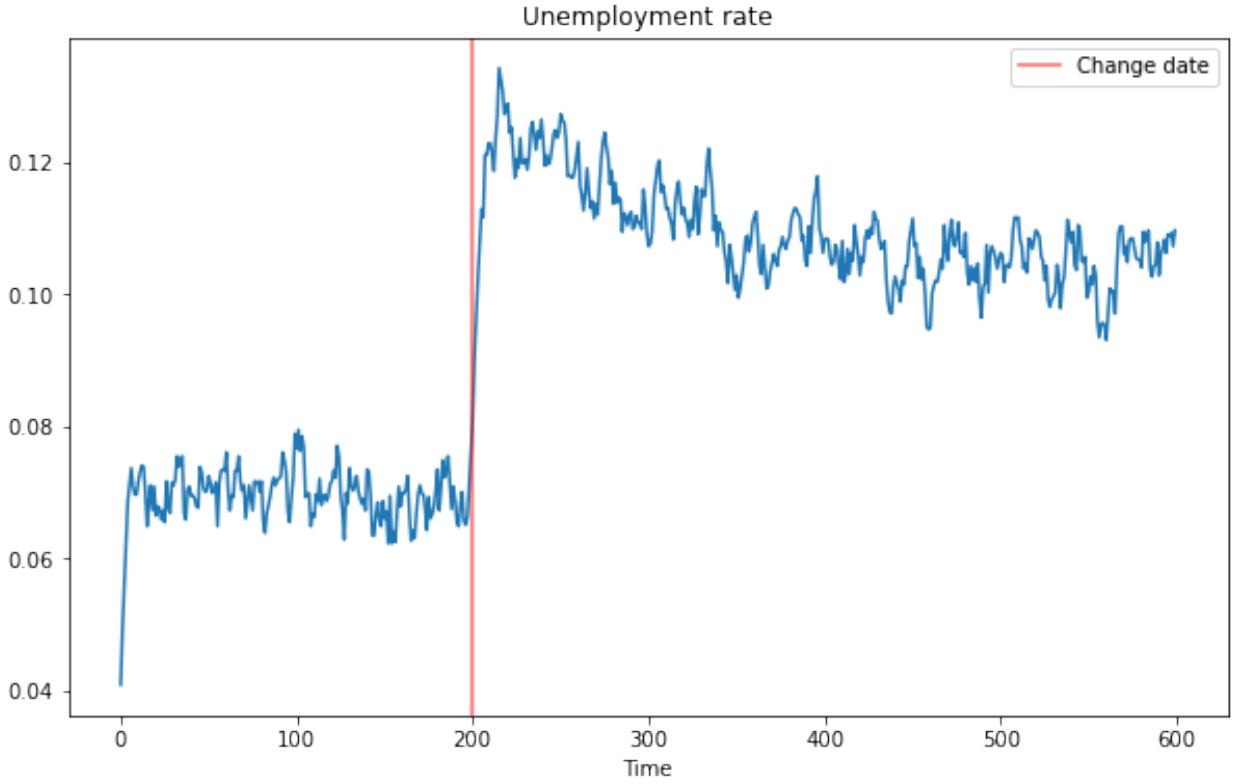
(continued from previous page)

```
@njit
def update(a, b, e, π):
    "Update e and π by drawing wage offer from beta distribution with parameters a
 ↪and b"

    if e == False:
        w = np.random.beta(a, b)       # Draw random wage
        if w >= w_func(π):
            e = True                   # Take new job
        else:
            π = 1 / (1 + ((1 - π) * g(w)) / (π * f(w)))

    return e, π

@njit
def simulate_path(F_a=F_a,
                  F_b=F_b,
                  G_a=G_a,
                  G_b=G_b,
                  N=5000,        # Number of agents
                  T=600,         # Simulation length
                  d=200,         # Change date
                  s=0.025):      # Separation rate

    """Simulates path of employment for N number of works over T periods"""

    e = np.ones((N, T+1))
    π = np.full((N, T+1), 1e-3)

    a, b = G_a, G_b   # Initial distribution parameters

    for t in range(T+1):

        if t == d:
            a, b = F_a, F_b  # Change distribution parameters

        # Update each agent
        for n in range(N):
            if e[n, t] == 1:                    # If agent is currently employment
                p = np.random.uniform(0, 1)
                if p <= s:                      # Randomly separate with probability s
                    e[n, t] = 0

            new_e, new_π = update(a, b, e[n, t], π[n, t])
            e[n, t+1] = new_e
            π[n, t+1] = new_π

    return e[:, 1:]

d = 200  # Change distribution at time d
unemployment_rate = 1 - simulate_path(d=d).mean(axis=0)

fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(unemployment_rate)
ax.axvline(d, color='r', alpha=0.6, label='Change date')
ax.set_xlabel('Time')
ax.set_title('Unemployment rate')
```

(continues on next page)

(continued from previous page)

```
ax.legend()
plt.show()
```



41.11 Appendix B

In this appendix we provide more details about how Bayes' Law contributes to the workings of the model.

We present some graphs that bring out additional insights about how learning works.

We build on graphs proposed in this lecture.

In particular, we'll add actions of our searching worker to a key graph presented in that lecture.

To begin, we first define two functions for computing the empirical distributions of unemployment duration and π at the time of employment.

```
@njit
def empirical_dist(F_a, F_b, G_a, G_b, w_bar, π_grid,
                   N=10000, T=600):
    """
    Simulates population for computing empirical cumulative
    distribution of unemployment duration and π at time when
    the worker accepts the wage offer. For each job searching
    problem, we simulate for two cases that either f or g is
    the true offer distribution.

    Parameters
```

(continues on next page)

(continued from previous page)

```
    ----------

    F_a, F_b, G_a, G_b : parameters of beta distributions F and G.
    w_bar : the reservation wage
    π_grid : grid points of π, for interpolation
    N : number of workers for simulation, optional
    T : maximum of time periods for simulation, optional

    Returns
    -------
    accpet_t : 2 by N ndarray. the empirical distribution of
               unemployment duration when f or g generates offers.
    accept_π : 2 by N ndarray. the empirical distribution of
               π at the time of employment when f or g generates offers.
    """

    accept_t = np.empty((2, N))
    accept_π = np.empty((2, N))

    # f or g generates offers
    for i, (a, b) in enumerate([(F_a, F_b), (G_a, G_b)]):
        # update each agent
        for n in range(N):

            # initial priori
            π = 0.5

            for t in range(T+1):

                # Draw random wage
                w = np.random.beta(a, b)
                lw = p(w, F_a, F_b) / p(w, G_a, G_b)
                π = π * lw / (π * lw + 1 - π)

                # move to next agent if accepts
                if w >= interp(π_grid, w_bar, π):
                    break

            # record the unemployment duration
            # and π at the time of acceptance
            accept_t[i, n] = t
            accept_π[i, n] = π

    return accept_t, accept_π

def cumfreq_x(res):
    """
    A helper function for calculating the x grids of
    the cumulative frequency histogram.
    """

    cumcount = res.cumcount
    lowerlimit, binsize = res.lowerlimit, res.binsize

    x = lowerlimit + np.linspace(0, binsize*cumcount.size, cumcount.size)

    return x
```

Now we define a wrapper function for analyzing job search models with learning under different parameterizations.

The wrapper takes parameters of beta distributions and unemployment compensation as inputs and then displays various things we want to know to interpret the solution of our search model.

In addition, it computes empirical cumulative distributions of two key objects.

```
def job_search_example(F_a=1, F_b=1, G_a=3, G_b=1.2, c=0.3):
    """
    Given the parameters that specify F and G distributions,
    calculate and display the rejection and acceptance area,
    the evolution of belief π, and the probability of accepting
    an offer at different π level, and simulate and calculate
    the empirical cumulative distribution of the duration of
    unemployment and π at the time the worker accepts the offer.
    """

    # construct a search problem
    sp = SearchProblem(F_a=F_a, F_b=F_b, G_a=G_a, G_b=G_b, c=c)
    f, g = sp.f, sp.g
    π_grid = sp.π_grid

    # Solve for reservation wage
    w_bar = solve_wbar(sp, verbose=False)

    # l(w) = f(w) / g(w)
    l = lambda w: f(w) / g(w)
    # objective function for solving l(w) = 1
    obj = lambda w: l(w) - 1.

    # the mode of beta distribution
    # use this to divide w into two intervals for root finding
    G_mode = (G_a - 1) / (G_a + G_b - 2)
    roots = np.empty(2)
    roots[0] = op.root_scalar(obj, bracket=[1e-10, G_mode]).root
    roots[1] = op.root_scalar(obj, bracket=[G_mode, 1-1e-10]).root

    fig, axs = plt.subplots(2, 2, figsize=(12, 9))

    # part 1: display the details of the model settings and some results
    w_grid = np.linspace(1e-12, 1-1e-12, 100)

    axs[0, 0].plot(l(w_grid), w_grid, label='$l$', lw=2)
    axs[0, 0].vlines(1., 0., 1., linestyle="--")
    axs[0, 0].hlines(roots, 0., 2., linestyle="--")
    axs[0, 0].set_xlim([0., 2.])
    axs[0, 0].legend(loc=4)
    axs[0, 0].set(xlabel='$l(w)=f(w)/g(w)$', ylabel='$w$')

    axs[0, 1].plot(sp.π_grid, w_bar, color='k')
    axs[0, 1].fill_between(sp.π_grid, 0, w_bar, color='blue', alpha=0.15)
    axs[0, 1].fill_between(sp.π_grid, w_bar, sp.w_max, color='green', alpha=0.15)
    axs[0, 1].text(0.5, 0.6, 'reject')
    axs[0, 1].text(0.7, 0.9, 'accept')

    W = np.arange(0.01, 0.99, 0.08)
    Π = np.arange(0.01, 0.99, 0.08)

    ΔW = np.zeros((len(W), len(Π)))
```

(continues on next page)

(continued from previous page)

```
    ΔΠ = np.empty((len(W), len(Π)))
    for i, w in enumerate(W):
        for j, π in enumerate(Π):
            lw = l(w)
            ΔΠ[i, j] = π * (lw / (π * lw + 1 - π) - 1)

    q = axs[0, 1].quiver(Π, W, ΔΠ, ΔW, scale=2, color='r', alpha=0.8)

    axs[0, 1].hlines(roots, 0., 1., linestyle="--")
    axs[0, 1].set(xlabel='$\pi$', ylabel='$w$')
    axs[0, 1].grid()

    axs[1, 0].plot(f(x_grid), x_grid, label='$f$', lw=2)
    axs[1, 0].plot(g(x_grid), x_grid, label='$g$', lw=2)
    axs[1, 0].vlines(1., 0., 1., linestyle="--")
    axs[1, 0].hlines(roots, 0., 2., linestyle="--")
    axs[1, 0].legend(loc=4)
    axs[1, 0].set(xlabel='$f(w), g(w)$', ylabel='$w$')

    axs[1, 1].plot(sp.π_grid, 1 - beta.cdf(w_bar, F_a, F_b), label='$f$')
    axs[1, 1].plot(sp.π_grid, 1 - beta.cdf(w_bar, G_a, G_b), label='$g$')
    axs[1, 1].set_ylim([0., 1.])
    axs[1, 1].grid()
    axs[1, 1].legend(loc=4)
    axs[1, 1].set(xlabel='$\pi$', ylabel='$\mathbb{P}\{w > \overline{w} (\pi)\}$')

    plt.show()

    # part 2: simulate empirical cumulative distribution
    accept_t, accept_π = empirical_dist(F_a, F_b, G_a, G_b, w_bar, π_grid)
    N = accept_t.shape[1]

    cfq_t_F = cumfreq(accept_t[0, :], numbins=100)
    cfq_π_F = cumfreq(accept_π[0, :], numbins=100)

    cfq_t_G = cumfreq(accept_t[1, :], numbins=100)
    cfq_π_G = cumfreq(accept_π[1, :], numbins=100)

    fig, axs = plt.subplots(2, 1, figsize=(12, 9))

    axs[0].plot(cumfreq_x(cfq_t_F), cfq_t_F.cumcount/N, label="f generates")
    axs[0].plot(cumfreq_x(cfq_t_G), cfq_t_G.cumcount/N, label="g generates")
    axs[0].grid(linestyle='--')
    axs[0].legend(loc=4)
    axs[0].title.set_text('CDF of duration of unemployment')
    axs[0].set(xlabel='time', ylabel='Prob(time)')

    axs[1].plot(cumfreq_x(cfq_π_F), cfq_π_F.cumcount/N, label="f generates")
    axs[1].plot(cumfreq_x(cfq_π_G), cfq_π_G.cumcount/N, label="g generates")
    axs[1].grid(linestyle='--')
    axs[1].legend(loc=4)
    axs[1].title.set_text('CDF of π at time worker accepts wage and leaves␣
↪unemployment')
    axs[1].set(xlabel='π', ylabel='Prob(π)')

    plt.show()
```

We now provide some examples that provide insights about how the model works.

41.12 Examples

41.12.1 Example 1 (Baseline)

F ~ Beta(1, 1), G ~ Beta(3, 1.2), c=0.3.

In the graphs below, the red arrows in the upper right figure show how π_t is updated in response to the new information w_t.

Recall the following formula from this lecture

$$\frac{\pi_{t+1}}{\pi_t} = \frac{l\left(w_{t+1}\right)}{\pi_t l\left(w_{t+1}\right) + \left(1 - \pi_t\right)} \begin{cases} > 1 & \text{if } l\left(w_{t+1}\right) > 1 \\ \leq 1 & \text{if } l\left(w_{t+1}\right) \leq 1 \end{cases}$$

The formula implies that the direction of motion of π_t is determined by the relationship between $l(w_t)$ and 1.

The magnitude is small if

- $l(w)$ is close to 1, which means the new w is not very informative for distinguishing two distributions,
- π_{t-1} is close to either 0 or 1, which means the priori is strong.

Will an unemployed worker accept an offer earlier or not, when the actual ruling distribution is g instead of f?

Two countervailing effects are at work.

- if f generates successive wage offers, then w is more likely to be low, but π is moving up toward to 1, which lowers the reservation wage, i.e., the worker becomes less selective the longer he or she remains unemployed.
- if g generates wage offers, then w is more likely to be high, but π is moving downward toward 0, increasing the reservation wage, i.e., the worker becomes more selective the longer he or she remains unemployed.

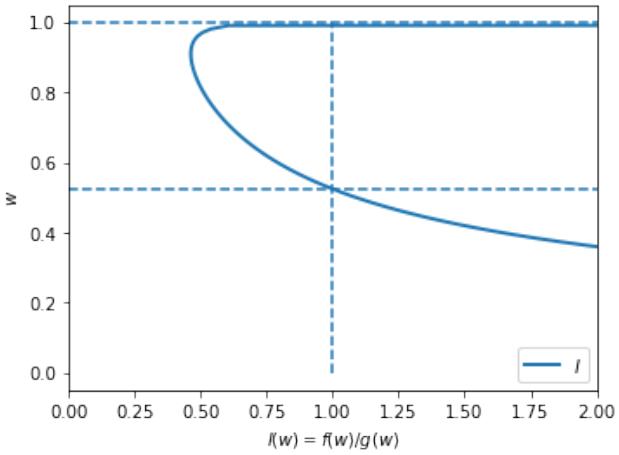
Quantitatively, the lower right figure sheds light on which effect dominates in this example.

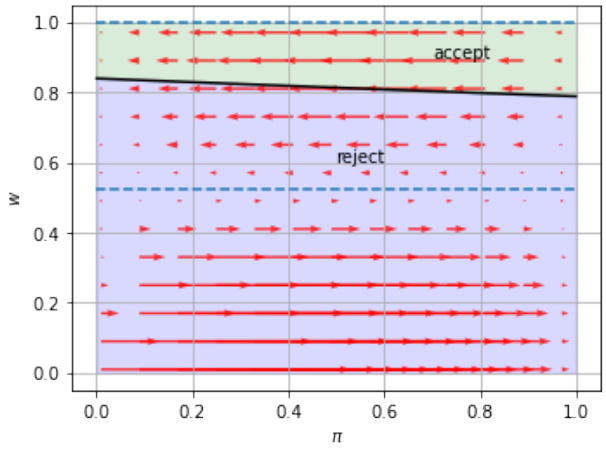
It shows the probability that a previously unemployed worker accepts an offer at different values of π when f or g generates wage offers.

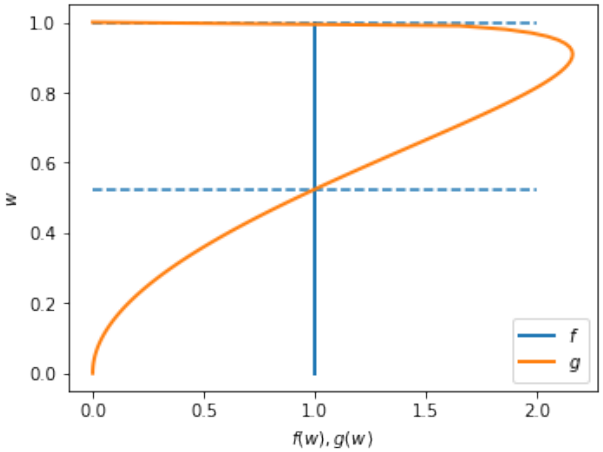
That graph shows that for the particular f and g in this example, the worker is always more likely to accept an offer when f generates the data even when π is close to zero so that the worker believes the true distribution is g and therefore is relatively more selective.

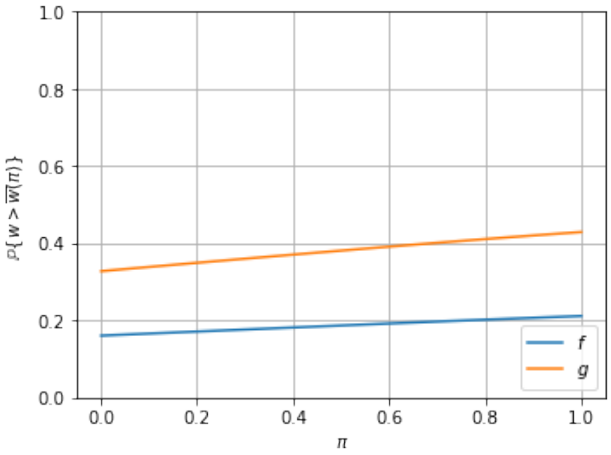
The empirical cumulative distribution of the duration of unemployment verifies our conjecture.

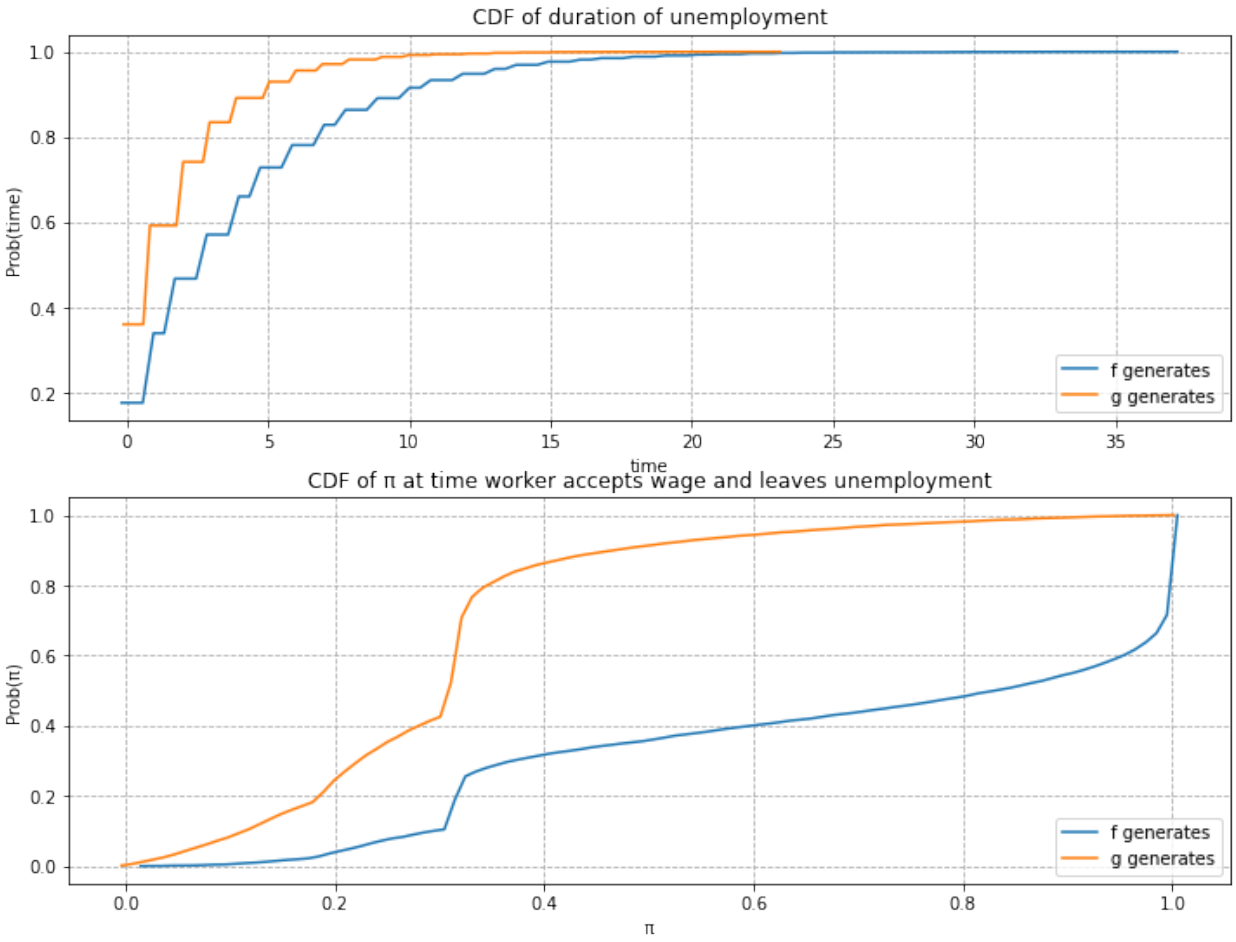
```
job_search_example()
```


w
l(w) = f(w)/g(w)
l


accept
reject
w
π


w
f(w), g(w)
f
g


$\rho\{w > \bar{w}(\pi)\}$
π
f
g



41.12.2 Example 2

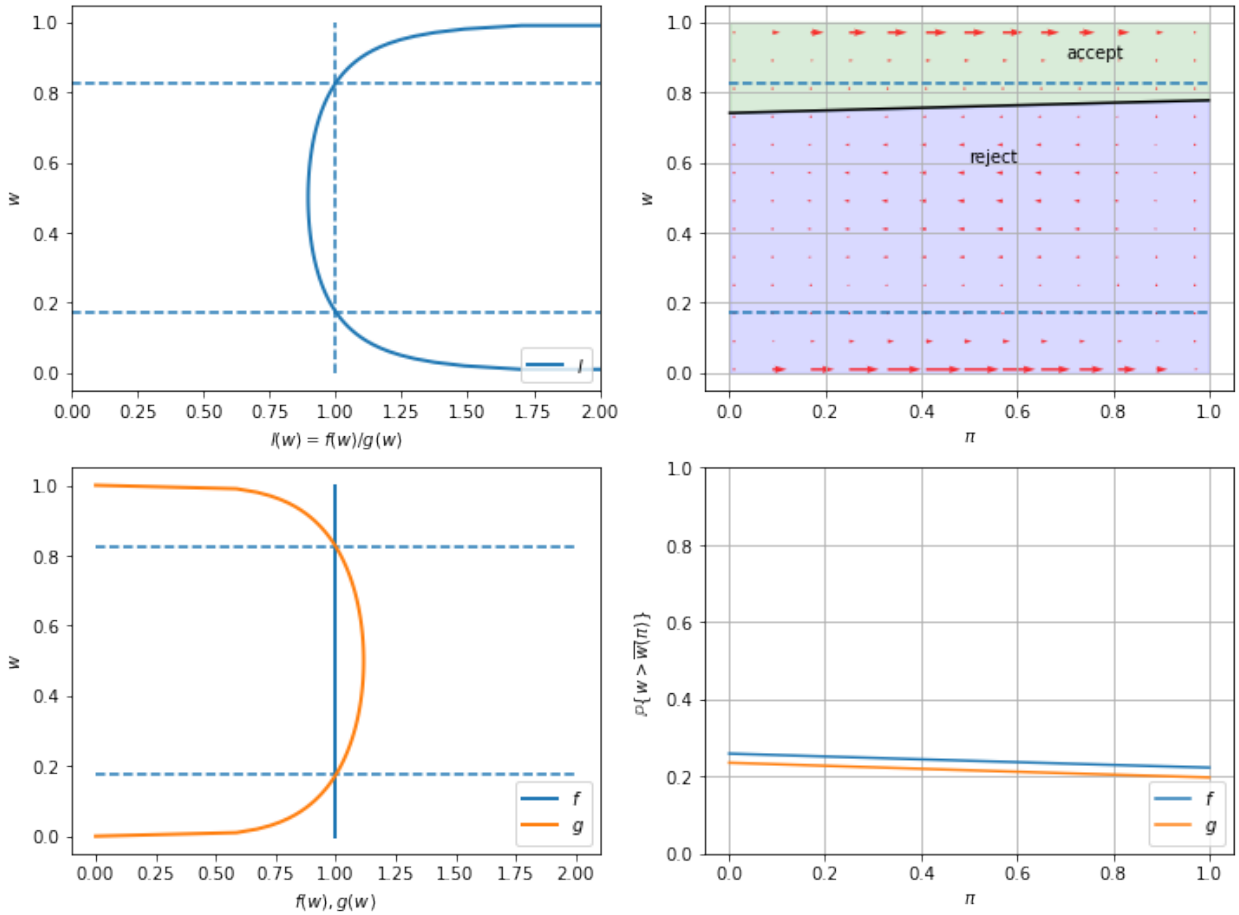
F ~ Beta(1, 1), G ~ Beta(1.2, 1.2), c=0.3.

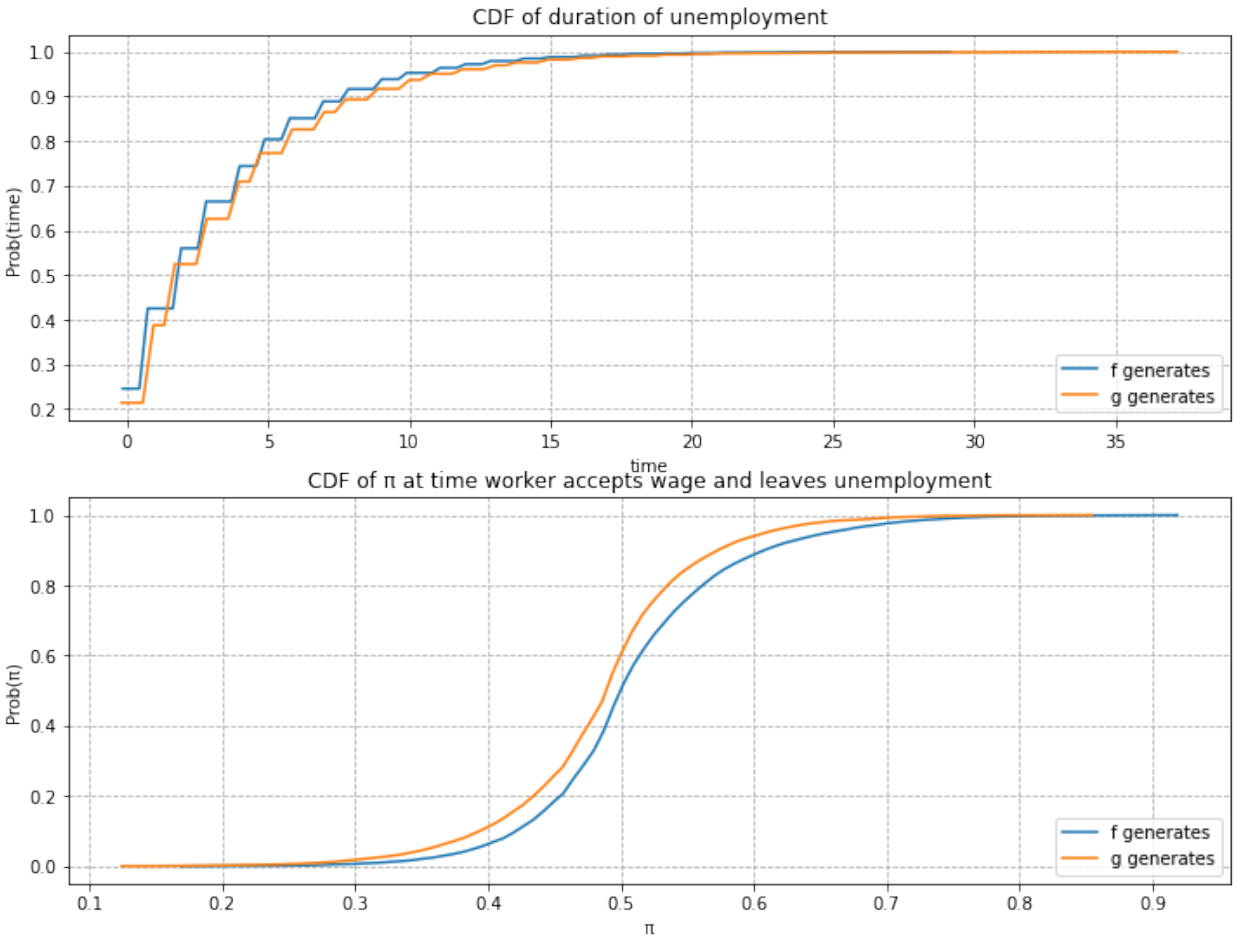
Now G has the same mean as F with a smaller variance.

Since the unemployment compensation c serves as a lower bound for bad wage offers, G is now an "inferior" distribution to F.

Consequently, we observe that the optimal policy $\overline{w}(\pi)$ is increasing in π.

```
job_search_example(1, 1, 1.2, 1.2, 0.3)
```


w
l(w) = f(w)/g(w)
l
accept
reject
π
f(w), g(w)
f
g
$p\{w > \bar{w}(\pi)\}$

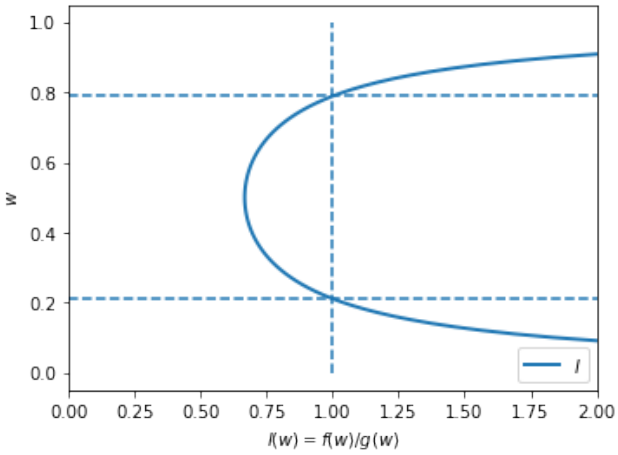


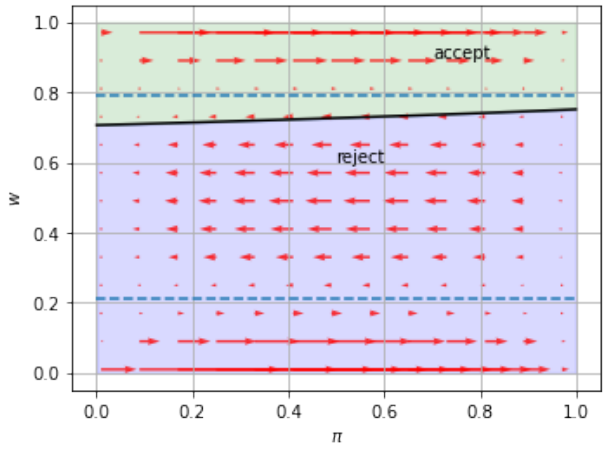
41.12.3 Example 3

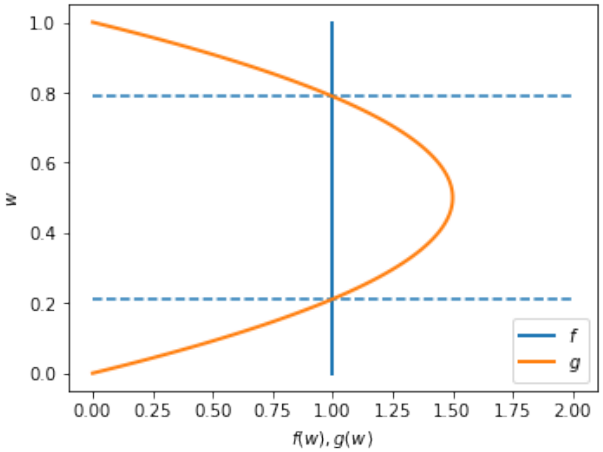
F ~ Beta(1, 1), G ~ Beta(2, 2), c=0.3.

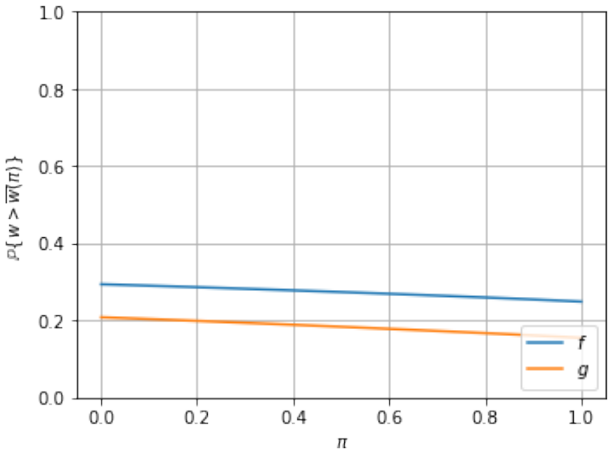
If the variance of G is smaller, we observe in the result that G is even more "inferior" and the slope of $\overline{w}(\pi)$ is larger.

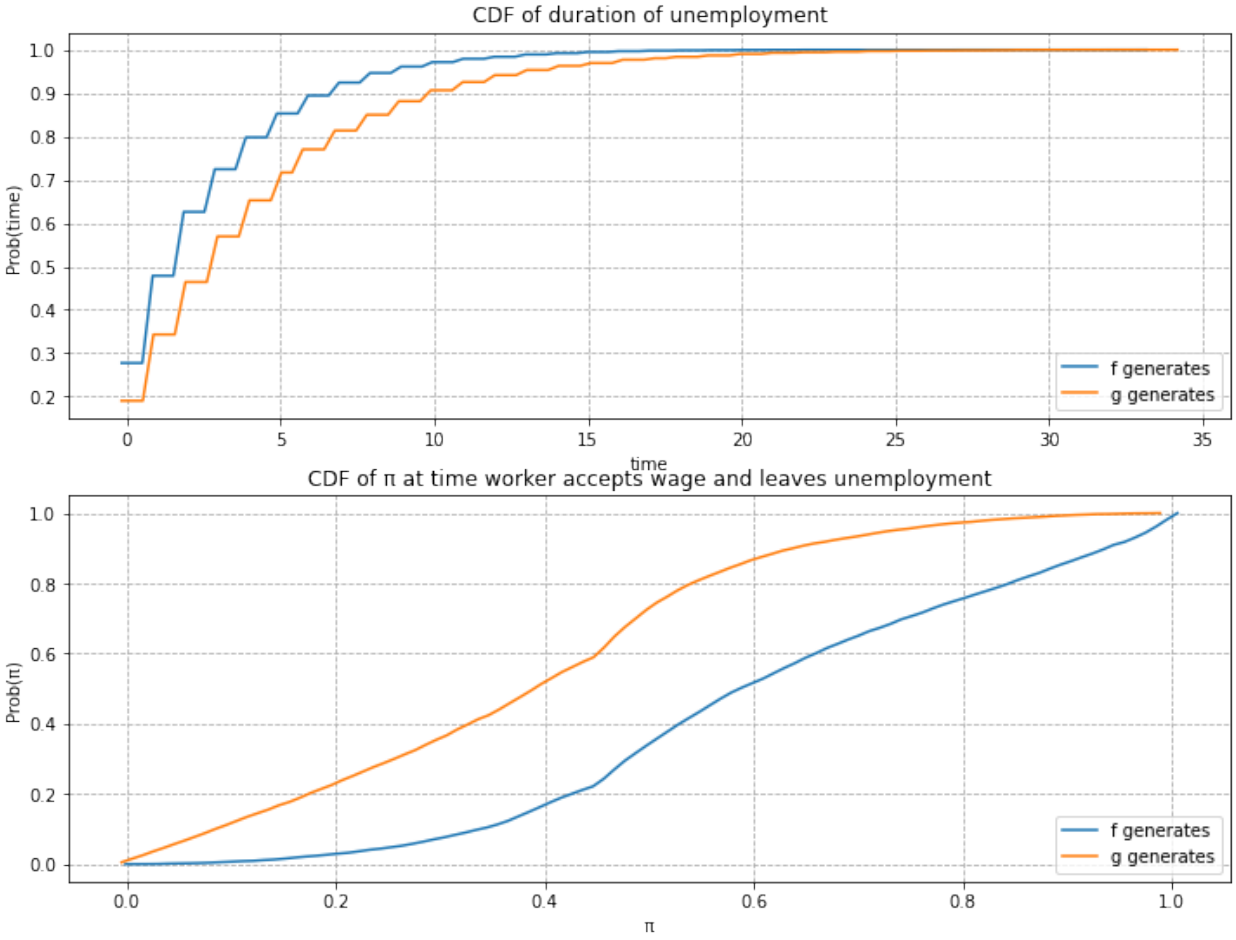
```
job_search_example(1, 1, 2, 2, 0.3)
```


w
l(w) = f(w)/g(w)
l


accept
reject
w
π


w
f(w), g(w)
f
g


$\rho\{w > \bar{w}(\pi)\}$
π
f
g



41.12.4 Example 4

F ~ Beta(1, 1), G ~ Beta(3, 1.2), and c=0.8.

In this example, we keep the parameters of beta distributions to be the same with the baseline case but increase the unemployment compensation c.

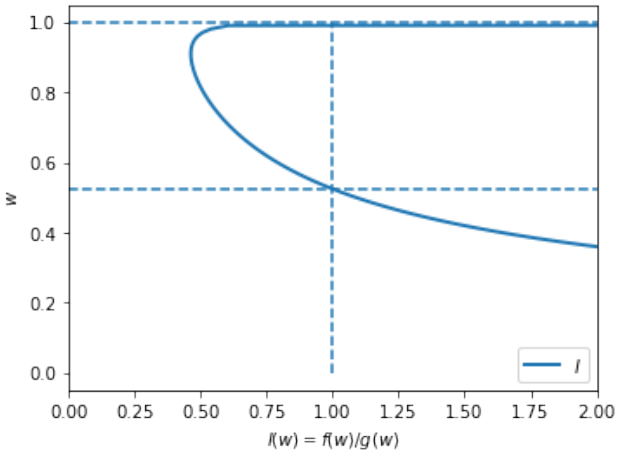
Comparing outcomes to the baseline case (example 1) in which unemployment compensation if low (c=0.3), now the worker can afford a longer learning period.

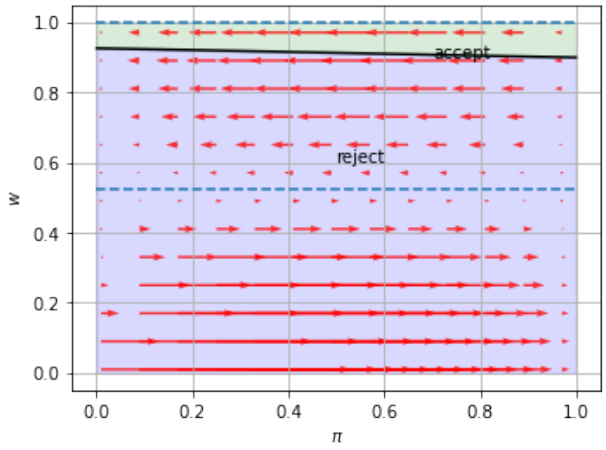
As a result, the worker tends to accept wage offers much later.

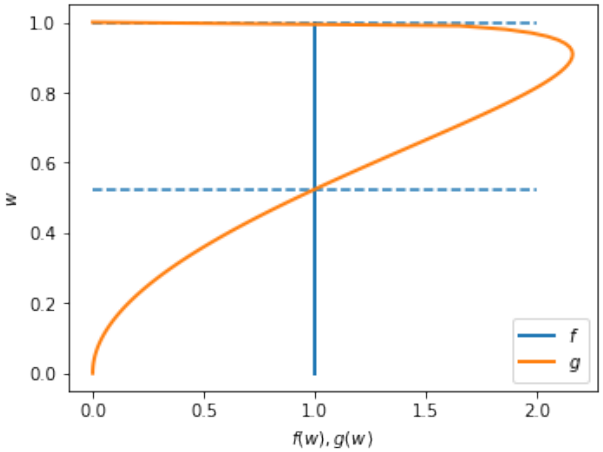
Furthermore, at the time of accepting employment, the belief π is closer to either 0 or 1.

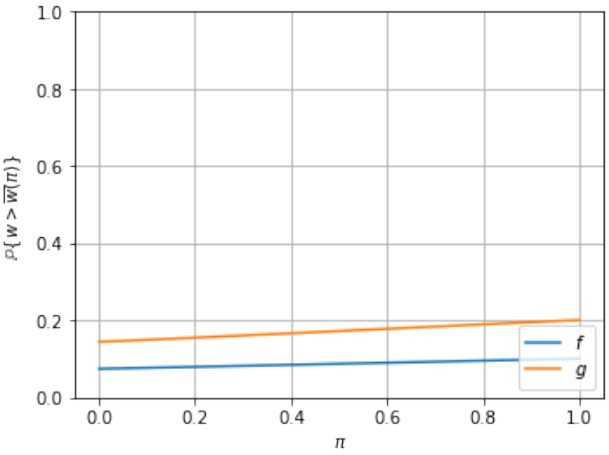
That means that the worker has a better idea about what the true distribution is when he eventually chooses to accept a wage offer.

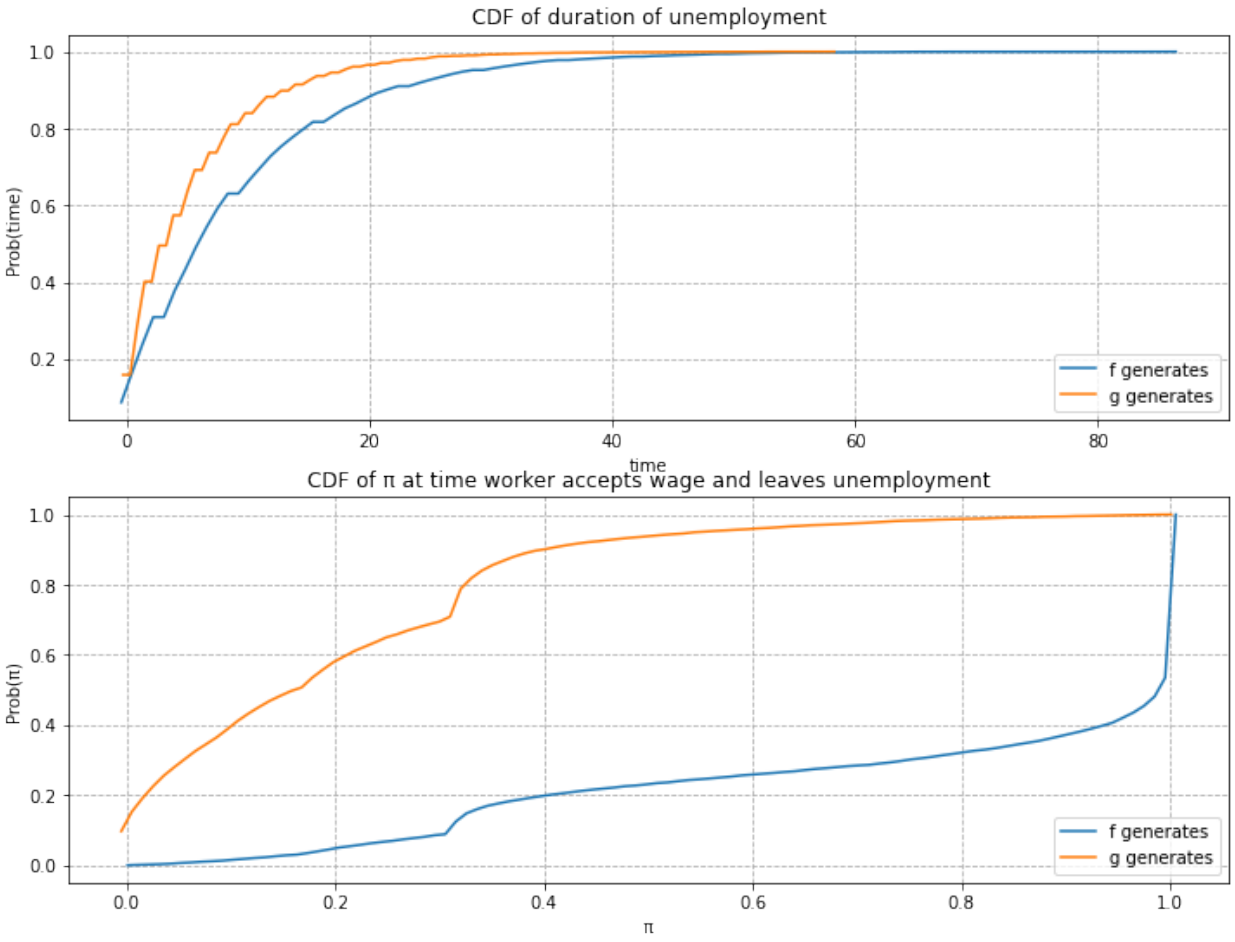
```
job_search_example(1, 1, 3, 1.2, c=0.8)
```


1.0
0.8
0.6
0.4
0.2
0.0
w
0.00 0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00
l(w) = f(w)/g(w)
l


1.0
0.8
0.6
0.4
0.2
0.0
w
accept
reject
0.0 0.2 0.4 0.6 0.8 1.0
π


1.0
0.8
0.6
0.4
0.2
0.0
w
0.0 0.5 1.0 1.5 2.0
f(w), g(w)
f
g


1.0
0.8
0.6
0.4
0.2
0.0
$p\{w > \bar{w}(\pi)\}$
0.0 0.2 0.4 0.6 0.8 1.0
π
f
g

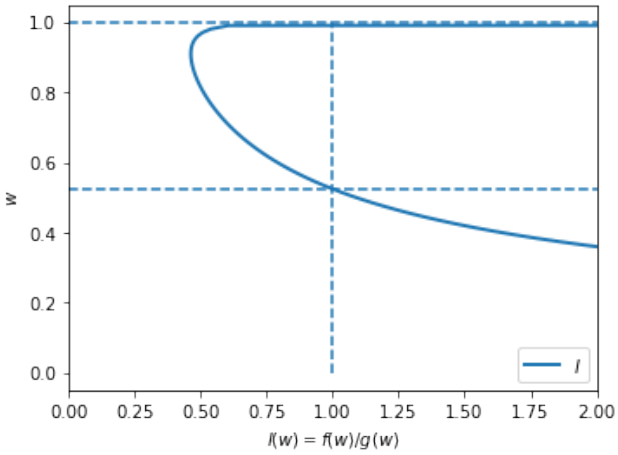


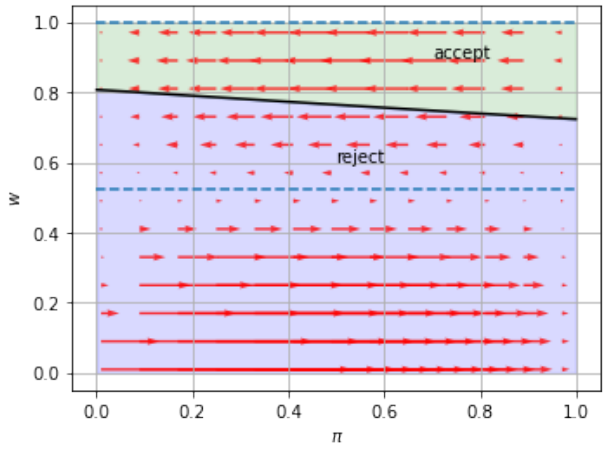
41.12.5 Example 5

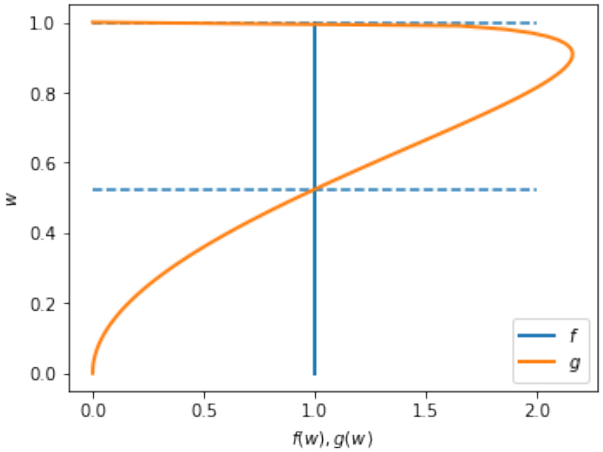
F ~ Beta(1, 1), G ~ Beta(3, 1.2), and c=0.1.

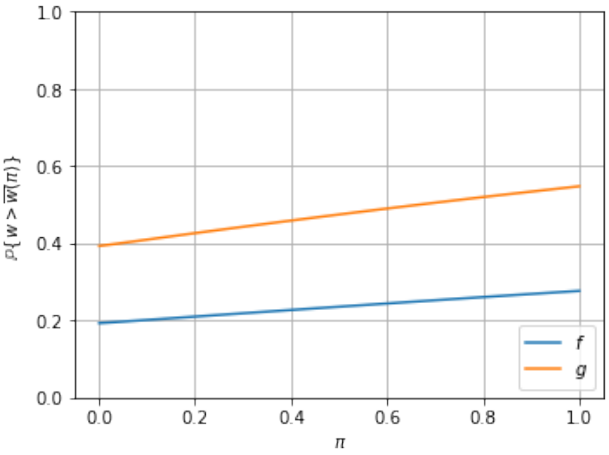
As expected, a smaller c makes an unemployed worker accept wage offers earlier after having acquired less information about the wage distribution.

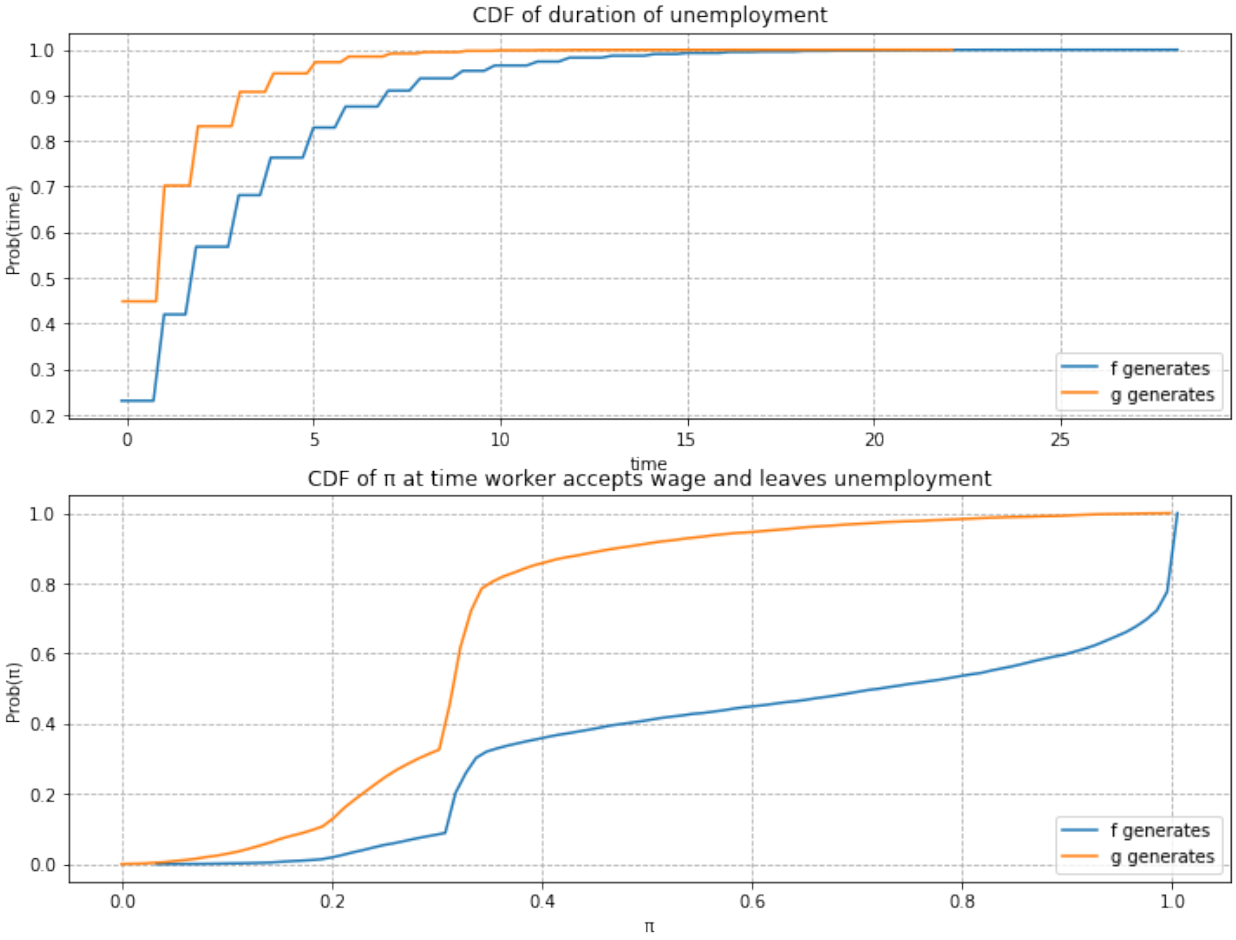
```
job_search_example(1, 1, 3, 1.2, c=0.1)
```


1.0
0.8
0.6
0.4
0.2
0.0
w
0.00 0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00
l(w) = f(w)/g(w)
l


1.0
0.8
0.6
0.4
0.2
0.0
w
accept
reject
0.0 0.2 0.4 0.6 0.8 1.0
π


1.0
0.8
0.6
0.4
0.2
0.0
w
0.0 0.5 1.0 1.5 2.0
f(w), g(w)
f
g


1.0
0.8
0.6
0.4
0.2
0.0
$p\{w > \bar{w}(\pi)\}$
0.0 0.2 0.4 0.6 0.8 1.0
π
f
g


CDF of duration of unemployment
Prob(time)
time
f generates
g generates
CDF of π at time worker accepts wage and leaves unemployment
Prob(π)
π
f generates
g generates

CHAPTER

FORTYTWO

LIKELIHOOD RATIO PROCESSES

Contents

42.1 Overview

This lecture describes likelihood ratio processes and some of their uses.

We'll use a setting described in *this lecture*.

Among things that we'll learn are

- A peculiar property of likelihood ratio processes
- How a likelihood ratio process is a key ingredient in frequentist hypothesis testing
- How a **receiver operator characteristic curve** summarizes information about a false alarm probability and power in frequentist hypothesis testing
- How during World War II the United States Navy devised a decision rule that Captain Garret L. Schyler challenged and asked Milton Friedman to justify to him, a topic to be studied in *this lecture*

Let's start by importing some Python tools.

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from numba import vectorize, njit
```

(continues on next page)

(continued from previous page)

```
from math import gamma
from scipy.integrate import quad
```

42.2 Likelihood Ratio Process

A nonnegative random variable W has one of two probability density functions, either f or g.

Before the beginning of time, nature once and for all decides whether she will draw a sequence of IID draws from either f or g.

We will sometimes let q be the density that nature chose once and for all, so that q is either f or g, permanently.

Nature knows which density it permanently draws from, but we the observers do not.

We do know both f and g but we don't know which density nature chose.

But we want to know.

To do that, we use observations.

We observe a sequence $\{w_t\}_{t=1}^T$ of T IID draws from either f or g.

We want to use these observations to infer whether nature chose f or g.

A **likelihood ratio process** is a useful tool for this task.

To begin, we define key component of a likelihood ratio process, namely, the time t likelihood ratio as the random variable

$$\ell(w_t) = \frac{f(w_t)}{g(w_t)}, \quad t \geq 1.$$

We assume that f and g both put positive probabilities on the same intervals of possible realizations of the random variable W.

That means that under the g density, $\ell(w_t) = \frac{f(w_t)}{g(w_t)}$ is evidently a nonnegative random variable with mean 1.

A **likelihood ratio process** for sequence $\{w_t\}_{t=1}^\infty$ is defined as

$$L(w^t) = \prod_{i=1}^t \ell(w_i),$$

where $w^t = \{w_1, \ldots, w_t\}$ is a history of observations up to and including time t.

Sometimes for shorthand we'll write $L_t = L(w^t)$.

Notice that the likelihood process satisfies the *recursion* or *multiplicative decomposition*

$$L(w^t) = \ell(w_t) L(w^{t-1}).$$

The likelihood ratio and its logarithm are key tools for making inferences using a classic frequentist approach due to Neyman and Pearson [NP33].

To help us appreciate how things work, the following Python code evaluates f and g as two different beta distributions, then computes and simulates an associated likelihood ratio process by generating a sequence w^t from one of the two probability distributionss, for example, a sequence of IID draws from g.

```
# Parameters in the two beta distributions.
F_a, F_b = 1, 1
G_a, G_b = 3, 1.2

@vectorize
def p(x, a, b):
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * x** (a-1) * (1 - x) ** (b-1)

# The two density functions.
f = njit(lambda x: p(x, F_a, F_b))
g = njit(lambda x: p(x, G_a, G_b))
```

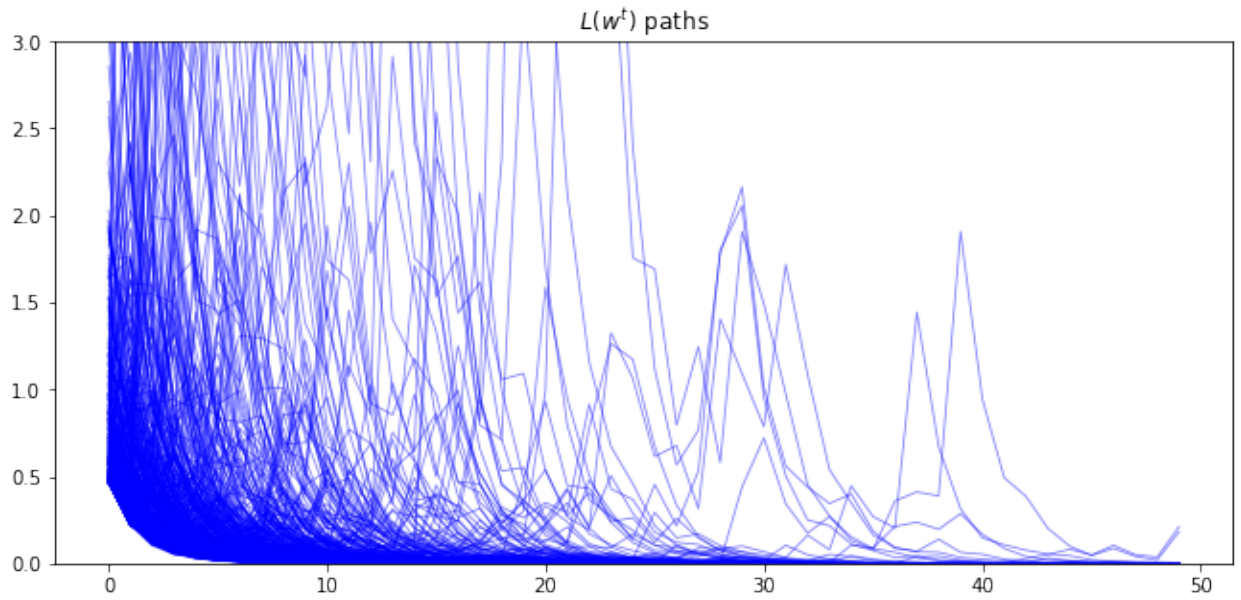
```
@njit
def simulate(a, b, T=50, N=500):
    '''
    Generate N sets of T observations of the likelihood ratio,
    return as N x T matrix.

    '''

    l_arr = np.empty((N, T))

    for i in range(N):

        for j in range(T):
            w = np.random.beta(a, b)
            l_arr[i, j] = f(w) / g(w)

    return l_arr
```

42.3 Nature Permanently Draws from Density g

We first simulate the likelihood ratio process when nature permanently draws from g.

```
l_arr_g = simulate(G_a, G_b)
l_seq_g = np.cumprod(l_arr_g, axis=1)
```

```
N, T = l_arr_g.shape

for i in range(N):

    plt.plot(range(T), l_seq_g[i, :], color='b', lw=0.8, alpha=0.5)

plt.ylim([0, 3])
plt.title("$L(w^{t})$ paths");
```

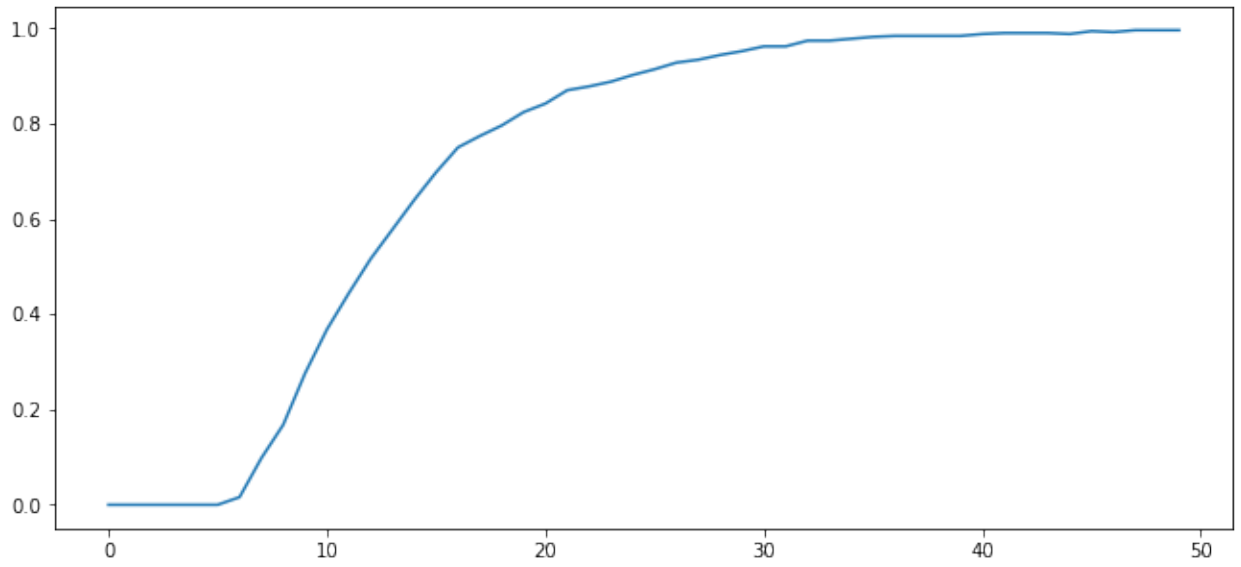


Evidently, as sample length T grows, most probability mass shifts toward zero

To see it this more clearly clearly, we plot over time the fraction of paths $L\left(w^t\right)$ that fall in the interval $[0, 0.01]$.

```
plt.plot(range(T), np.sum(l_seq_g <= 0.01, axis=0) / N)
```

```
[<matplotlib.lines.Line2D at 0x7fa27ce080d0>]
```



Despite the evident convergence of most probability mass to a very small interval near 0, the unconditional mean of $L\left(w^t\right)$ under probability density g is identically 1 for all t.

To verify this assertion, first notice that as mentioned earlier the unconditional mean $E\left[\ell\left(w_t\right) \mid q=g\right]$ is 1 for all t:

$$\begin{aligned} E\left[\ell\left(w_t\right) \mid q=g\right] &= \int \frac{f\left(w_t\right)}{g\left(w_t\right)} g\left(w_t\right) dw_t \\ &= \int f\left(w_t\right) dw_t \\ &= 1, \end{aligned}$$

which immediately implies

$$\begin{aligned} E\left[L\left(w^1\right) \mid q=g\right] &= E\left[\ell\left(w_1\right) \mid q=g\right] \\ &= 1. \end{aligned}$$

Because $L(w^t) = \ell(w_t) L(w^{t-1})$ and $\{w_t\}_{t=1}^t$ is an IID sequence, we have

$$\begin{aligned} E\left[L\left(w^t\right) \mid q=g\right] &= E\left[L\left(w^{t-1}\right) \ell\left(w_t\right) \mid q=g\right] \\ &= E\left[L\left(w^{t-1}\right) E\left[\ell\left(w_t\right) \mid q=g, w^{t-1}\right] \mid q=g\right] \\ &= E\left[L\left(w^{t-1}\right) E\left[\ell\left(w_t\right) \mid q=g\right] \mid q=g\right] \\ &= E\left[L\left(w^{t-1}\right) \mid q=g\right] \end{aligned}$$

for any $t \geq 1$.

Mathematical induction implies $E\left[L\left(w^t\right) \mid q=g\right] = 1$ for all $t \geq 1$.

42.4 Peculiar Property of Likelihood Ratio Process

How can $E\left[L\left(w^t\right) \mid q=g\right] = 1$ possibly be true when most probability mass of the likelihood ratio process is piling up near 0 as $t \to +\infty$?

The answer has to be that as $t \to +\infty$, the distribution of L_t becomes more and more fat-tailed: enough mass shifts to larger and larger values of L_t to make the mean of L_t continue to be one despite most of the probability mass piling up near 0.

To illustrate this peculiar property, we simulate many paths and calculate the unconditional mean of $L\left(w^t\right)$ by averaging across these many paths at each t.

```
l_arr_g = simulate(G_a, G_b, N=50000)
l_seq_g = np.cumprod(l_arr_g, axis=1)
```

It would be useful to use simulations to verify that unconditional means $E\left[L\left(w^t\right)\right]$ equal unity by averaging across sample paths.

But it would be too computer-time-consuming for us to that here simply by applying a standard Monte Carlo simulation approach.

The reason is that the distribution of $L\left(w^t\right)$ is extremely skewed for large values of t.

Because the probability density in the right tail is close to 0, it just takes too much computer time to sample enough points from the right tail.

We explain the problem in more detail in *this lecture*.

There we describe a way to an alternative way to compute the mean of a likelihood ratio by computing the mean of a *different* random variable by sampling from a *different* probability distribution.

42.5 Nature Permanently Draws from Density f

Now suppose that before time 0 nature permanently decided to draw repeatedly from density f.

While the mean of the likelihood ratio $\ell(w_t)$ under density g is 1, its mean under the density f exceeds one.

To see this, we compute

$$
\begin{aligned}
E[\ell(w_t) \mid q = f] &= \int \frac{f(w_t)}{g(w_t)} f(w_t)\, dw_t \\
&= \int \frac{f(w_t)}{g(w_t)} \frac{f(w_t)}{g(w_t)} g(w_t)\, dw_t \\
&= \int \ell(w_t)^2 g(w_t)\, dw_t \\
&= E\left[\ell(w_t)^2 \mid q = g\right] \\
&= E[\ell(w_t) \mid q = g]^2 + Var(\ell(w_t) \mid q = g) \\
&> E[\ell(w_t) \mid q = g]^2 = 1
\end{aligned}
$$

This in turn implies that the unconditional mean of the likelihood ratio process $L(w^t)$ diverges toward $+\infty$.

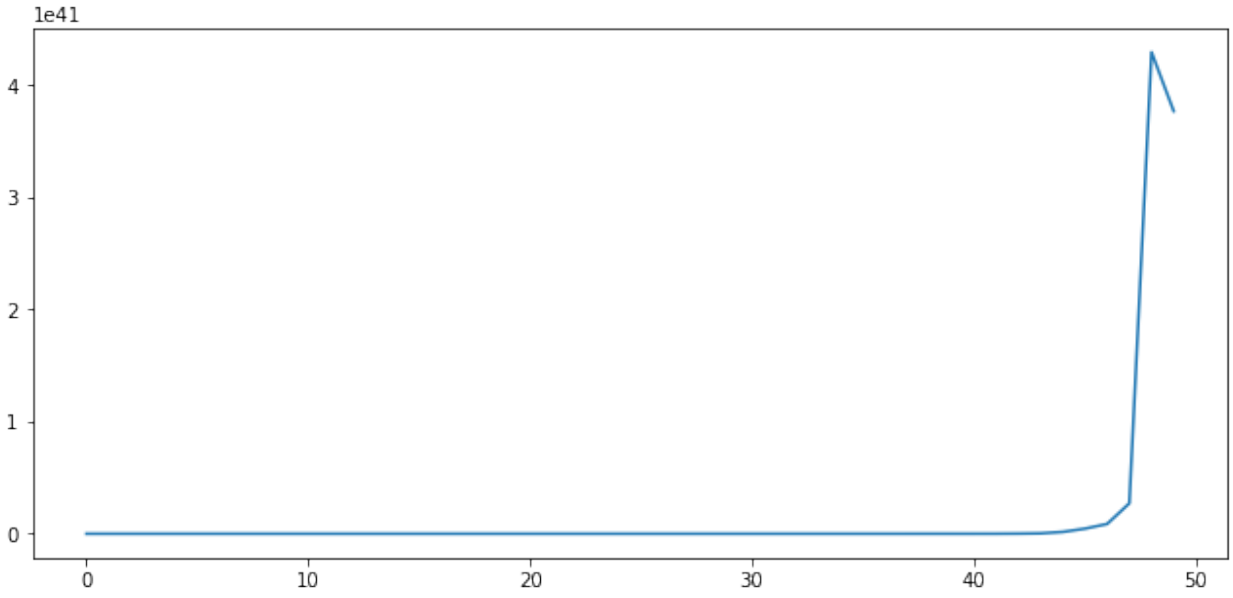
Simulations below confirm this conclusion.

Please note the scale of the y axis.

```
l_arr_f = simulate(F_a, F_b, N=50000)
l_seq_f = np.cumprod(l_arr_f, axis=1)
```

```
N, T = l_arr_f.shape
plt.plot(range(T), np.mean(l_seq_f, axis=0))
```

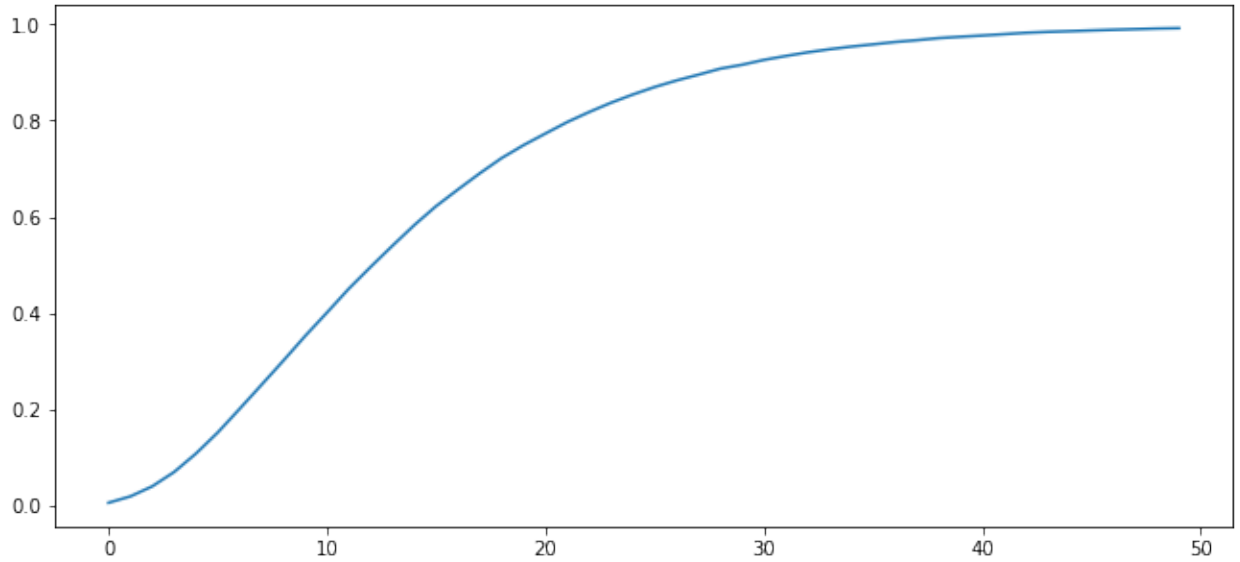
```
[<matplotlib.lines.Line2D at 0x7fa27fde2b80>]
```



We also plot the probability that $L(w^t)$ falls into the interval $[10000, \infty)$ as a function of time and watch how fast probability mass diverges to $+\infty$.

```
plt.plot(range(T), np.sum(l_seq_f > 10000, axis=0) / N)
```

```
[<matplotlib.lines.Line2D at 0x7fa27ff537c0>]
```



42.6 Likelihood Ratio Test

We now describe how to employ the machinery of Neyman and Pearson [NP33] to test the hypothesis that history w^t is generated by repeated IID draws from density g.

Denote q as the data generating process, so that $q = f$ or g.

Upon observing a sample $\{W_i\}_{i=1}^t$, we want to decide whether nature is drawing from g or from f by performing a (frequentist) hypothesis test.

We specify

- Null hypothesis H_0: $q = f$,
- Alternative hypothesis H_1: $q = g$.

Neyman and Pearson proved that the best way to test this hypothesis is to use a **likelihood ratio test** that takes the form:

- reject H_0 if $L(W^t) < c$,
- accept H_0 otherwise.

where c is a given discrimination threshold, to be chosen in a way we'll soon describe.

This test is *best* in the sense that it is a **uniformly most powerful** test.

To understand what this means, we have to define probabilities of two important events that allow us to characterize a test associated with a given threshold c.

The two probabilities are:

- Probability of detection (= power = 1 minus probability of Type II error):

$$1 - \beta \equiv \Pr\{L(w^t) < c \mid q = g\}$$

- Probability of false alarm (= significance level = probability of Type I error):

$$\alpha \equiv \Pr\{L(w^t) < c \mid q = f\}$$

The Neyman-Pearson Lemma states that among all possible tests, a likelihood ratio test maximizes the probability of detection for a given probability of false alarm.

Another way to say the same thing is that among all possible tests, a likelihood ratio test maximizes **power** for a given **significance level**.

To have made a good inference, we want a small probability of false alarm and a large probability of detection.

With sample size t fixed, we can change our two probabilities by adjusting c.

A troublesome "that's life" fact is that these two probabilities move in the same direction as we vary the critical value c.

Without specifying quantitative losses from making Type I and Type II errors, there is little that we can say about how we *should* trade off probabilities of the two types of mistakes.

We do know that increasing sample size t improves statistical inference.

Below we plot some informative figures that illustrate this.

We also present a classical frequentist method for choosing a sample size t.

Let's start with a case in which we fix the threshold c at 1.

```
c = 1
```

Below we plot empirical distributions of logarithms of the cumulative likelihood ratios simulated above, which are generated by either f or g.

Taking logarithms has no effect on calculating the probabilities because the log is a monotonic transformation.

As t increases, the probabilities of making Type I and Type II errors both decrease, which is good.

This is because most of the probability mass of $\log(L(w^t))$ moves toward $-\infty$ when g is the data generating process, ; while $\log(L(w^t))$ goes to ∞ when data are generated by f.

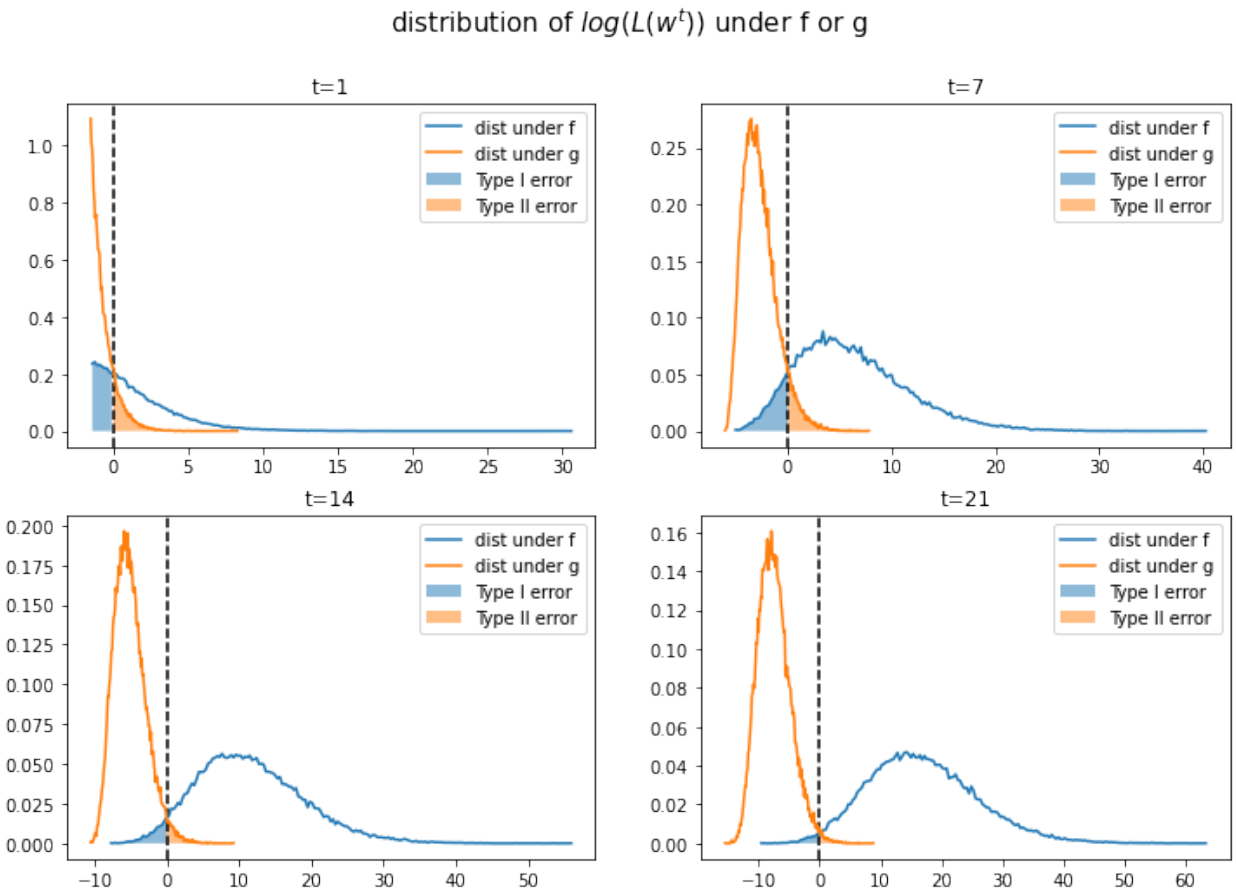
That disparate behavior of $\log(L(w^t))$ under f and q is what makes it possible to distinguish $q = f$ from $q = g$.

```
fig, axs = plt.subplots(2, 2, figsize=(12, 8))
fig.suptitle('distribution of $log(L(w^t))$ under f or g', fontsize=15)

for i, t in enumerate([1, 7, 14, 21]):
    nr = i // 2
    nc = i % 2

    axs[nr, nc].axvline(np.log(c), color="k", ls="--")

    hist_f, x_f = np.histogram(np.log(l_seq_f[:, t]), 200, density=True)
    hist_g, x_g = np.histogram(np.log(l_seq_g[:, t]), 200, density=True)

    axs[nr, nc].plot(x_f[1:], hist_f, label="dist under f")
    axs[nr, nc].plot(x_g[1:], hist_g, label="dist under g")

    for i, (x, hist, label) in enumerate(zip([x_f, x_g], [hist_f, hist_g], ["Type I␣
↪error", "Type II error"])):
        ind = x[1:] <= np.log(c) if i == 0 else x[1:] > np.log(c)
        axs[nr, nc].fill_between(x[1:][ind], hist[ind], alpha=0.5, label=label)

    axs[nr, nc].legend()
```

(continues on next page)

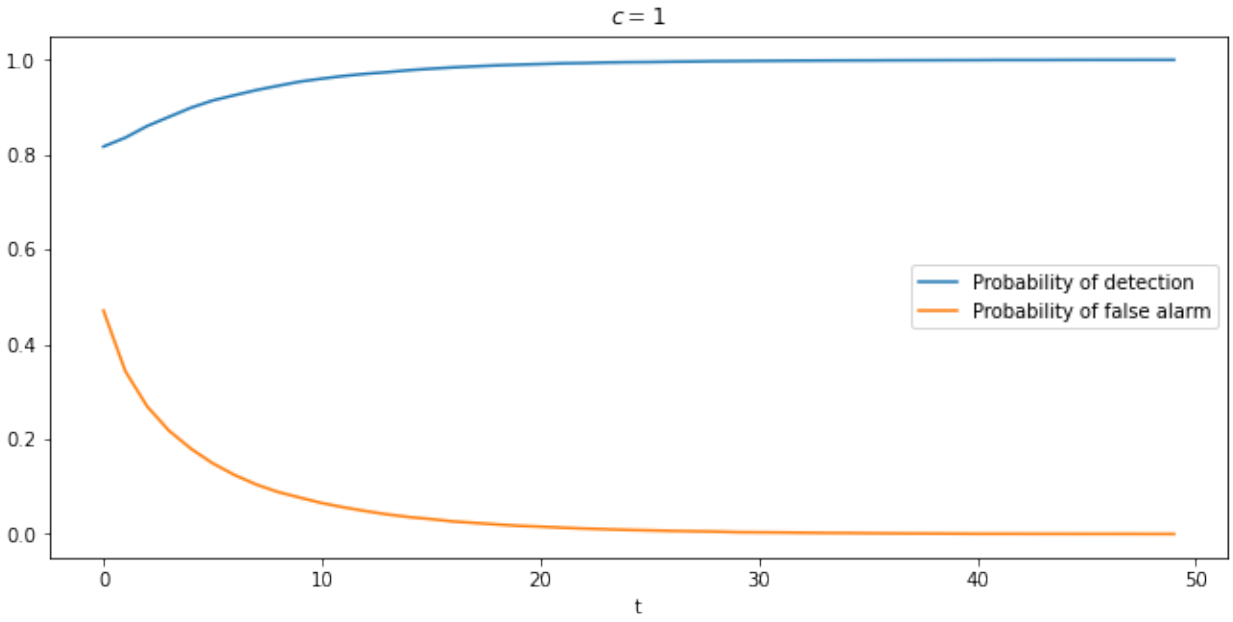
(continued from previous page)

```
    axs[nr, nc].set_title(f"t={t}")

plt.show()
```



The graph below shows more clearly that, when we hold the threshold c fixed, the probability of detection monotonically increases with increases in t and that the probability of a false alarm monotonically decreases.

```
PD = np.empty(T)
PFA = np.empty(T)

for t in range(T):
    PD[t] = np.sum(l_seq_g[:, t] < c) / N
    PFA[t] = np.sum(l_seq_f[:, t] < c) / N

plt.plot(range(T), PD, label="Probability of detection")
plt.plot(range(T), PFA, label="Probability of false alarm")
plt.xlabel("t")
plt.title("$c=1$")
plt.legend()
plt.show()
```



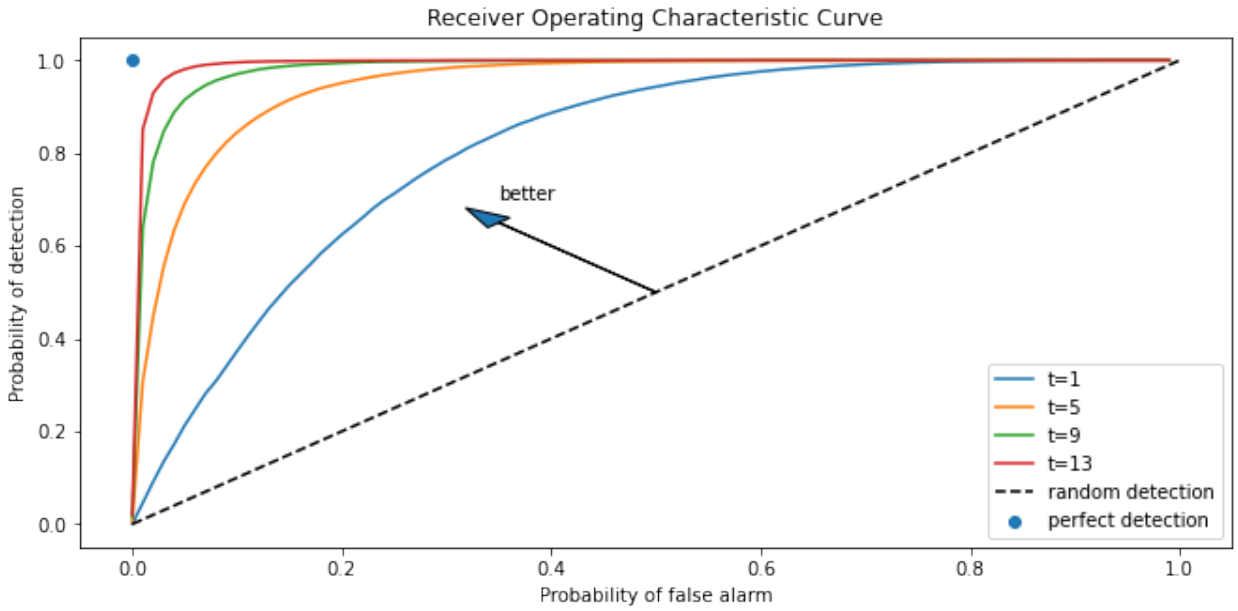
For a given sample size t, the threshold c uniquely pins down probabilities of both types of error.

If for a fixed t we now free up and move c, we will sweep out the probability of detection as a function of the probability of false alarm.

This produces what is called a receiver operating characteristic curve.

Below, we plot receiver operating characteristic curves for different sample sizes t.

```
PFA = np.arange(0, 100, 1)

for t in range(1, 15, 4):
    percentile = np.percentile(l_seq_f[:, t], PFA)
    PD = [np.sum(l_seq_g[:, t] < p) / N for p in percentile]

    plt.plot(PFA / 100, PD, label=f"t={t}")

plt.scatter(0, 1, label="perfect detection")
plt.plot([0, 1], [0, 1], color='k', ls='--', label="random detection")

plt.arrow(0.5, 0.5, -0.15, 0.15, head_width=0.03)
plt.text(0.35, 0.7, "better")
plt.xlabel("Probability of false alarm")
plt.ylabel("Probability of detection")
plt.legend()
plt.title("Receiver Operating Characteristic Curve")
plt.show()
```



Notice that as t increases, we are assured a larger probability of detection and a smaller probability of false alarm associated with a given discrimination threshold c.

As $t \to +\infty$, we approach the perfect detection curve that is indicated by a right angle hinging on the blue dot.

For a given sample size t, the discrimination threshold c determines a point on the receiver operating characteristic curve.

It is up to the test designer to trade off probabilities of making the two types of errors.

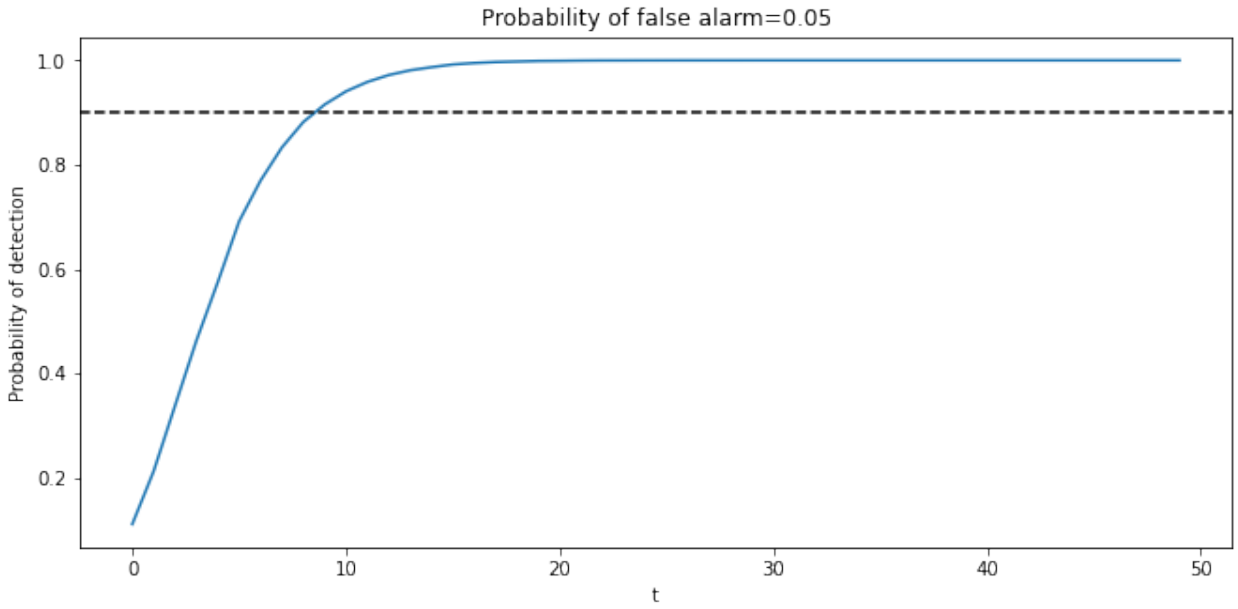
But we know how to choose the smallest sample size to achieve given targets for the probabilities.

Typically, frequentists aim for a high probability of detection that respects an upper bound on the probability of false alarm.

Below we show an example in which we fix the probability of false alarm at 0.05.

The required sample size for making a decision is then determined by a target probability of detection, for example, 0.9, as depicted in the following graph.

```
PFA = 0.05
PD = np.empty(T)

for t in range(T):

    c = np.percentile(l_seq_f[:, t], PFA * 100)
    PD[t] = np.sum(l_seq_g[:, t] < c) / N

plt.plot(range(T), PD)
plt.axhline(0.9, color="k", ls="--")

plt.xlabel("t")
plt.ylabel("Probability of detection")
plt.title(f"Probability of false alarm={PFA}")
plt.show()
```



The United States Navy evidently used a procedure like this to select a sample size t for doing quality control tests during World War II.

A Navy Captain who had been ordered to perform tests of this kind had doubts about it that he presented to Milton Friedman, as we describe in *this lecture*.

42.7 Kullback–Leibler divergence

Now let's consider a case in which neither g nor f generates the data.

Instead, a third distribution h does.

Let's watch how how the cumulated likelihood ratios f/g behave when h governs the data.

A key tool here is called **Kullback–Leibler divergence**.

It is also called **relative entropy**.

It measures how one probability distribution differs from another.

In our application, we want to measure how f or g diverges from h

The two Kullback–Leibler divergences pertinent for us are K_f and K_g defined as

$$\begin{aligned} K_f &= E_h \left[\log \left(\frac{f(w)}{h(w)} \right) \frac{f(w)}{h(w)} \right] \\ &= \int \log \left(\frac{f(w)}{h(w)} \right) \frac{f(w)}{h(w)} h(w) \, dw \\ &= \int \log \left(\frac{f(w)}{h(w)} \right) f(w) \, dw \end{aligned}$$

$$
\begin{aligned}
K_g &= E_h \left[\log \left(\frac{g(w)}{h(w)} \right) \frac{g(w)}{h(w)} \right] \\
&= \int \log \left(\frac{g(w)}{h(w)} \right) \frac{g(w)}{h(w)} h(w) \, dw \\
&= \int \log \left(\frac{g(w)}{h(w)} \right) g(w) \, dw
\end{aligned}
$$

When $K_g < K_f$, g is closer to h than f is.

- In that case we'll find that $L(w^t) \to 0$.

When $K_g > K_f$, f is closer to h than g is.

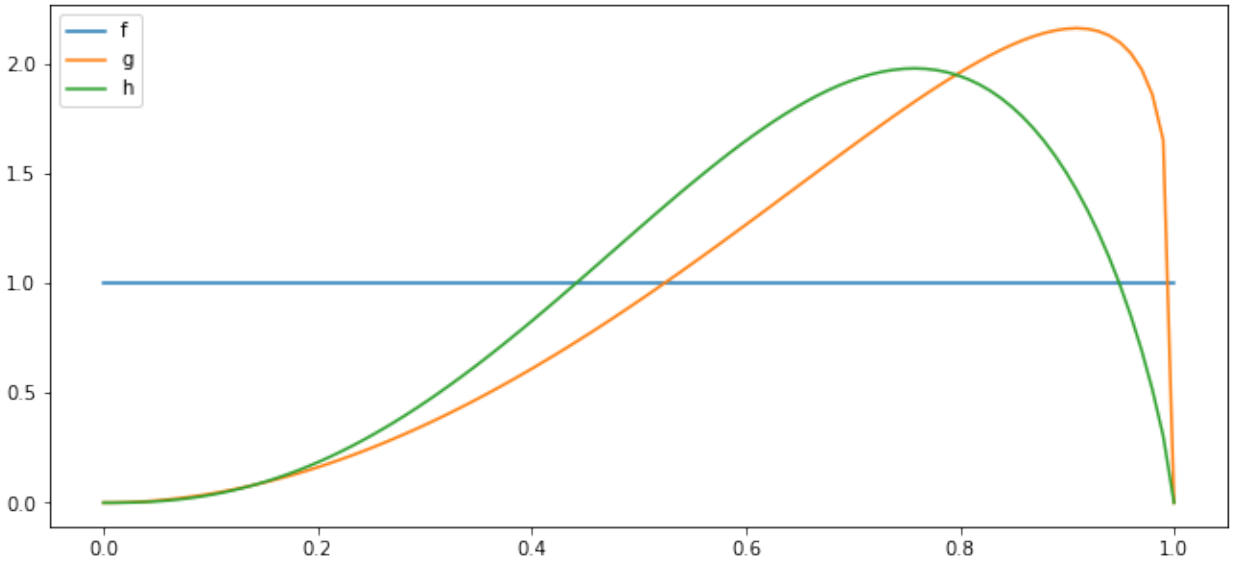
- In that case we'll find that $L(w^t) \to +\infty$

We'll now experiment with an h is also a beta distribution

We'll start by setting parameters G_a and G_b so that h is closer to g

```
H_a, H_b = 3.5, 1.8

h = njit(lambda x: p(x, H_a, H_b))
```

```
x_range = np.linspace(0, 1, 100)
plt.plot(x_range, f(x_range), label='f')
plt.plot(x_range, g(x_range), label='g')
plt.plot(x_range, h(x_range), label='h')

plt.legend()
plt.show()
```



Let's compute the Kullback–Leibler discrepancies by quadrature integration.

```
def KL_integrand(w, q, h):

    m = q(w) / h(w)

    return np.log(m) * q(w)
```

```
def compute_KL(h, f, g):

    Kf, _ = quad(KL_integrand, 0, 1, args=(f, h))
    Kg, _ = quad(KL_integrand, 0, 1, args=(g, h))

    return Kf, Kg
```

```
Kf, Kg = compute_KL(h, f, g)
Kf, Kg
```

```
(0.7902536603660161, 0.08554075759988769)
```

We have $K_g < K_f$.

Next, we can verify our conjecture about $L\left(w^t\right)$ by simulation.

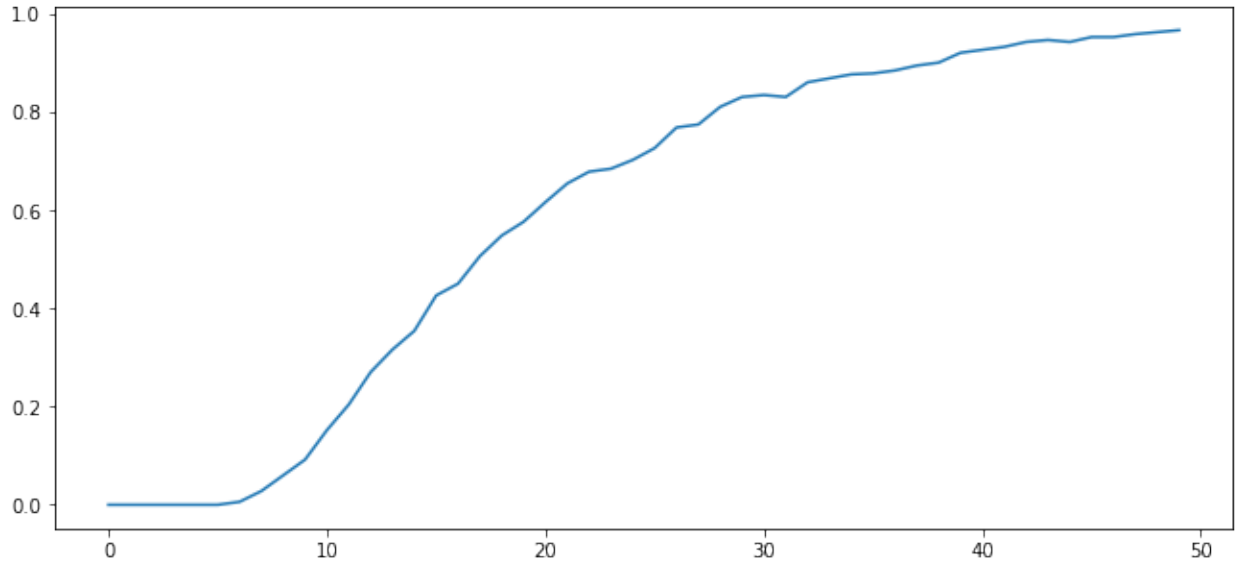
```
l_arr_h = simulate(H_a, H_b)
l_seq_h = np.cumprod(l_arr_h, axis=1)
```

The figure below plots over time the fraction of paths $L\left(w^t\right)$ that fall in the interval $[0, 0.01]$.

Notice that it converges to 1 as expected when g is closer to h than f is.

```
N, T = l_arr_h.shape
plt.plot(range(T), np.sum(l_seq_h <= 0.01, axis=0) / N)
```

```
[<matplotlib.lines.Line2D at 0x7fa280005070>]
```



We can also try an h that is closer to f than is g so that now K_g is larger than K_f.

```
H_a, H_b = 1.2, 1.2
h = njit(lambda x: p(x, H_a, H_b))
```

```
Kf, Kg = compute_KL(h, f, g)
Kf, Kg
```

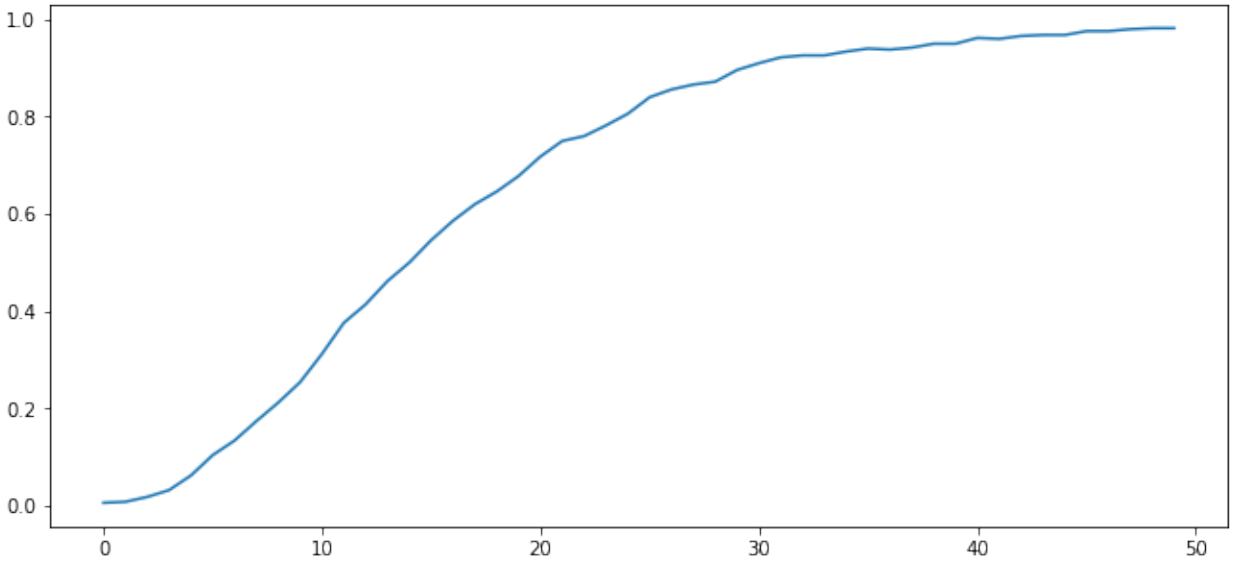
```
(0.01239249754452668, 0.35377684280997646)
```

```
l_arr_h = simulate(H_a, H_b)
l_seq_h = np.cumprod(l_arr_h, axis=1)
```

Now probability mass of $L\left(w^t\right)$ falling above 10000 diverges to $+\infty$.

```
N, T = l_arr_h.shape
plt.plot(range(T), np.sum(l_seq_h > 10000, axis=0) / N)
```

```
[<matplotlib.lines.Line2D at 0x7fa27ef0e9a0>]
```



42.8 Sequels

Likelihood processes play an important role in Bayesian learning, as described in *this lecture* and as applied in *this lecture*.

Likelihood ratio processes appear again in this lecture, which contains another illustration of the **peculiar property** of likelihood ratio processes described above.

CHAPTER

FORTYTHREE

COMPUTING MEAN OF A LIKELIHOOD RATIO PROCESS

Contents

43.1 Overview

In *this lecture* we described a peculiar property of a likelihood ratio process, namely, that it's mean equals one for all $t \geq 0$ despite it's converging to zero almost surely.

While it is easy to verify that peculiar properly analytically (i.e., in population), it is challenging to use a computer simulation to verify it via an application of a law of large numbers that entails studying sample averages of repeated simulations.

To confront this challenge, this lecture puts **importance sampling** to work to accelerate convergence of sample averages to population means.

We use importance sampling to estimate the mean of a cumulative likelihood ratio $L\left(\omega^t\right) = \prod_{i=1}^{t} \ell\left(\omega_i\right)$.

We start by importing some Python packages.

```
import numpy as np
from numba import njit, vectorize, prange
import matplotlib.pyplot as plt
%matplotlib inline
from math import gamma
from scipy.stats import beta
```

43.2 Mathematical Expectation of Likelihood Ratio

In *this lecture*, we studied a likelihood ratio $\ell(\omega_t)$

$$\ell(\omega_t) = \frac{f(\omega_t)}{g(\omega_t)}$$

where f and g are densities for Beta distributions with parameters F_a, F_b, G_a, G_b.

Assume that an i.i.d. random variable $\omega_t \in \Omega$ is generated by g.

The **cumulative likelihood ratio** $L(\omega^t)$ is

$$L(\omega^t) = \prod_{i=1}^{t} \ell(\omega_i)$$

Our goal is to approximate the mathematical expectation $E[L(\omega^t)]$ well.

In *this lecture*, we showed that $E[L(\omega^t)]$ equals 1 for all t. We want to check out how well this holds if we replace E by with sample averages from simulations.

This turns out to be easier said than done because for Beta distributions assumed above, $L(\omega^t)$ has a very skewed distribution with a very long tail as $t \to \infty$.

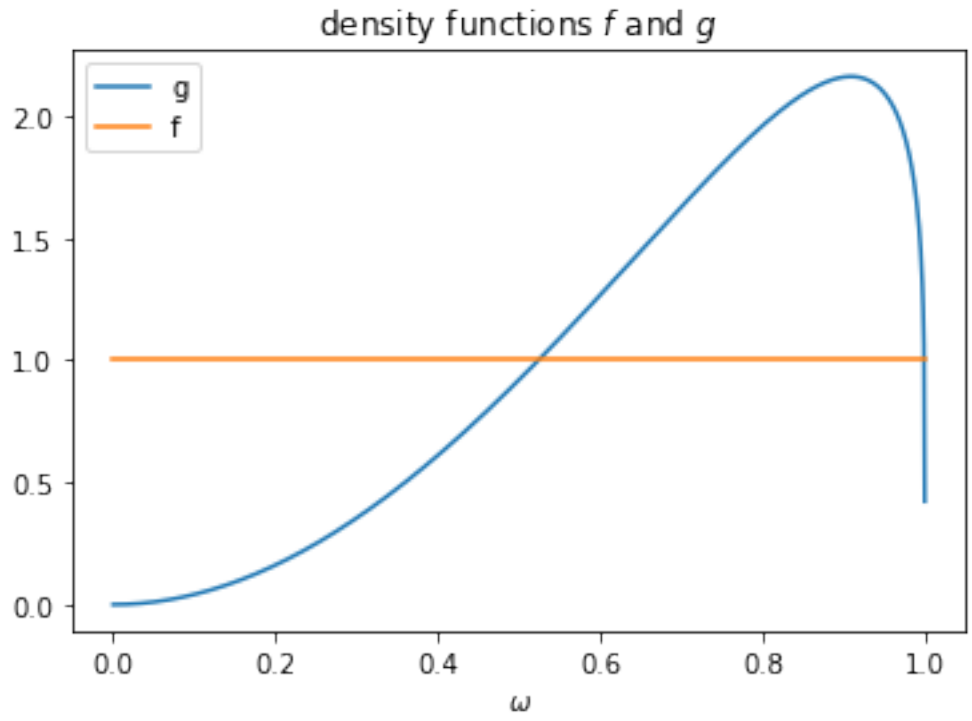
This property makes it difficult efficiently and accurately to estimate the mean by standard Monte Carlo simulation methods.

In this lecture we explore how a standard Monte Carlo method fails and how **importance sampling** provides a more computationally efficient way to approximate the mean of the cumulative likelihood ratio.

We first take a look at the density functions `f` and `g` .

```
# Parameters in the two beta distributions.
F_a, F_b = 1, 1
G_a, G_b = 3, 1.2

@vectorize
def p(w, a, b):
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * w ** (a-1) * (1 - w) ** (b-1)

# The two density functions.
f = njit(lambda w: p(w, F_a, F_b))
g = njit(lambda w: p(w, G_a, G_b))
```

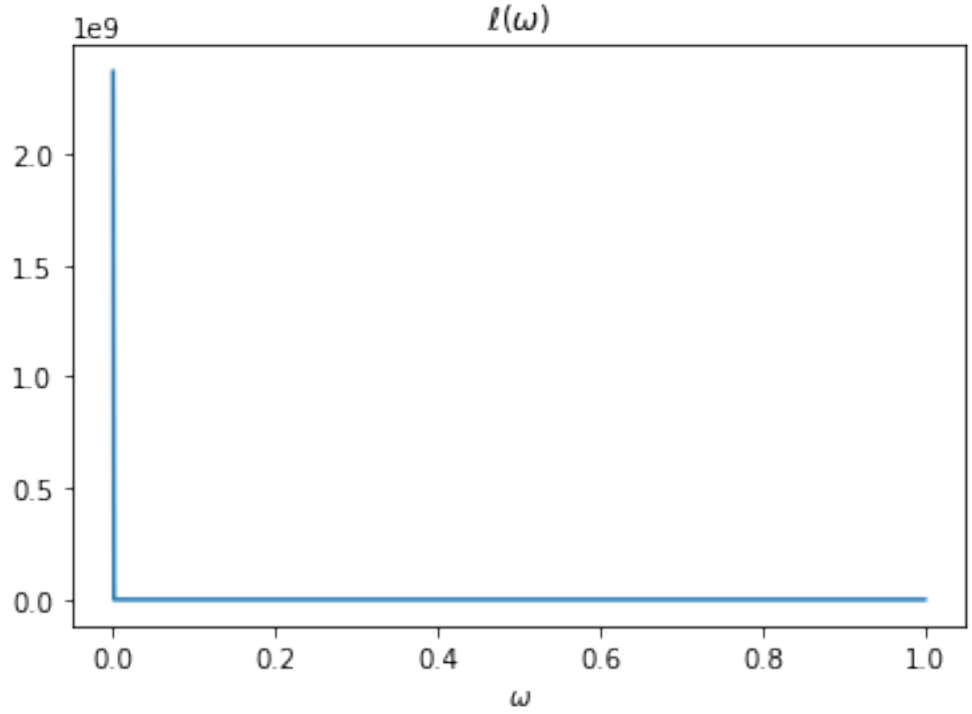
```
w_range = np.linspace(1e-5, 1-1e-5, 1000)

plt.plot(w_range, g(w_range), label='g')
plt.plot(w_range, f(w_range), label='f')
plt.xlabel('$\omega$')
plt.legend()
plt.title('density functions $f$ and $g$')
plt.show()
```



The likelihood ratio is `l(w)=f(w)/g(w)`.

```
l = njit(lambda w: f(w) / g(w))
```

```
plt.plot(w_range, l(w_range))
plt.title('$\ell(\omega)$')
plt.xlabel('$\omega$')
plt.show()
```



The above plots shows that as $\omega \to 0$, $f(\omega)$ is unchanged and $g(\omega) \to 0$, so the likelihood ratio approaches infinity.

A Monte Carlo approximation of $\hat{E}[L(\omega^t)] = \hat{E}\left[\prod_{i=1}^{t} \ell(\omega_i)\right]$ would repeatedly draw ω from g, calculate the likelihood ratio $\ell(\omega) = \frac{f(\omega)}{g(\omega)}$ for each draw, then average these over all draws.

Because $g(\omega) \to 0$ as $\omega \to 0$, such a simulation procedure undersamples a part of the sample space $[0, 1]$ that it is important to visit often in order to do a good job of approximating the mathematical expectation of the likelihood ratio $\ell(\omega)$.

We illustrate this numerically below.

43.3 Importance sampling

We circumvent the issue by using a *change of distribution* called **importance sampling**.

Instead of drawing from g to generate data during the simulation, we use an alternative distribution h to generate draws of ω.

The idea is to design h so that it oversamples the region of Ω where $\ell(\omega_t)$ has large values but low density under g.

After we construct a sample in this way, we must then weight each realization by the likelihood ratio of g and h when we compute the empirical mean of the likelihood ratio.

By doing this, we properly account for the fact that we are using h and not g to simulate data.

To illustrate, suppose were interested in $E[\ell(\omega)]$.

We could simply compute:

$$\hat{E}^g[\ell(\omega)] = \frac{1}{N}\sum_{i=1}^{N} \ell(w_i^g)$$

where ω_i^g indicates that ω_i is drawn from g.

But using our insight from importance sampling, we could instead calculate the object:

$$\hat{E}^h\left[\ell\left(\omega\right)\frac{g(w)}{h(w)}\right] = \frac{1}{N}\sum_{i=1}^{N}\ell(w_i^h)\frac{g(w_i^h)}{h(w_i^h)}$$

where w_i is now drawn from importance distribution h.

Notice that the above two are exactly the same population objects:

$$E^g\left[\ell\left(\omega\right)\right] = \int_\Omega \ell(\omega)g(\omega)d\omega = \int_\Omega \ell(\omega)\frac{g(\omega)}{h(\omega)}h(\omega)d\omega = E^h\left[\ell\left(\omega\right)\frac{g(\omega)}{h(\omega)}\right]$$

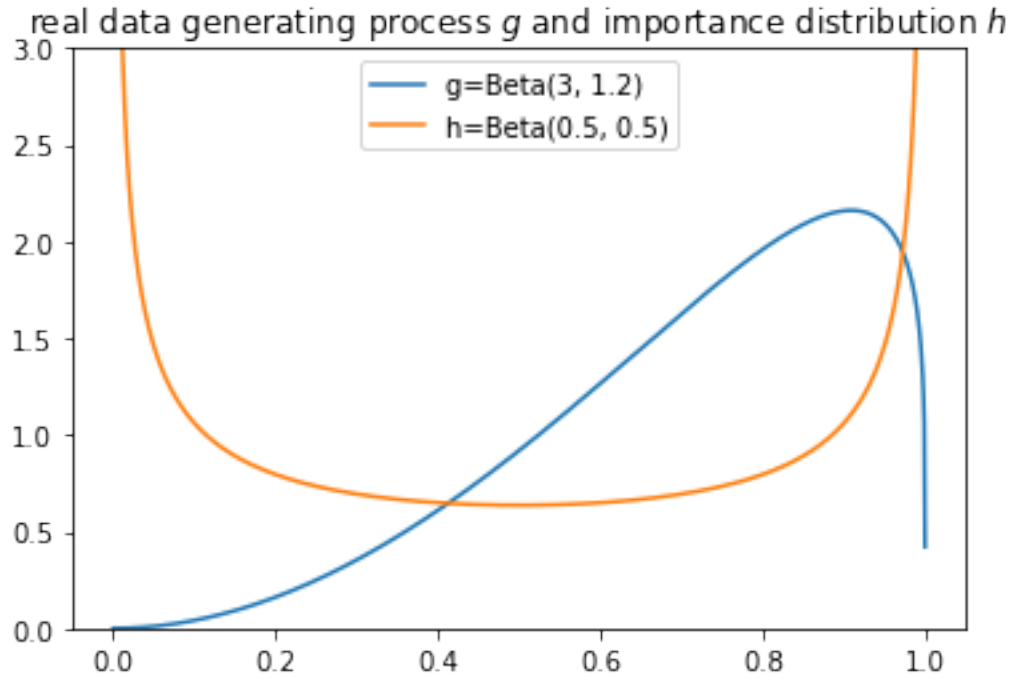
43.4 Selecting a Sampling Distribution

Since we must use an h that has larger mass in parts of the distribution to which g puts low mass, we use $h = Beta(0.5, 0.5)$ as our importance distribution.

The plots compare g and h.

```
g_a, g_b = G_a, G_b
h_a, h_b = 0.5, 0.5
```

```
w_range = np.linspace(1e-5, 1-1e-5, 1000)

plt.plot(w_range, g(w_range), label=f'g=Beta({g_a}, {g_b})')
plt.plot(w_range, p(w_range, 0.5, 0.5), label=f'h=Beta({h_a}, {h_b})')
plt.title('real data generating process $g$ and importance distribution $h$')
plt.legend()
plt.ylim([0., 3.])
plt.show()
```



43.5 Approximating a cumulative likelihood ratio

We now study how to use importance sampling to approximate $E\left[L(\omega^t)\right] = \left[\prod_{i=1}^T \ell\left(\omega_i\right)\right]$.

As above, our plan is to draw sequences ω^t from q and then re-weight the likelihood ratio appropriately:

$$\hat{E}^p\left[L\left(\omega^t\right)\right] = \hat{E}^p\left[\prod_{t=1}^T \ell\left(\omega_t\right)\right] = \hat{E}^q\left[\prod_{t=1}^T \ell\left(\omega_t\right)\frac{p\left(\omega_t\right)}{q\left(\omega_t\right)}\right] = \frac{1}{N}\sum_{i=1}^N\left(\prod_{t=1}^T \ell(\omega_{i,t}^h)\frac{p\left(\omega_{i,t}^h\right)}{q\left(\omega_{i,t}^h\right)}\right)$$

where the last equality uses $\omega_{i,t}^h$ drawn from the importance distribution q.

Here $\frac{p\left(\omega_{i,t}^q\right)}{q\left(\omega_{i,t}^q\right)}$ is the weight we assign to each data point $\omega_{i,t}^q$.

Below we prepare a Python function for computing the importance sampling estimates given any beta distributions p, q.

```
@njit(parallel=True)
def estimate(p_a, p_b, q_a, q_b, T=1, N=10000):

    μ_L = 0
    for i in prange(N):

        L = 1
        weight = 1
        for t in range(T):
            w = np.random.beta(q_a, q_b)
            l = f(w) / g(w)

            L *= l
            weight *= p(w, p_a, p_b) / p(w, q_a, q_b)

        μ_L += L * weight

    μ_L /= N

    return μ_L
```

Consider the case when $T = 1$, which amounts to approximating $E_0\left[\ell\left(\omega\right)\right]$

For the standard Monte Carlo estimate, we can set $p = g$ and $q = g$.

```
estimate(g_a, g_b, g_a, g_b, T=1, N=10000)
```

```
0.9559149232111366
```

For our importance sampling estimate, we set $q = h$.

```
estimate(g_a, g_b, h_a, h_b, T=1, N=10000)
```

```
0.9962071094803098
```

Evidently, even at T=1, our importance sampling estimate is closer to 1 than is the Monte Carlo estimate.

Bigger differences arise when computing expectations over longer sequences, $E_0\left[L\left(\omega^t\right)\right]$.

Setting $T = 10$, we find that the Monte Carlo method severely underestimates the mean while importance sampling still produces an estimate close to its theoretical value of unity.

```
estimate(g_a, g_b, g_a, g_b, T=10, N=10000)
```

```
0.7260288007351615
```

```
estimate(g_a, g_b, h_a, h_b, T=10, N=10000)
```

```
0.9549436706617315
```

43.6 Distribution of Sample Mean

We next study the bias and efficiency of the Monte Carlo and importance sampling approaches.

The code below produces distributions of estimates using both Monte Carlo and importance sampling methods.

```
@njit(parallel=True)
def simulate(p_a, p_b, q_a, q_b, N_simu, T=1):

    μ_L_p = np.empty(N_simu)
    μ_L_q = np.empty(N_simu)

    for i in prange(N_simu):
        μ_L_p[i] = estimate(p_a, p_b, p_a, p_b, T=T)
        μ_L_q[i] = estimate(p_a, p_b, q_a, q_b, T=T)

    return μ_L_p, μ_L_q
```

Again, we first consider estimating $E[\ell(\omega)]$ by setting T=1.

We simulate 1000 times for each method.

```
N_simu = 1000
μ_L_p, μ_L_q = simulate(g_a, g_b, h_a, h_b, N_simu)
```

```
# standard Monte Carlo (mean and std)
np.nanmean(μ_L_p), np.nanvar(μ_L_p)
```

```
(0.9924439178644028, 0.0035506892131557622)
```

```
# importance sampling (mean and std)
np.nanmean(μ_L_q), np.nanvar(μ_L_q)
```

```
(0.9997807159694483, 2.2279259919174323e-05)
```

Although both methods tend to provide a mean estimate of $E[\ell(\omega)]$ close to 1, the importance sampling estimates have smaller variance.

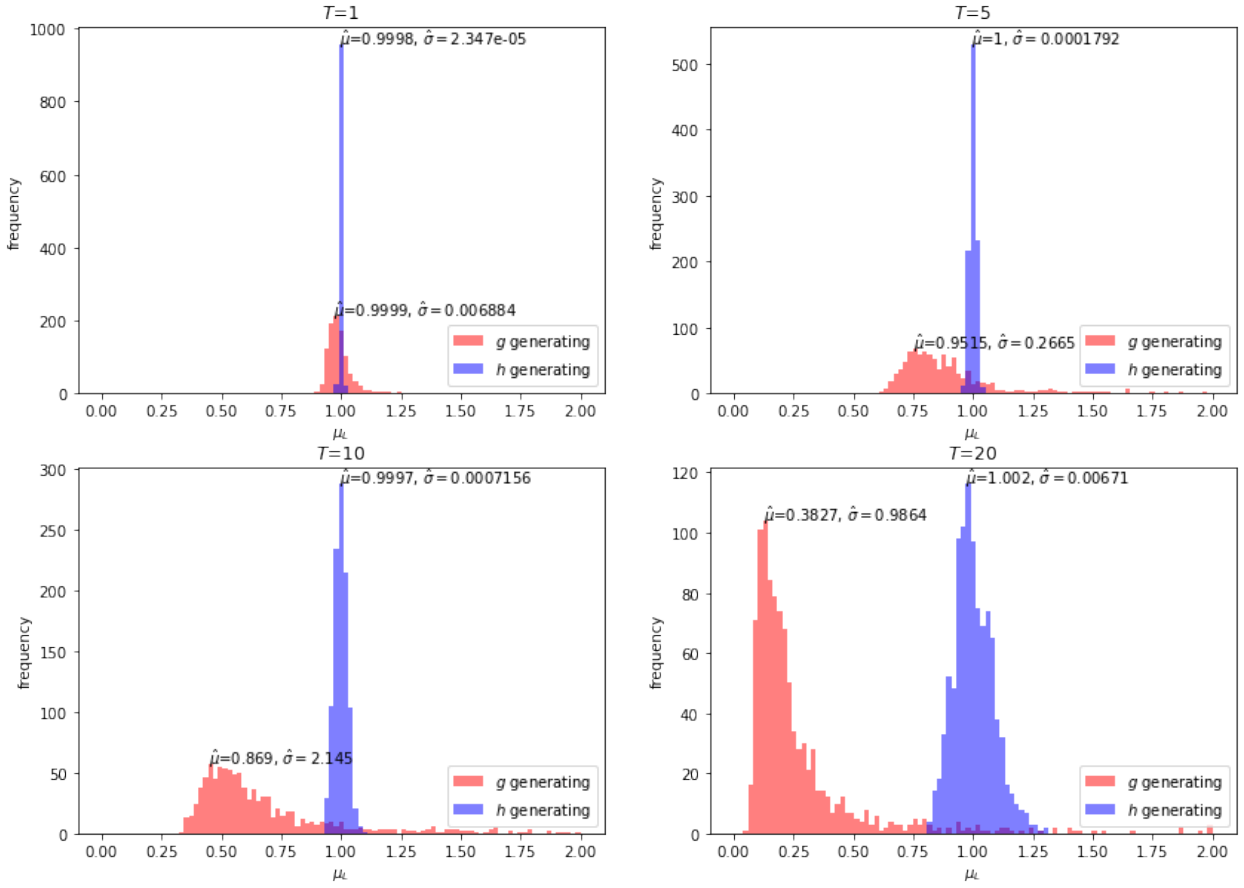
Next, we present distributions of estimates for $\hat{E}[L(\omega^t)]$, in cases for $T = 1, 5, 10, 20$.

```
fig, axs = plt.subplots(2, 2, figsize=(14, 10))

μ_range = np.linspace(0, 2, 100)

for i, t in enumerate([1, 5, 10, 20]):
```

(continues on next page)

(continued from previous page)

```
    row = i // 2
    col = i % 2

    μ_L_p, μ_L_q = simulate(g_a, g_b, h_a, h_b, N_simu, T=t)
    μ_hat_p, μ_hat_q = np.nanmean(μ_L_p), np.nanmean(μ_L_q)
    σ_hat_p, σ_hat_q = np.nanvar(μ_L_p), np.nanvar(μ_L_q)

    axs[row, col].set_xlabel('$μ_L$')
    axs[row, col].set_ylabel('frequency')
    axs[row, col].set_title(f'$T$={t}')
    n_p, bins_p, _ = axs[row, col].hist(μ_L_p, bins=μ_range, color='r', alpha=0.5,␣
↪label='$g$ generating')
    n_q, bins_q, _ = axs[row, col].hist(μ_L_q, bins=μ_range, color='b', alpha=0.5,␣
↪label='$h$ generating')
    axs[row, col].legend(loc=4)

    for n, bins, μ_hat, σ_hat in [[n_p, bins_p, μ_hat_p, σ_hat_p],
                                  [n_q, bins_q, μ_hat_q, σ_hat_q]]:
        idx = np.argmax(n)
        axs[row, col].text(bins[idx], n[idx], '$\hat{μ}$='+f'{μ_hat:.4g}'+', $\hat{σ}=
↪$'+f'{σ_hat:.4g}')

plt.show()
```



The simulation exercises above show that the importance sampling estimates are unbiased under all T while the standard Monte Carlo estimates are biased downwards.

Evidently, the bias increases with increases in T.

43.7 More Thoughts about Choice of Sampling Distribution

Above, we arbitraily chose $h = Beta(0.5, 0.5)$ as the importance distribution.

Is there an optimal importance distribution?

In our particular case, since we know in advance that $E_0 \left[L \left(\omega^t \right) \right] = 1$.

We can use that knowledge to our advantage.

Thus, suppose that we simply use $h = f$.

When estimating the mean of the likelihood ratio (T=1), we get:

$$\hat{E}^f \left[\ell(\omega) \frac{g(\omega)}{f(\omega)} \right] = \hat{E}^f \left[\frac{f(\omega)}{g(\omega)} \frac{g(\omega)}{f(\omega)} \right] = \frac{1}{N} \sum_{i=1}^{N} \ell(w_i^f) \frac{g(w_i^f)}{f(w_i^f)} = 1$$

```
μ_L_p, μ_L_q = simulate(g_a, g_b, F_a, F_b, N_simu)
```

```
# importance sampling (mean and std)
np.nanmean(μ_L_q), np.nanvar(μ_L_q)
```

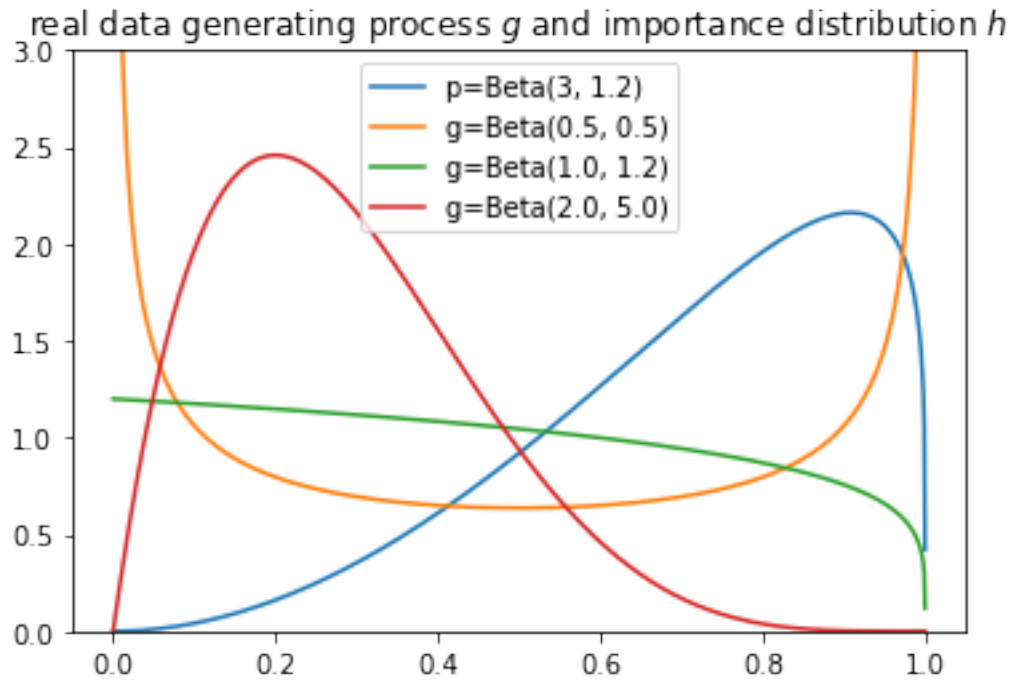
```
(1.0, 0.0)
```

We could also use other distributions as our importance distribution.

Below we choose just a few and compare their sampling properties.

```
a_list = [0.5, 1., 2.]
b_list = [0.5, 1.2, 5.]
```

```
w_range = np.linspace(1e-5, 1-1e-5, 1000)

plt.plot(w_range, g(w_range), label=f'p=Beta({g_a}, {g_b})')
plt.plot(w_range, p(w_range, a_list[0], b_list[0]), label=f'g=Beta({a_list[0]}, {b_
↪list[0]})')
plt.plot(w_range, p(w_range, a_list[1], b_list[1]), label=f'g=Beta({a_list[1]}, {b_
↪list[1]})')
plt.plot(w_range, p(w_range, a_list[2], b_list[2]), label=f'g=Beta({a_list[2]}, {b_
↪list[2]})')
plt.title('real data generating process $g$ and importance distribution $h$')
plt.legend()
plt.ylim([0., 3.])
plt.show()
```



We consider two additonal distributions.

As a reminder h_1 is the original $Beta(0.5, 0.5)$ distribution that we used above.

h_2 is the $Beta(1, 1.2)$ distribution.

Note how h_2 has a similar shape to g at higher values of distribution but more mass at lower values.

Our hunch is that h_2 should be a good importance sampling distribution.

h_3 is the $Beta(2, 5)$ distribution.

Note how h_3 has zero mass at values very close to 0 and at values close to 1.

Our hunch is that h_3 will be a poor importance sampling distribution.

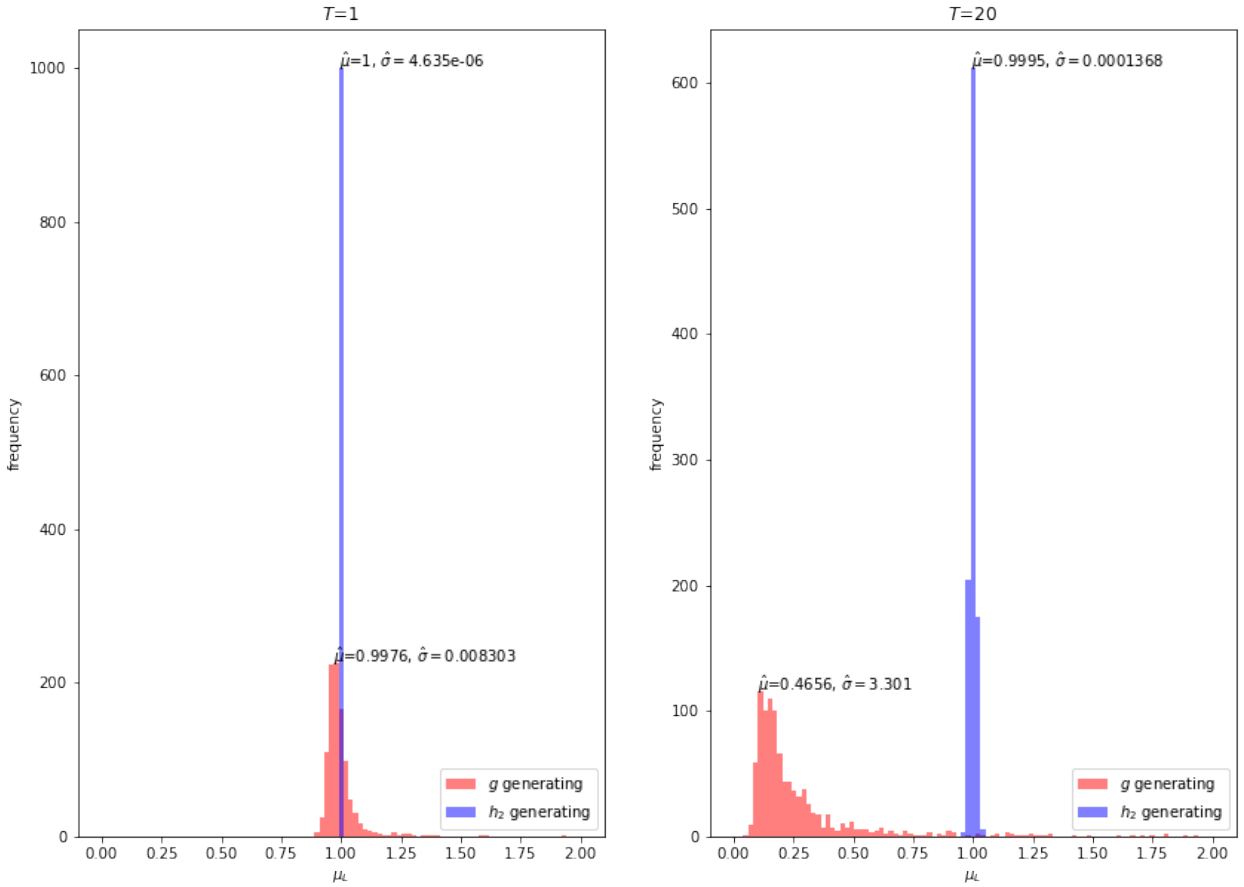
We first simulate a plot the distribution of estimates for $\hat{E}\left[L\left(\omega^t\right)\right]$ using h_2 as the importance sampling distribution.

```
h_a = a_list[1]
h_b = b_list[1]

fig, axs = plt.subplots(1,2, figsize=(14, 10))

μ_range = np.linspace(0, 2, 100)

for i, t in enumerate([1, 20]):


    μ_L_p, μ_L_q = simulate(g_a, g_b, h_a, h_b, N_simu, T=t)
    μ_hat_p, μ_hat_q = np.nanmean(μ_L_p), np.nanmean(μ_L_q)
    σ_hat_p, σ_hat_q = np.nanvar(μ_L_p), np.nanvar(μ_L_q)

    axs[i].set_xlabel('$μ_L$')
    axs[i].set_ylabel('frequency')
    axs[i].set_title(f'$T$={t}')
    n_p, bins_p, _ = axs[i].hist(μ_L_p, bins=μ_range, color='r', alpha=0.5, label='$g
↪$ generating')
```

(continues on next page)

(continued from previous page)

```
    n_q, bins_q, _ = axs[i].hist(μ_L_q, bins=μ_range, color='b', alpha=0.5, label='$h_
↪2$ generating')
    axs[i].legend(loc=4)

    for n, bins, μ_hat, σ_hat in [[n_p, bins_p, μ_hat_p, σ_hat_p],
                                  [n_q, bins_q, μ_hat_q, σ_hat_q]]:
        idx = np.argmax(n)
        axs[i].text(bins[idx], n[idx], '$\hat{μ}$='+f'{μ_hat:.4g}'+', $\hat{σ}=$'+f'
↪{σ_hat:.4g}')

plt.show()
```



Our simulations suggest that indeed h_2 is a quite good importance sampling distribution for our problem.

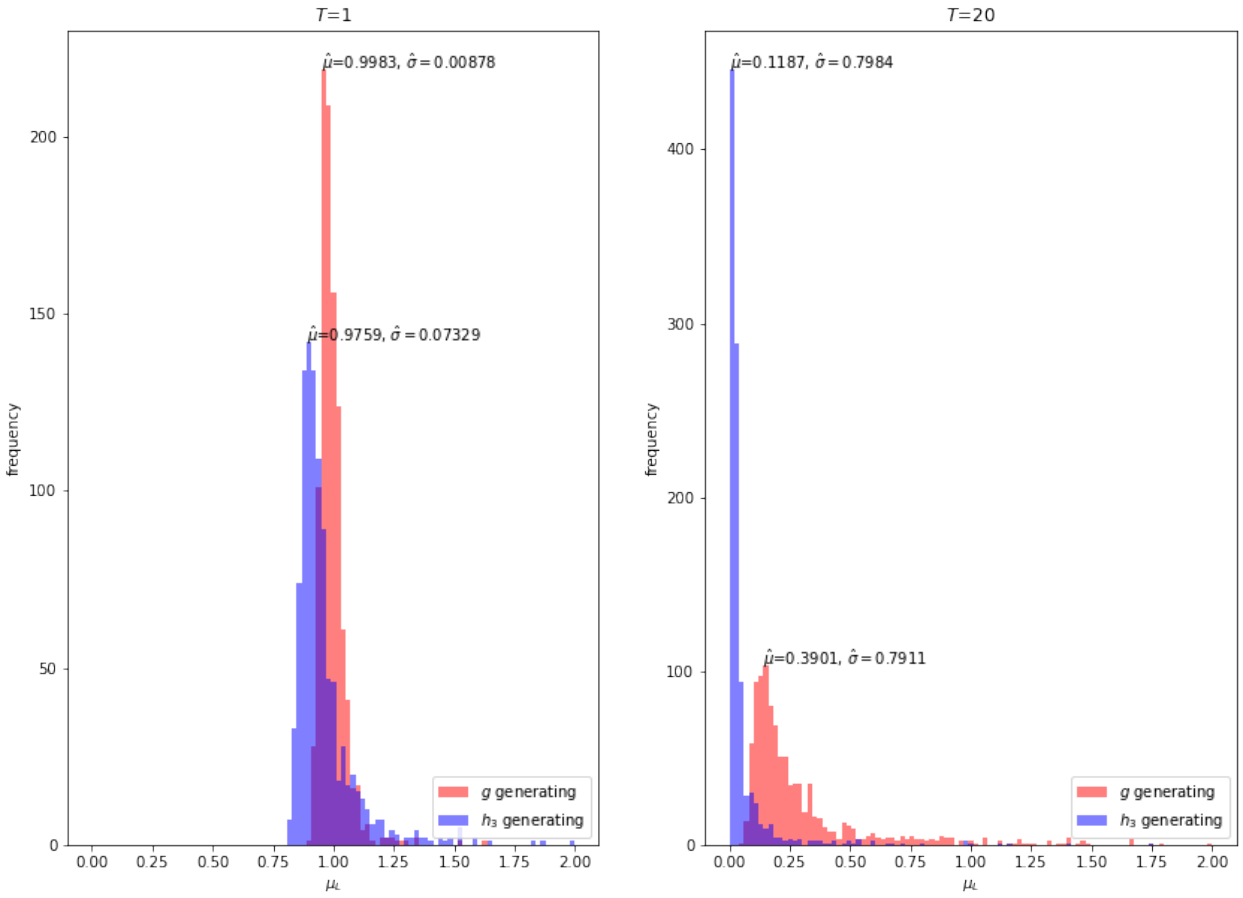
Even at $T = 20$, the mean is very close to 1 and the variance is small.

```
h_a = a_list[2]
h_b = b_list[2]

fig, axs = plt.subplots(1,2, figsize=(14, 10))

μ_range = np.linspace(0, 2, 100)

for i, t in enumerate([1, 20]):
```

(continues on next page)

(continued from previous page)

```
    μ_L_p, μ_L_q = simulate(g_a, g_b, h_a, h_b, N_simu, T=t)
    μ_hat_p, μ_hat_q = np.nanmean(μ_L_p), np.nanmean(μ_L_q)
    σ_hat_p, σ_hat_q = np.nanvar(μ_L_p), np.nanvar(μ_L_q)

    axs[i].set_xlabel('$μ_L$')
    axs[i].set_ylabel('frequency')
    axs[i].set_title(f'$T$={t}')
    n_p, bins_p, _ = axs[i].hist(μ_L_p, bins=μ_range, color='r', alpha=0.5, label='$g
↪$ generating')
    n_q, bins_q, _ = axs[i].hist(μ_L_q, bins=μ_range, color='b', alpha=0.5, label='$h_
↪3$ generating')
    axs[i].legend(loc=4)

    for n, bins, μ_hat, σ_hat in [[n_p, bins_p, μ_hat_p, σ_hat_p],
                                  [n_q, bins_q, μ_hat_q, σ_hat_q]]:
        idx = np.argmax(n)
        axs[i].text(bins[idx], n[idx], '$\hat{μ}$='+f'{μ_hat:.4g}'+', $\hat{σ}$='+f'
↪{σ_hat:.4g}')

plt.show()
```



However, h_3 is evidently a poor importance sampling distribution forpir problem, with a mean estimate far away from 1 for $T = 20$.

Notice that evan at $T = 1$, the mean estimate with importance sampling is more biased than just sampling with g itself.

Thus, our simulations suggest that we would be better off simply using Monte Carlo approximations under g than using h_3 as an importance sampling distribution for our problem.

CHAPTER

FORTYFOUR

A PROBLEM THAT STUMPED MILTON FRIEDMAN

(and that Abraham Wald solved by inventing sequential analysis)

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!conda install -y quantecon
!pip install interpolation
```

44.1 Overview

This lecture describes a statistical decision problem encountered by Milton Friedman and W. Allen Wallis during World War II when they were analysts at the U.S. Government's Statistical Research Group at Columbia University.

This problem led Abraham Wald [Wal47] to formulate **sequential analysis**, an approach to statistical decision problems intimately related to dynamic programming.

In this lecture, we apply dynamic programming algorithms to Friedman and Wallis and Wald's problem.

Key ideas in play will be:

- Bayes' Law
- Dynamic programming
- Type I and type II statistical errors
  - a type I error occurs when you reject a null hypothesis that is true
  - a type II error is when you accept a null hypothesis that is false

- Abraham Wald's **sequential probability ratio test**
- The **power** of a statistical test
- The **critical region** of a statistical test
- A **uniformly most powerful test**

We'll begin with some imports:

```
import numpy as np
import matplotlib.pyplot as plt
from numba import jit, prange, float64, int64
from numba.experimental import jitclass
from interpolation import interp
from math import gamma
```

This lecture uses ideas studied in *this lecture*, *this lecture*. and *this lecture*.

44.2 Origin of the Problem

On pages 137-139 of his 1998 book *Two Lucky People* with Rose Friedman [FF98], Milton Friedman described a problem presented to him and Allen Wallis during World War II, when they worked at the US Government's Statistical Research Group at Columbia University.

Let's listen to Milton Friedman tell us what happened

> In order to understand the story, it is necessary to have an idea of a simple statistical problem, and of the standard procedure for dealing with it. The actual problem out of which sequential analysis grew will serve. The Navy has two alternative designs (say A and B) for a projectile. It wants to determine which is superior. To do so it undertakes a series of paired firings. On each round, it assigns the value 1 or 0 to A accordingly as its performance is superior or inferior to that of B and conversely 0 or 1 to B. The Navy asks the statistician how to conduct the test and how to analyze the results.
>
> The standard statistical answer was to specify a number of firings (say 1,000) and a pair of percentages (e.g., 53% and 47%) and tell the client that if A receives a 1 in more than 53% of the firings, it can be regarded as superior; if it receives a 1 in fewer than 47%, B can be regarded as superior; if the percentage is between 47% and 53%, neither can be so regarded.
>
> When Allen Wallis was discussing such a problem with (Navy) Captain Garret L. Schyler, the captain objected that such a test, to quote from Allen's account, may prove wasteful. If a wise and seasoned ordnance officer like Schyler were on the premises, he would see after the first few thousand or even few hundred [rounds] that the experiment need not be completed either because the new method is obviously inferior or because it is obviously superior beyond what was hoped for ….

Friedman and Wallis struggled with the problem but, after realizing that they were not able to solve it, described the problem to Abraham Wald.

That started Wald on the path that led him to *Sequential Analysis* [Wal47].

We'll formulate the problem using dynamic programming.

44.3 A Dynamic Programming Approach

The following presentation of the problem closely follows Dmitri Berskekas's treatment in **Dynamic Programming and Stochastic Control** [Ber75].

A decision-maker observes a sequence of draws of a random variable z.

He (or she) wants to know which of two probability distributions f_0 or f_1 governs z.

Conditional on knowing that successive observations are drawn from distribution f_0, the sequence of random variables is independently and identically distributed (IID).

Conditional on knowing that successive observations are drawn from distribution f_1, the sequence of random variables is also independently and identically distributed (IID).

But the observer does not know which of the two distributions generated the sequence.

For reasons explained in Exchangeability and Bayesian Updating, this means that the sequence is not IID and that the observer has something to learn, even though he knows both f_0 and f_1.

The decision maker chooses a number of draws (i.e., random samples from the unknown distribution) and uses them to decide which of the two distributions is generating outcomes.

He starts with prior

$$\pi_{-1} = \mathbb{P}\{f = f_0 \mid \text{ no observations}\} \in (0, 1)$$

After observing $k+1$ observations $z_k, z_{k-1}, \ldots, z_0$, he updates this value to

$$\pi_k = \mathbb{P}\{f = f_0 \mid z_k, z_{k-1}, \ldots, z_0\}$$

which is calculated recursively by applying Bayes' law:

$$\pi_{k+1} = \frac{\pi_k f_0(z_{k+1})}{\pi_k f_0(z_{k+1}) + (1-\pi_k) f_1(z_{k+1})}, \quad k = -1, 0, 1, \ldots$$

After observing $z_k, z_{k-1}, \ldots, z_0$, the decision-maker believes that z_{k+1} has probability distribution

$$f_{\pi_k}(v) = \pi_k f_0(v) + (1-\pi_k) f_1(v),$$

which is a mixture of distributions f_0 and f_1, with the weight on f_0 being the posterior probability that $f = f_0$[1].

To illustrate such a distribution, let's inspect some mixtures of beta distributions.

The density of a beta probability distribution with parameters a and b is

$$f(z; a, b) = \frac{\Gamma(a+b) z^{a-1} (1-z)^{b-1}}{\Gamma(a)\Gamma(b)} \quad \text{where} \quad \Gamma(t) := \int_0^\infty x^{t-1} e^{-x} dx$$

The next figure shows two beta distributions in the top panel.

The bottom panel presents mixtures of these distributions, with various mixing probabilities π_k

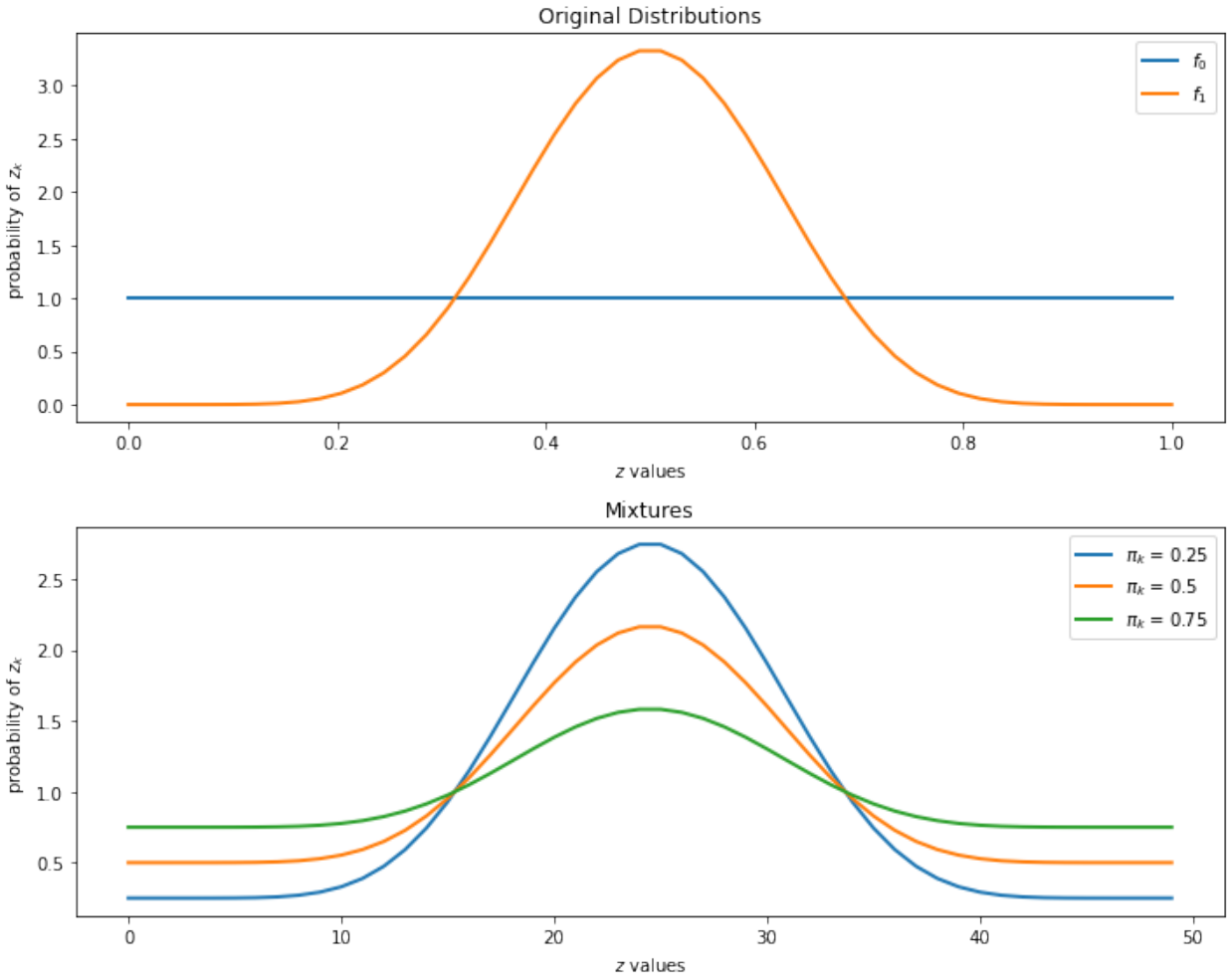
```
@jit(nopython=True)
def p(x, a, b):
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * x**(a-1) * (1 - x)**(b-1)
```

(continues on next page)

[1] The decision maker acts as if he believes that the sequence of random variables $[z_0, z_1, \ldots]$ is *exchangeable*. See Exchangeability and Bayesian Updating and [Kre88] chapter 11, for discussions of exchangeability.

(continued from previous page)

```
f0 = lambda x: p(x, 1, 1)
f1 = lambda x: p(x, 9, 9)
grid = np.linspace(0, 1, 50)

fig, axes = plt.subplots(2, figsize=(10, 8))

axes[0].set_title("Original Distributions")
axes[0].plot(grid, f0(grid), lw=2, label="$f_0$")
axes[0].plot(grid, f1(grid), lw=2, label="$f_1$")

axes[1].set_title("Mixtures")
for π in 0.25, 0.5, 0.75:
    y = π * f0(grid) + (1 - π) * f1(grid)
    axes[1].plot(y, lw=2, label=f"$\pi_k$ = {π}")

for ax in axes:
    ax.legend()
    ax.set(xlabel="$z$ values", ylabel="probability of $z_k$")

plt.tight_layout()
plt.show()
```



44.3.1 Losses and Costs

After observing $z_k, z_{k-1}, \ldots, z_0$, the decision-maker chooses among three distinct actions:

- He decides that $f = f_0$ and draws no more z's
- He decides that $f = f_1$ and draws no more z's
- He postpones deciding now and instead chooses to draw a z_{k+1}

Associated with these three actions, the decision-maker can suffer three kinds of losses:

- A loss L_0 if he decides $f = f_0$ when actually $f = f_1$
- A loss L_1 if he decides $f = f_1$ when actually $f = f_0$
- A cost c if he postpones deciding and chooses instead to draw another z

44.3.2 Digression on Type I and Type II Errors

If we regard $f = f_0$ as a null hypothesis and $f = f_1$ as an alternative hypothesis, then L_1 and L_0 are losses associated with two types of statistical errors

- a type I error is an incorrect rejection of a true null hypothesis (a "false positive")
- a type II error is a failure to reject a false null hypothesis (a "false negative")

So when we treat $f = f_0$ as the null hypothesis

- We can think of L_1 as the loss associated with a type I error.
- We can think of L_0 as the loss associated with a type II error.

44.3.3 Intuition

Let's try to guess what an optimal decision rule might look like before we go further.

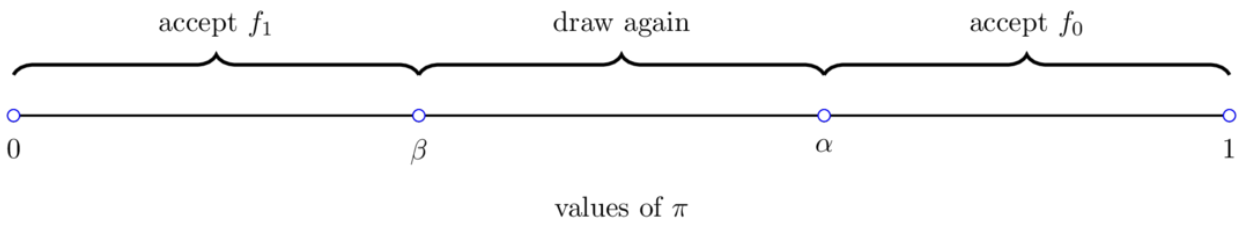
Suppose at some given point in time that π is close to 1.

Then our prior beliefs and the evidence so far point strongly to $f = f_0$.

If, on the other hand, π is close to 0, then $f = f_1$ is strongly favored.

Finally, if π is in the middle of the interval $[0, 1]$, then we have little information in either direction.

This reasoning suggests a decision rule such as the one shown in the figure



As we'll see, this is indeed the correct form of the decision rule.

The key problem is to determine the threshold values α, β, which will depend on the parameters listed above.

You might like to pause at this point and try to predict the impact of a parameter such as c or L_0 on α or β.

44.3.4 A Bellman Equation

Let $J(\pi)$ be the total loss for a decision-maker with current belief π who chooses optimally.

With some thought, you will agree that J should satisfy the Bellman equation

$$J(\pi) = \min\{(1-\pi)L_0,\ \pi L_1,\ c + \mathbb{E}[J(\pi')]\} \tag{1}$$

where π' is the random variable defined by Bayes' Law

$$\pi' = \kappa(z', \pi) = \frac{\pi f_0(z')}{\pi f_0(z') + (1-\pi) f_1(z')}$$

when π is fixed and z' is drawn from the current best guess, which is the distribution f defined by

$$f_\pi(v) = \pi f_0(v) + (1-\pi) f_1(v)$$

In the Bellman equation, minimization is over three actions:

1. Accept the hypothesis that $f = f_0$
2. Accept the hypothesis that $f = f_1$
3. Postpone deciding and draw again

We can represent the Bellman equation as

$$J(\pi) = \min\{(1-\pi)L_0,\ \pi L_1,\ h(\pi)\} \tag{2}$$

where $\pi \in [0, 1]$ and

- $(1-\pi)L_0$ is the expected loss associated with accepting f_0 (i.e., the cost of making a type II error).
- πL_1 is the expected loss associated with accepting f_1 (i.e., the cost of making a type I error).
- $h(\pi) := c + \mathbb{E}[J(\pi')]$ the continuation value; i.e., the expected cost associated with drawing one more z.

The optimal decision rule is characterized by two numbers $\alpha, \beta \in (0,1) \times (0,1)$ that satisfy

$$(1-\pi)L_0 < \min\{\pi L_1, c + \mathbb{E}[J(\pi')]\} \text{ if } \pi \geq \alpha$$

and

$$\pi L_1 < \min\{(1-\pi)L_0, c + \mathbb{E}[J(\pi')]\} \text{ if } \pi \leq \beta$$

The optimal decision rule is then

$$\begin{aligned} \text{accept } f = f_0 \text{ if } \pi \geq \alpha \\ \text{accept } f = f_1 \text{ if } \pi \leq \beta \\ \text{draw another } z \text{ if } \beta \leq \pi \leq \alpha \end{aligned}$$

Our aim is to compute the value function J, and from it the associated cutoffs α and β.

To make our computations simpler, using (2), we can write the continuation value $h(\pi)$ as

$$\begin{aligned} h(\pi) &= c + \mathbb{E}[J(\pi')] \\ &= c + \mathbb{E}_{\pi'} \min\{(1-\pi')L_0, \pi' L_1, h(\pi')\} \\ &= c + \int \min\{(1-\kappa(z',\pi))L_0, \kappa(z',\pi)L_1, h(\kappa(z',\pi))\} f_\pi(z') dz' \end{aligned} \tag{3}$$

The equality

$$h(\pi) = c + \int \min\{(1 - \kappa(z', \pi))L_0, \kappa(z', \pi)L_1, h(\kappa(z', \pi))\} f_\pi(z') dz' \tag{4}$$

can be understood as a functional equation, where h is the unknown.

Using the functional equation, (4), for the continuation value, we can back out optimal choices using the right side of (2).

This functional equation can be solved by taking an initial guess and iterating to find a fixed point.

Thus, we iterate with an operator Q, where

$$Qh(\pi) = c + \int \min\{(1 - \kappa(z', \pi))L_0, \kappa(z', \pi)L_1, h(\kappa(z', \pi))\} f_\pi(z') dz'$$

44.4 Implementation

First, we will construct a `jitclass` to store the parameters of the model

```
wf_data = [('a0', float64),          # Parameters of beta distributions
           ('b0', float64),
           ('a1', float64),
           ('b1', float64),
           ('c', float64),           # Cost of another draw
           ('π_grid_size', int64),
           ('L0', float64),          # Cost of selecting f0 when f1 is true
           ('L1', float64),          # Cost of selecting f1 when f0 is true
           ('π_grid', float64[:]),
           ('mc_size', int64),
           ('z0', float64[:]),
           ('z1', float64[:])]
```

```
@jitclass(wf_data)
class WaldFriedman:

    def __init__(self,
                 c=1.25,
                 a0=1,
                 b0=1,
                 a1=3,
                 b1=1.2,
                 L0=25,
                 L1=25,
                 π_grid_size=200,
                 mc_size=1000):

        self.a0, self.b0 = a0, b0
        self.a1, self.b1 = a1, b1
        self.c, self.π_grid_size = c, π_grid_size
        self.L0, self.L1 = L0, L1
        self.π_grid = np.linspace(0, 1, π_grid_size)
        self.mc_size = mc_size

        self.z0 = np.random.beta(a0, b0, mc_size)
        self.z1 = np.random.beta(a1, b1, mc_size)
```

(continues on next page)

(continued from previous page)

```
    def f0(self, x):

        return p(x, self.a0, self.b0)

    def f1(self, x):

        return p(x, self.a1, self.b1)

    def f0_rvs(self):
        return np.random.beta(self.a0, self.b0)

    def f1_rvs(self):
        return np.random.beta(self.a1, self.b1)

    def κ(self, z, π):
        """
        Updates π using Bayes' rule and the current observation z
        """

        f0, f1 = self.f0, self.f1

        π_f0, π_f1 = π * f0(z), (1 - π) * f1(z)
        π_new = π_f0 / (π_f0 + π_f1)

        return π_new
```

As in the *optimal growth lecture*, to approximate a continuous value function

- We iterate at a finite grid of possible values of π.
- When we evaluate $\mathbb{E}[J(\pi')]$ between grid points, we use linear interpolation.

We define the operator function `Q` below.

```
@jit(nopython=True, parallel=True)
def Q(h, wf):

    c, π_grid = wf.c, wf.π_grid
    L0, L1 = wf.L0, wf.L1
    z0, z1 = wf.z0, wf.z1
    mc_size = wf.mc_size

    κ = wf.κ

    h_new = np.empty_like(π_grid)
    h_func = lambda p: interp(π_grid, h, p)

    for i in prange(len(π_grid)):
        π = π_grid[i]

        # Find the expected value of J by integrating over z
        integral_f0, integral_f1 = 0, 0
        for m in range(mc_size):
            π_0 = κ(z0[m], π)  # Draw z from f0 and update π
            integral_f0 += min((1 - π_0) * L0, π_0 * L1, h_func(π_0))

            π_1 = κ(z1[m], π)  # Draw z from f1 and update π
            integral_f1 += min((1 - π_1) * L0, π_1 * L1, h_func(π_1))
```

(continues on next page)

(continued from previous page)

```
        integral = (π * integral_f0 + (1 - π) * integral_f1) / mc_size

        h_new[i] = c + integral

    return h_new
```

To solve the model, we will iterate using `Q` to find the fixed point

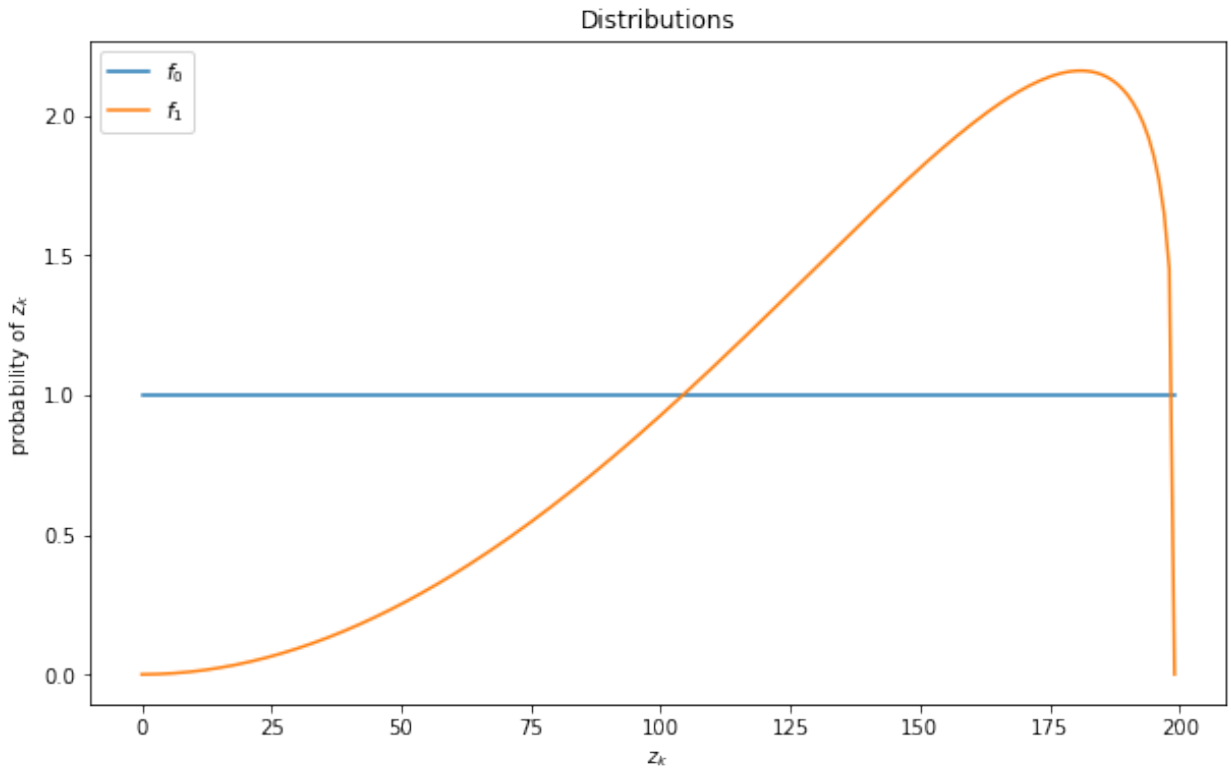
```
@jit(nopython=True)
def solve_model(wf, tol=1e-4, max_iter=1000):
    """
    Compute the continuation value function

    * wf is an instance of WaldFriedman
    """

    # Set up loop
    h = np.zeros(len(wf.π_grid))
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        h_new = Q(h, wf)
        error = np.max(np.abs(h - h_new))
        i += 1
        h = h_new

    if i == max_iter:
        print("Failed to converge!")

    return h_new
```

44.5 Analysis

Let's inspect outcomes.

We will be using the default parameterization with distributions like so

```
wf = WaldFriedman()

fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(wf.f0(wf.π_grid), label="$f_0$")
ax.plot(wf.f1(wf.π_grid), label="$f_1$")
ax.set(ylabel="probability of $z_k$", xlabel="$z_k$", title="Distributions")
ax.legend()

plt.show()
```



44.5.1 Value Function

To solve the model, we will call our `solve_model` function

```
h_star = solve_model(wf)     # Solve the model
```

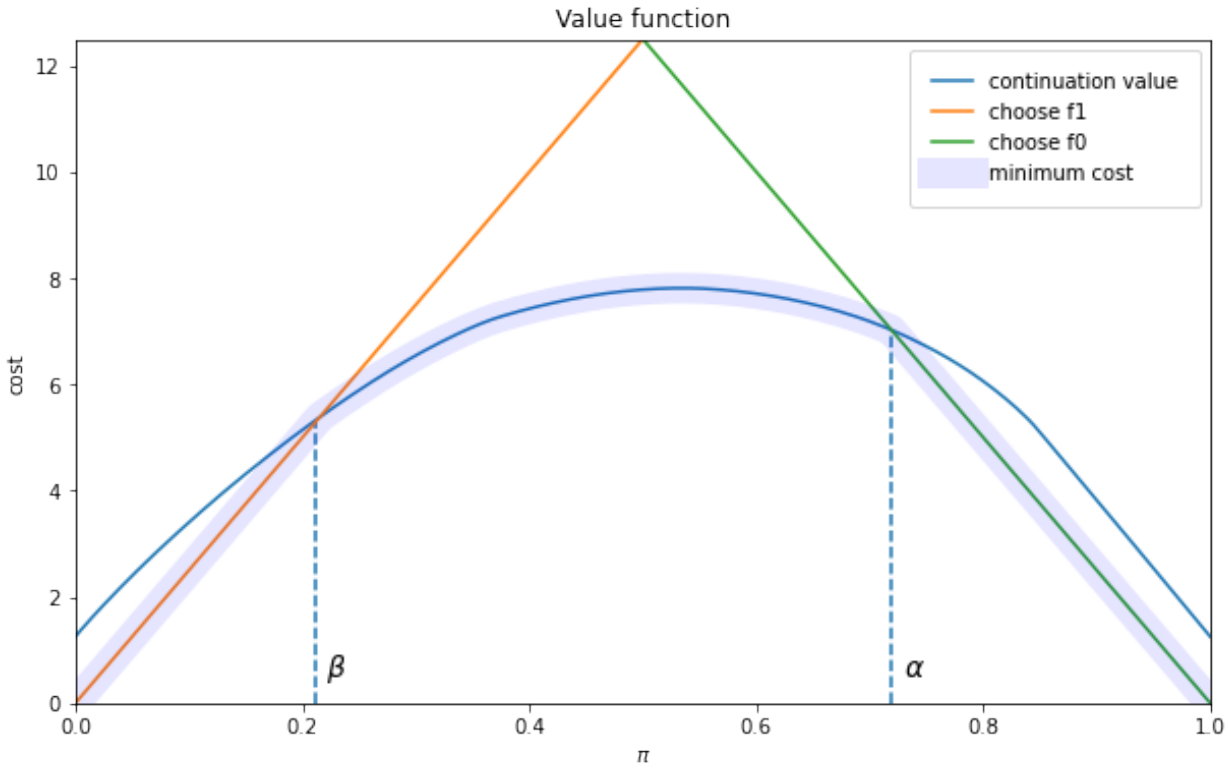
We will also set up a function to compute the cutoffs α and β and plot these on our value function plot

```
@jit(nopython=True)
def find_cutoff_rule(wf, h):

    """
    This function takes a continuation value function and returns the
    corresponding cutoffs of where you transition between continuing and
    choosing a specific model
    """

    π_grid = wf.π_grid
    L0, L1 = wf.L0, wf.L1

    # Evaluate cost at all points on grid for choosing a model
    payoff_f0 = (1 - π_grid) * L0
    payoff_f1 = π_grid * L1

    # The cutoff points can be found by differencing these costs with
    # The Bellman equation (J is always less than or equal to p_c_i)
    β = π_grid[np.searchsorted(
                               payoff_f1 - np.minimum(h, payoff_f0),
```

(continues on next page)

(continued from previous page)

```
                                1e-10)
                - 1]
    α = π_grid[np.searchsorted(
                                np.minimum(h, payoff_f1) - payoff_f0,
                                1e-10)
                - 1]

    return (β, α)

β, α = find_cutoff_rule(wf, h_star)
cost_L0 = (1 - wf.π_grid) * wf.L0
cost_L1 = wf.π_grid * wf.L1

fig, ax = plt.subplots(figsize=(10, 6))

ax.plot(wf.π_grid, h_star, label='continuation value')
ax.plot(wf.π_grid, cost_L1, label='choose f1')
ax.plot(wf.π_grid, cost_L0, label='choose f0')
ax.plot(wf.π_grid,
        np.amin(np.column_stack([h_star, cost_L0, cost_L1]),axis=1),
        lw=15, alpha=0.1, color='b', label='minimum cost')

ax.annotate(r"$\beta$", xy=(β + 0.01, 0.5), fontsize=14)
ax.annotate(r"$\alpha$", xy=(α + 0.01, 0.5), fontsize=14)

plt.vlines(β, 0, β * wf.L0, linestyle="--")
plt.vlines(α, 0, (1 - α) * wf.L1, linestyle="--")

ax.set(xlim=(0, 1), ylim=(0, 0.5 * max(wf.L0, wf.L1)), ylabel="cost",
       xlabel="$\pi$", title="Value function")

plt.legend(borderpad=1.1)
plt.show()
```



The value function equals πL_1 for $\pi \leq \beta$, and $(1-\pi)L_0$ for $\pi \geq \alpha$.

The slopes of the two linear pieces of the value function are determined by L_1 and $-L_0$.

The value function is smooth in the interior region, where the posterior probability assigned to f_0 is in the indecisive region $\pi \in (\beta, \alpha)$.

The decision-maker continues to sample until the probability that he attaches to model f_0 falls below β or above α.

44.5.2 Simulations

The next figure shows the outcomes of 500 simulations of the decision process.

On the left is a histogram of the stopping times, which equal the number of draws of z_k required to make a decision.

The average number of draws is around 6.6.

On the right is the fraction of correct decisions at the stopping time.

In this case, the decision-maker is correct 80% of the time

```
def simulate(wf, true_dist, h_star, π_0=0.5):

    """
    This function takes an initial condition and simulates until it
    stops (when a decision is made)
    """

    f0, f1 = wf.f0, wf.f1
    f0_rvs, f1_rvs = wf.f0_rvs, wf.f1_rvs
    π_grid = wf.π_grid
```

(continues on next page)

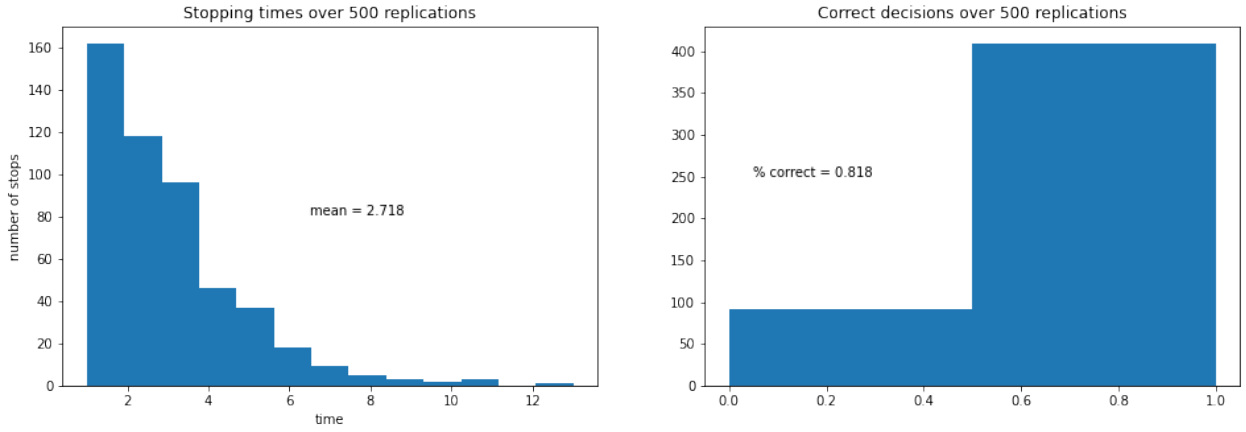
(continued from previous page)

```
    κ = wf.κ

    if true_dist == "f0":
        f, f_rvs = wf.f0, wf.f0_rvs
    elif true_dist == "f1":
        f, f_rvs = wf.f1, wf.f1_rvs

    # Find cutoffs
    β, α = find_cutoff_rule(wf, h_star)

    # Initialize a couple of useful variables
    decision_made = False
    π = π_0
    t = 0

    while decision_made is False:
        # Maybe should specify which distribution is correct one so that
        # the draws come from the "right" distribution
        z = f_rvs()
        t = t + 1
        π = κ(z, π)
        if π < β:
            decision_made = True
            decision = 1
        elif π > α:
            decision_made = True
            decision = 0

    if true_dist == "f0":
        if decision == 0:
            correct = True
        else:
            correct = False

    elif true_dist == "f1":
        if decision == 1:
            correct = True
        else:
            correct = False

    return correct, π, t

def stopping_dist(wf, h_star, ndraws=250, true_dist="f0"):

    """
    Simulates repeatedly to get distributions of time needed to make a
    decision and how often they are correct
    """

    tdist = np.empty(ndraws, int)
    cdist = np.empty(ndraws, bool)

    for i in range(ndraws):
        correct, π, t = simulate(wf, true_dist, h_star)
        tdist[i] = t
        cdist[i] = correct
```

(continues on next page)

(continued from previous page)

```
    return cdist, tdist

def simulation_plot(wf):
    h_star = solve_model(wf)
    ndraws = 500
    cdist, tdist = stopping_dist(wf, h_star, ndraws)

    fig, ax = plt.subplots(1, 2, figsize=(16, 5))

    ax[0].hist(tdist, bins=np.max(tdist))
    ax[0].set_title(f"Stopping times over {ndraws} replications")
    ax[0].set(xlabel="time", ylabel="number of stops")
    ax[0].annotate(f"mean = {np.mean(tdist)}", xy=(max(tdist) / 2,
                   max(np.histogram(tdist, bins=max(tdist))[0]) / 2))

    ax[1].hist(cdist.astype(int), bins=2)
    ax[1].set_title(f"Correct decisions over {ndraws} replications")
    ax[1].annotate(f"% correct = {np.mean(cdist)}",
                   xy=(0.05, ndraws / 2))

    plt.show()

simulation_plot(wf)
```



44.5.3 Comparative Statics

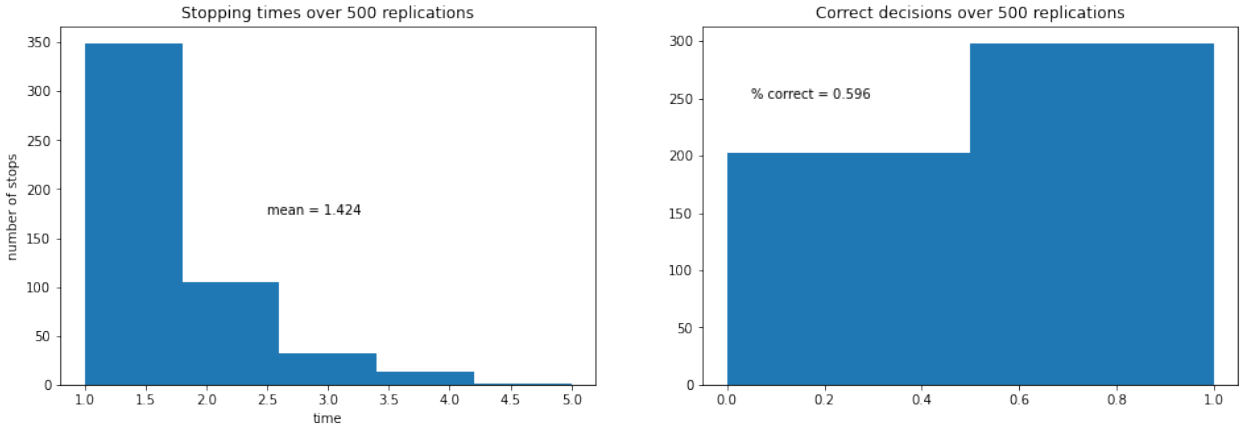
Now let's consider the following exercise.

We double the cost of drawing an additional observation.

Before you look, think about what will happen:

- Will the decision-maker be correct more or less often?
- Will he make decisions sooner or later?

```
wf = WaldFriedman(c=2.5)
simulation_plot(wf)
```



Increased cost per draw has induced the decision-maker to take fewer draws before deciding.

Because he decides with fewer draws, the percentage of time he is correct drops.

This leads to him having a higher expected loss when he puts equal weight on both models.

44.5.4 A Notebook Implementation

To facilitate comparative statics, we provide a Jupyter notebook that generates the same plots, but with sliders.

With these sliders, you can adjust parameters and immediately observe

- effects on the smoothness of the value function in the indecisive middle range as we increase the number of grid points in the piecewise linear approximation.
- effects of different settings for the cost parameters L_0, L_1, c, the parameters of two beta distributions f_0 and f_1, and the number of points and linear functions m to use in the piece-wise continuous approximation to the value function.
- various simulations from f_0 and associated distributions of waiting times to making a decision.
- associated histograms of correct and incorrect decisions.

44.6 Comparison with Neyman-Pearson Formulation

For several reasons, it is useful to describe the theory underlying the test that Navy Captain G. S. Schuyler had been told to use and that led him to approach Milton Friedman and Allan Wallis to convey his conjecture that superior practical procedures existed.

Evidently, the Navy had told Captail Schuyler to use what it knew to be a state-of-the-art Neyman-Pearson test.

We'll rely on Abraham Wald's [Wal47] elegant summary of Neyman-Pearson theory.

For our purposes, watch for there features of the setup:

- the assumption of a *fixed* sample size n
- the application of laws of large numbers, conditioned on alternative probability models, to interpret the probabilities α and β defined in the Neyman-Pearson theory

Recall that in the sequential analytic formulation above, that

- The sample size n is not fixed but rather an object to be chosen; technically n is a random variable.

- The parameters β and α characterize cut-off rules used to determine n as a random variable.
- Laws of large numbers make no appearances in the sequential construction.

In chapter 1 of **Sequential Analysis** [Wal47] Abraham Wald summarizes the Neyman-Pearson approach to hypothesis testing.

Wald frames the problem as making a decision about a probability distribution that is partially known.

(You have to assume that *something* is already known in order to state a well-posed problem – usually, *something* means *a lot*)

By limiting what is unknown, Wald uses the following simple structure to illustrate the main ideas:

- A decision-maker wants to decide which of two distributions f_0, f_1 govern an IID random variable z.
- The null hypothesis H_0 is the statement that f_0 governs the data.
- The alternative hypothesis H_1 is the statement that f_1 governs the data.
- The problem is to devise and analyze a test of hypothesis H_0 against the alternative hypothesis H_1 on the basis of a sample of a fixed number n independent observations $z_1, z_2, \ldots, z_n$ of the random variable z.

To quote Abraham Wald,

> A test procedure leading to the acceptance or rejection of the [null] hypothesis in question is simply a rule specifying, for each possible sample of size n, whether the [null] hypothesis should be accepted or rejected on the basis of the sample. This may also be expressed as follows: A test procedure is simply a subdivision of the totality of all possible samples of size n into two mutually exclusive parts, say part 1 and part 2, together with the application of the rule that the [null] hypothesis be accepted if the observed sample is contained in part 2. Part 1 is also called the critical region. Since part 2 is the totality of all samples of size n which are not included in part 1, part 2 is uniquely determined by part 1. Thus, choosing a test procedure is equivalent to determining a critical region.

Let's listen to Wald longer:

> As a basis for choosing among critical regions the following considerations have been advanced by Neyman and Pearson: In accepting or rejecting H_0 we may commit errors of two kinds. We commit an error of the first kind if we reject H_0 when it is true; we commit an error of the second kind if we accept H_0 when H_1 is true. After a particular critical region W has been chosen, the probability of committing an error of the first kind, as well as the probability of committing an error of the second kind is uniquely determined. The probability of committing an error of the first kind is equal to the probability, determined by the assumption that H_0 is true, that the observed sample will be included in the critical region W. The probability of committing an error of the second kind is equal to the probability, determined on the assumption that H_1 is true, that the probability will fall outside the critical region W. For any given critical region W we shall denote the probability of an error of the first kind by α and the probability of an error of the second kind by β.

Let's listen carefully to how Wald applies law of large numbers to interpret α and β:

> The probabilities α and β have the following important practical interpretation: Suppose that we draw a large number of samples of size n. Let M be the number of such samples drawn. Suppose that for each of these M samples we reject H_0 if the sample is included in W and accept H_0 if the sample lies outside W. In this way we make M statements of rejection or acceptance. Some of these statements will in general be wrong. If H_0 is true and if M is large, the probability is nearly 1 (i.e., it is practically certain) that the proportion of wrong statements (i.e., the number of wrong statements divided by M) will be approximately α. If H_1 is true, the probability is nearly 1 that the proportion of wrong statements will be approximately β. Thus, we can say that in the long run [here Wald applies law of large numbers by driving $M \to \infty$ (our comment, not Wald's)] the proportion of wrong statements will be α if $H_0$is true and β if H_1 is true.

The quantity α is called the *size* of the critical region, and the quantity $1 - \beta$ is called the *power* of the critical region.

Wald notes that

> one critical region W is more desirable than another if it has smaller values of α and β. Although either α or β can be made arbitrarily small by a proper choice of the critical region W, it is possible to make both α and β arbitrarily small for a fixed value of n, i.e., a fixed sample size.

Wald summarizes Neyman and Pearson's setup as follows:

> Neyman and Pearson show that a region consisting of all samples $(z_1, z_2, \ldots, z_n)$ which satisfy the inequality
>
> $$\frac{f_1(z_1) \cdots f_1(z_n)}{f_0(z_1) \cdots f_0(z_n)} \geq k$$
>
> is a most powerful critical region for testing the hypothesis H_0 against the alternative hypothesis H_1. The term k on the right side is a constant chosen so that the region will have the required size α.

Wald goes on to discuss Neyman and Pearson's concept of *uniformly most powerful* test.

Here is how Wald introduces the notion of a sequential test

> A rule is given for making one of the following three decisions at any stage of the experiment (at the m th trial for each integral value of m): (1) to accept the hypothesis H , (2) to reject the hypothesis H , (3) to continue the experiment by making an additional observation. Thus, such a test procedure is carried out sequentially. On the basis of the first observation, one of the aforementioned decision is made. If the first or second decision is made, the process is terminated. If the third decision is made, a second trial is performed. Again, on the basis of the first two observations, one of the three decision is made. If the third decision is made, a third trial is performed, and so on. The process is continued until either the first or the second decisions is made. The number n of observations required by such a test procedure is a random variable, since the value of n depends on the outcome of the observations.

44.7 Sequels

We'll dig deeper into some of the ideas used here in the following lectures:

- *this lecture* discusses the key concept of **exchangeability** that rationalizes statistical learning
- *this lecture* describes **likelihood ratio processes** and their role in frequentist and Bayesian statistical theories
- *this lecture* discusses the role of likelihood ratio processes in **Bayesian learning**
- *this lecture* returns to the subject of this lecture and studies whether the Captain's hunch that the (frequentist) decision rule that the Navy had ordered him to use can be expected to be better or worse than the rule sequential rule that Abraham Wald designed

CHAPTER

FORTYFIVE

EXCHANGEABILITY AND BAYESIAN UPDATING

Contents

45.1 Overview

This lecture studies an example of learning via Bayes' Law.

We touch on foundations of Bayesian statistical inference invented by Bruno DeFinetti [dF37].

The relevance of DeFinetti's work for economists is presented forcefully in chapter 11 of [Kre88] by David Kreps.

The example that we study in this lecture is a key component of *this lecture* that augments the *classic* job search model of McCall [McC70] by presenting an unemployed worker with a statistical inference problem.

Here we create graphs that illustrate the role that a likelihood ratio plays in Bayes' Law.

We'll use such graphs to provide insights into the mechanics driving outcomes in *this lecture* about learning in an augmented McCall job search model.

Among other things, this lecture discusses connections between the statistical concepts of sequences of random variables that are

- independently and identically distributed
- exchangeable

Understanding the distinction between these concepts is essential for appreciating how Bayesian updating works in our example.

You can read about exchangeability here.

Below, we'll often use

- W to denote a random variable
- w to denote a particular realization of a random variable W

Let's start with some imports:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
from numba import njit, vectorize
from math import gamma
import scipy.optimize as op
from scipy.integrate import quad
import numpy as np
```

45.2 Independently and Identically Distributed

We begin by looking at the notion of an **independently and identically distributed sequence** of random variables.

An independently and identically distributed sequence is often abbreviated as IID.

Two notions are involved, **independently** and **identically** distributed.

A sequence $W_0, W_1, ...$ is **independently distributed** if the joint probability density of the sequence is the **product** of the densities of the components of the sequence.

The sequence $W_0, W_1, ...$ is **independently and identically distributed** if in addition the marginal density of W_t is the same for all $t = 0, 1, ...$

For example, let $p(W_0, W_1, ...)$ be the **joint density** of the sequence and let $p(W_t)$ be the **marginal density** for a particular W_t for all $t = 0, 1, ...$

Then the joint density of the sequence $W_0, W_1, ...$ is IID if

$$p(W_0, W_1, ...) = p(W_0)p(W_1)\cdots$$

so that the joint density is the product of a sequence of identical marginal densities.

45.2.1 IID Means Past Observations Don't Tell Us Anything About Future Observations

If a sequence is random variables is IID, past information provides no information about future realizations.

In this sense, there is **nothing to learn** about the future from the past.

To understand these statements, let the joint distribution of a sequence of random variables $\{W_t\}_{t=0}^T$ that is not necessarily IID, be

$$p(W_T, W_{T-1}, ..., W_1, W_0)$$

Using the laws of probability, we can always factor such a joint density into a product of conditional densities:

$$p(W_T, W_{T-1}, \ldots, W_1, W_0) = p(W_T|W_{T-1}, \ldots, W_0)p(W_{T-1}|W_{T-2}, \ldots, W_0)\cdots \\ \cdots p(W_1|W_0)p(W_0)$$

In general,

$$p(W_t|W_{t-1}, \ldots, W_0) \neq p(W_t)$$

which states that the **conditional density** on the left side does not equal the **marginal density** on the right side.

In the special IID case,

$$p(W_t|W_{t-1}, \ldots, W_0) = p(W_t)$$

and partial history $W_{t-1}, \ldots, W_0$ contains no information about the probability of W_t.

So in the IID case, there is **nothing to learn** about the densities of future random variables from past data.

In the general case, there is something to learn from past data.

We turn next to an instance of this general case.

Please keep your eye out for **what** there is to learn from past data.

45.3 A Setting in Which Past Observations Are Informative

Let $\{W_t\}_{t=0}^{\infty}$ be a sequence of nonnegative scalar random variables with a joint probability distribution constructed as follows.

There are two distinct cumulative distribution functions F and G — with densities f and g for a nonnegative scalar random variable W.

Before the start of time, say at time $t = -1$, "nature" once and for all selects **either** f **or** g — and thereafter at each time $t \geq 0$ draws a random W from the selected distribution.

So the data are permanently generated as independently and identically distributed (IID) draws from **either** F **or** G.

We could say that *objectively* the probability that the data are generated as draws from F is either 0 or 1.

We now drop into this setting a decision maker who knows F and G and that nature picked one of them once and for all and then drew an IID sequence of draws from that distribution.

But our decision maker does not know which of the two distributions nature selected.

The decision maker summarizes his ignorance with a **subjective probability** $\tilde{\pi}$ and reasons as if nature had selected F with probability $\tilde{\pi} \in (0, 1)$ and G with probability $1 - \tilde{\pi}$.

Thus, we assume that the decision maker

- **knows** both F and G
- **doesn't know** which of these two distributions that nature has drawn
- summarizing his ignorance by acting as if or **thinking** that nature chose distribution F with probability $\tilde{\pi} \in (0, 1)$ and distribution G with probability $1 - \tilde{\pi}$
- at date $t \geq 0$ has observed the partial history $w_t, w_{t-1}, \ldots, w_0$ of draws from the appropriate joint density of the partial history

But what do we mean by the *appropriate joint distribution*?

We'll discuss that next and in the process describe the concept of **exchangeability**.

45.4 Relationship Between IID and Exchangeable

Conditional on nature selecting F, the joint density of the sequence $W_0, W_1, ...$ is

$$f(W_0)f(W_1)\cdots$$

Conditional on nature selecting G, the joint density of the sequence $W_0, W_1, ...$ is

$$g(W_0)g(W_1)\cdots$$

Notice that **conditional on nature having selected** F, the sequence $W_0, W_1, ...$ is independently and identically distributed.

Furthermore, **conditional on nature having selected** G, the sequence $W_0, W_1, ...$ is also independently and identically distributed.

But what about the unconditional distribution?

The unconditional distribution of $W_0, W_1, ...$ is evidently

$$h(W_0, W_1, ...) \equiv \tilde{\pi}[f(W_0)f(W_1)\cdots] + (1-\tilde{\pi})[g(W_0)g(W_1)\cdots] \tag{1}$$

Under the unconditional distribution $h(W_0, W_1, ...)$, the sequence $W_0, W_1, ...$ is **not** independently and identically distributed.

To verify this claim, it is sufficient to notice, for example, that

$$h(w_0, w_1) = \tilde{\pi}f(w_0)f(w_1) + (1-\tilde{\pi})g(w_0)g(w_1) \neq (\tilde{\pi}f(w_0) + (1-\tilde{\pi})g(w_0))(\tilde{\pi}f(w_1) + (1-\tilde{\pi})g(w_1))$$

Thus, the conditional distribution

$$h(w_1|w_0) \equiv \frac{h(w_0, w_1)}{(\tilde{\pi}f(w_0) + (1-\tilde{\pi})g(w_0))} \neq (\tilde{\pi}f(w_1) + (1-\tilde{\pi})g(w_1))$$

This means that the realization w_0 contains information about w_1.

So there is something to learn.

But what and how?

45.5 Exchangeability

While the sequence $W_0, W_1, ...$ is not IID, it can be verified that it is **exchangeable**, which means that

$$h(w_0, w_1) = h(w_1, w_0)$$

and so on.

More generally, a sequence of random variables is said to be **exchangeable** if the joint probability distribution for the sequence does not change when the positions in the sequence in which finitely many of the random variables appear are altered.

Equation (1) represents our instance of an exchangeable joint density over a sequence of random variables as a **mixture** of two IID joint densities over a sequence of random variables.

For a Bayesian statistician, the mixing parameter $\tilde{\pi} \in (0, 1)$ has a special interpretation as a **prior probability** that nature selected probability distribution F.

DeFinetti [dF37] established a related representation of an exchangeable process created by mixing sequences of IID Bernoulli random variables with parameters θ and mixing probability $\pi(\theta)$ for a density $\pi(\theta)$ that a Bayesian statistician would interpret as a prior over the unknown Bernoulli parameter θ.

45.6 Bayes' Law

We noted above that in our example model there is something to learn about about the future from past data drawn from our particular instance of a process that is exchangeable but not IID.

But how can we learn?

And about what?

The answer to the *about what* question is about $\tilde{\pi}$.

The answer to the *how* question is to use Bayes' Law.

Another way to say *use Bayes' Law* is to say *compute an appropriate conditional distribution*.

Let's dive into Bayes' Law in this context.

Let q represent the distribution that nature actually draws from w from and let

$$\pi = \mathbb{P}\{q = f\}$$

where we regard π as the decision maker's **subjective probability** (also called a **personal probability**).

Suppose that at $t \geq 0$, the decision maker has observed a history $w^t \equiv [w_t, w_{t-1}, \ldots, w_0]$.

We let

$$\pi_t = \mathbb{P}\{q = f | w^t\}$$

where we adopt the convention

$$\pi_{-1} = \tilde{\pi}$$

The distribution of w_{t+1} conditional on w^t is then

$$\pi_t f + (1 - \pi_t) g.$$

Bayes' rule for updating π_{t+1} is

$$\pi_{t+1} = \frac{\pi_t f(w_{t+1})}{\pi_t f(w_{t+1}) + (1 - \pi_t) g(w_{t+1})} \tag{2}$$

The last expression follows from Bayes' rule, which tells us that

$$\mathbb{P}\{q = f \,|\, W = w\} = \frac{\mathbb{P}\{W = w \,|\, q = f\}\mathbb{P}\{q = f\}}{\mathbb{P}\{W = w\}} \quad \text{and} \quad \mathbb{P}\{W = w\} = \sum_{\omega \in \{f, g\}} \mathbb{P}\{W = w \,|\, q = \omega\}\mathbb{P}\{q = \omega\}$$

45.7 More Details about Bayesian Updating

Let's stare at and rearrange Bayes' Law as represented in equation (2) with the aim of understanding how the **posterior** π_{t+1} is influenced by the **prior** π_t and the **likelihood ratio**

$$l(w) = \frac{f(w)}{g(w)}$$

It is convenient for us to rewrite the updating rule (2) as

$$\pi_{t+1} = \frac{\pi_t f\left(w_{t+1}\right)}{\pi_t f\left(w_{t+1}\right) + \left(1 - \pi_t\right) g\left(w_{t+1}\right)} = \frac{\pi_t \frac{f(w_{t+1})}{g(w_{t+1})}}{\pi_t \frac{f(w_{t+1})}{g(w_{t+1})} + \left(1 - \pi_t\right)} = \frac{\pi_t l\left(w_{t+1}\right)}{\pi_t l\left(w_{t+1}\right) + \left(1 - \pi_t\right)}$$

This implies that

$$\frac{\pi_{t+1}}{\pi_t} = \frac{l\left(w_{t+1}\right)}{\pi_t l\left(w_{t+1}\right) + \left(1-\pi_t\right)} \begin{cases} > 1 & \text{if } l\left(w_{t+1}\right) > 1 \\ \leq 1 & \text{if } l\left(w_{t+1}\right) \leq 1 \end{cases} \tag{3}$$

Notice how the likelihood ratio and the prior interact to determine whether an observation w_{t+1} leads the decision maker to increase or decrease the subjective probability he/she attaches to distribution F.

When the likelihood ratio $l(w_{t+1})$ exceeds one, the observation w_{t+1} nudges the probability π put on distribution F upward, and when the likelihood ratio $l(w_{t+1})$ is less that one, the observation w_{t+1} nudges π downward.

Representation (3) is the foundation of the graphs that we'll use to display the dynamics of $\{\pi_t\}_{t=0}^{\infty}$ that are induced by Bayes' Law.

We'll plot $l\left(w\right)$ as a way to enlighten us about how learning – i.e., Bayesian updating of the probability π that nature has chosen distribution f – works.

To create the Python infrastructure to do our work for us, we construct a wrapper function that displays informative graphs given parameters of f and g.

```
@vectorize
def p(x, a, b):
    "The general beta distribution function."
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * x ** (a-1) * (1 - x) ** (b-1)

def learning_example(F_a=1, F_b=1, G_a=3, G_b=1.2):
    """
    A wrapper function that displays the updating rule of belief π,
    given the parameters which specify F and G distributions.
    """

    f = njit(lambda x: p(x, F_a, F_b))
    g = njit(lambda x: p(x, G_a, G_b))

    # l(w) = f(w) / g(w)
    l = lambda w: f(w) / g(w)
    # objective function for solving l(w) = 1
    obj = lambda w: l(w) - 1

    x_grid = np.linspace(0, 1, 100)
    π_grid = np.linspace(1e-3, 1-1e-3, 100)

    w_max = 1
    w_grid = np.linspace(1e-12, w_max-1e-12, 100)

    # the mode of beta distribution
    # use this to divide w into two intervals for root finding
    G_mode = (G_a - 1) / (G_a + G_b - 2)
    roots = np.empty(2)
    roots[0] = op.root_scalar(obj, bracket=[1e-10, G_mode]).root
    roots[1] = op.root_scalar(obj, bracket=[G_mode, 1-1e-10]).root

    fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 5))

    ax1.plot(l(w_grid), w_grid, label='$l$', lw=2)
    ax1.vlines(1., 0., 1., linestyle="--")
    ax1.hlines(roots, 0., 2., linestyle="--")
    ax1.set_xlim([0., 2.])
```

(continues on next page)

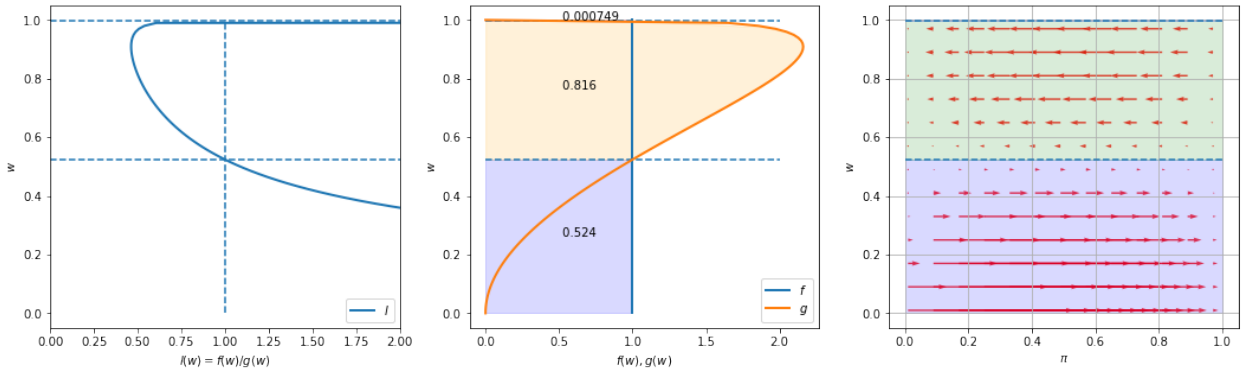
(continued from previous page)

```
    ax1.legend(loc=4)
    ax1.set(xlabel='$l(w)=f(w)/g(w)$', ylabel='$w$')

    ax2.plot(f(x_grid), x_grid, label='$f$', lw=2)
    ax2.plot(g(x_grid), x_grid, label='$g$', lw=2)
    ax2.vlines(1., 0., 1., linestyle="--")
    ax2.hlines(roots, 0., 2., linestyle="--")
    ax2.legend(loc=4)
    ax2.set(xlabel='$f(w), g(w)$', ylabel='$w$')

    area1 = quad(f, 0, roots[0])[0]
    area2 = quad(g, roots[0], roots[1])[0]
    area3 = quad(f, roots[1], 1)[0]

    ax2.text((f(0) + f(roots[0])) / 4, roots[0] / 2, f"{area1: .3g}")
    ax2.fill_between([0, 1], 0, roots[0], color='blue', alpha=0.15)
    ax2.text(np.mean(g(roots)) / 2, np.mean(roots), f"{area2: .3g}")
    w_roots = np.linspace(roots[0], roots[1], 20)
    ax2.fill_betweenx(w_roots, 0, g(w_roots), color='orange', alpha=0.15)
    ax2.text((f(roots[1]) + f(1)) / 4, (roots[1] + 1) / 2, f"{area3: .3g}")
    ax2.fill_between([0, 1], roots[1], 1, color='blue', alpha=0.15)

    W = np.arange(0.01, 0.99, 0.08)
    Π = np.arange(0.01, 0.99, 0.08)

    ΔW = np.zeros((len(W), len(Π)))
    ΔΠ = np.empty((len(W), len(Π)))
    for i, w in enumerate(W):
        for j, π in enumerate(Π):
            lw = l(w)
            ΔΠ[i, j] = π * (lw / (π * lw + 1 - π) - 1)

    q = ax3.quiver(Π, W, ΔΠ, ΔW, scale=2, color='r', alpha=0.8)

    ax3.fill_between(π_grid, 0, roots[0], color='blue', alpha=0.15)
    ax3.fill_between(π_grid, roots[0], roots[1], color='green', alpha=0.15)
    ax3.fill_between(π_grid, roots[1], w_max, color='blue', alpha=0.15)
    ax3.hlines(roots, 0., 1., linestyle="--")
    ax3.set(xlabel='$\pi$', ylabel='$w$')
    ax3.grid()

    plt.show()
```

Now we'll create a group of graphs designed to illustrate the dynamics induced by Bayes' Law.

We'll begin with the default values of various objects, then change them in a subsequent example.

```
learning_example()
```



Please look at the three graphs above created for an instance in which f is a uniform distribution on $[0, 1]$ (i.e., a Beta distribution with parameters $F_a = 1, F_b = 1$), while g is a Beta distribution with the default parameter values $G_a = 3, G_b = 1.2$.

The graph on the left plots the likelihood ratio $l(w)$ on the coordinate axis against w on the ordinate axis.

The middle graph plots both $f(w)$ and $g(w)$ against w, with the horizontal dotted lines showing values of w at which the likelihood ratio equals 1.

The graph on the right plots arrows to the right that show when Bayes' Law makes π increase and arrows to the left that show when Bayes' Law make π decrease.

Notice how the length of the arrows, which show the magnitude of the force from Bayes' Law impelling π to change, depends on both the prior probability π on the ordinate axis and the evidence in the form of the current draw of w on the coordinate axis.

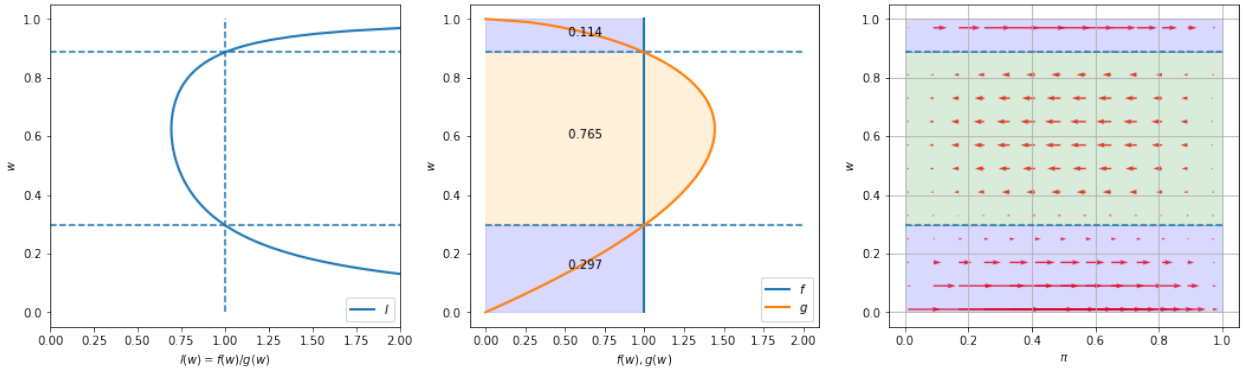
The fractions in the colored areas of the middle graphs are probabilities under F and G, respectively, that realizations of w fall into the interval that updates the belief π in a correct direction (i.e., toward 0 when G is the true distribution, and towards 1 when F is the true distribution).

For example, in the above example, under true distribution F, π will be updated toward 0 if w falls into the interval $[0.524, 0.999]$, which occurs with probability $1 - .524 = .476$ under F. But this would occur with probability 0.816 if G were the true distribution. The fraction 0.816 in the orange region is the integral of $g(w)$ over this interval.

Next we use our code to create graphs for another instance of our model.

We keep F the same as in the preceding instance, namely a uniform distribution, but now assume that G is a Beta distribution with parameters $G_a = 2, G_b = 1.6$.

```
learning_example(G_a=2, G_b=1.6)
```



Notice how the likelihood ratio, the middle graph, and the arrows compare with the previous instance of our example.

45.8 Appendix

45.8.1 Sample Paths of π_t

Now we'll have some fun by plotting multiple realizations of sample paths of π_t under two possible assumptions about nature's choice of distribution:

- that nature permanently draws from F
- that nature permanently draws from G

Outcomes depend on a peculiar property of likelihood ratio processes that are discussed in this lecture.

To do this, we create some Python code.

```
def function_factory(F_a=1, F_b=1, G_a=3, G_b=1.2):

    # define f and g
    f = njit(lambda x: p(x, F_a, F_b))
    g = njit(lambda x: p(x, G_a, G_b))

    @njit
    def update(a, b, π):
        "Update π by drawing from beta distribution with parameters a and b"

        # Draw
        w = np.random.beta(a, b)

        # Update belief
        π = 1 / (1 + ((1 - π) * g(w)) / (π * f(w)))

        return π

    @njit
    def simulate_path(a, b, T=50):
        "Simulates a path of beliefs π with length T"

        π = np.empty(T+1)

        # initial condition
        π[0] = 0.5

        for t in range(1, T+1):
            π[t] = update(a, b, π[t-1])

        return π

    def simulate(a=1, b=1, T=50, N=200, display=True):
        "Simulates N paths of beliefs π with length T"

        π_paths = np.empty((N, T+1))
        if display:
            fig = plt.figure()

        for i in range(N):
            π_paths[i] = simulate_path(a=a, b=b, T=T)
            if display:
                plt.plot(range(T+1), π_paths[i], color='b', lw=0.8, alpha=0.5)
```

(continues on next page)

(continued from previous page)

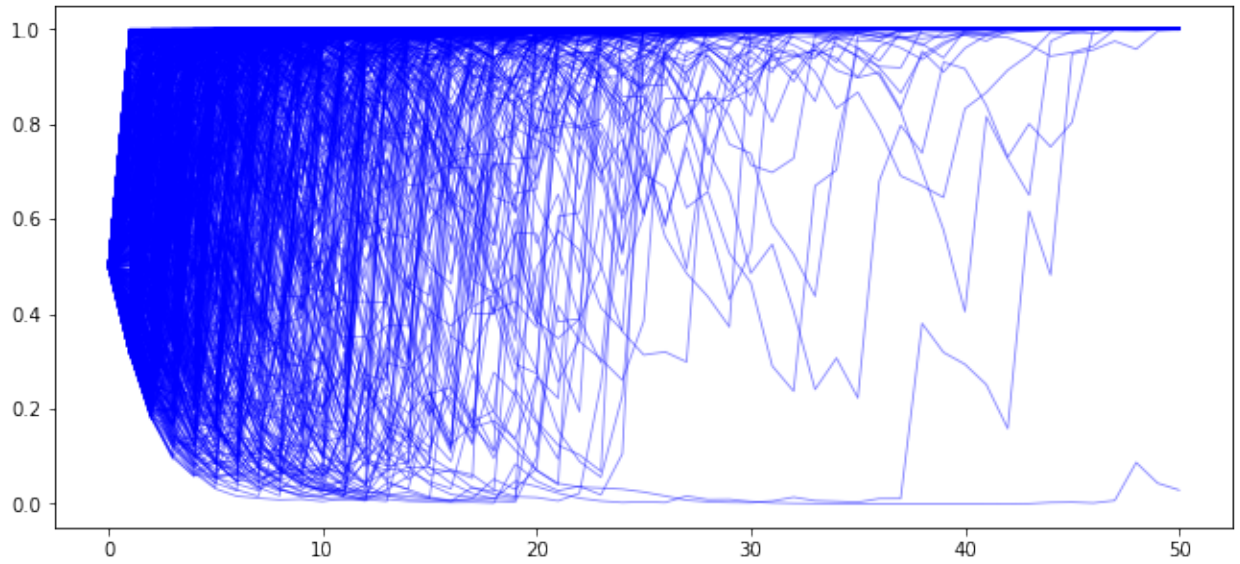
```
        if display:
            plt.show()

        return π_paths

    return simulate
```

```
simulate = function_factory()
```

We begin by generating N simulated $\{\pi_t\}$ paths with T periods when the sequence is truly IID draws from F. We set the initial prior $\pi_{-1} = .5$.

```
T = 50
```

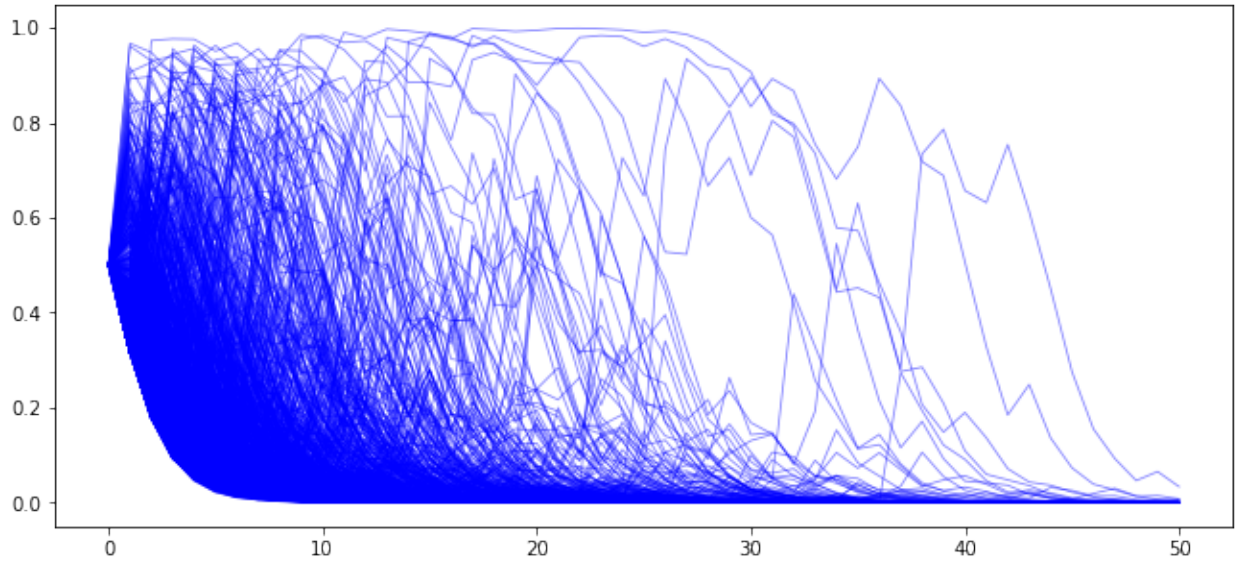
```
# when nature selects F
π_paths_F = simulate(a=1, b=1, T=T, N=1000)
```



In the above graph we observe that for most paths $\pi_t \to 1$. So Bayes' Law evidently eventually discovers the truth for most of our paths.

Next, we generate paths with T periods when the sequence is truly IID draws from G. Again, we set the initial prior $\pi_{-1} = .5$.

```
# when nature selects G
π_paths_G = simulate(a=3, b=1.2, T=T, N=1000)
```



In the above graph we observe that now most paths $\pi_t \to 0$.

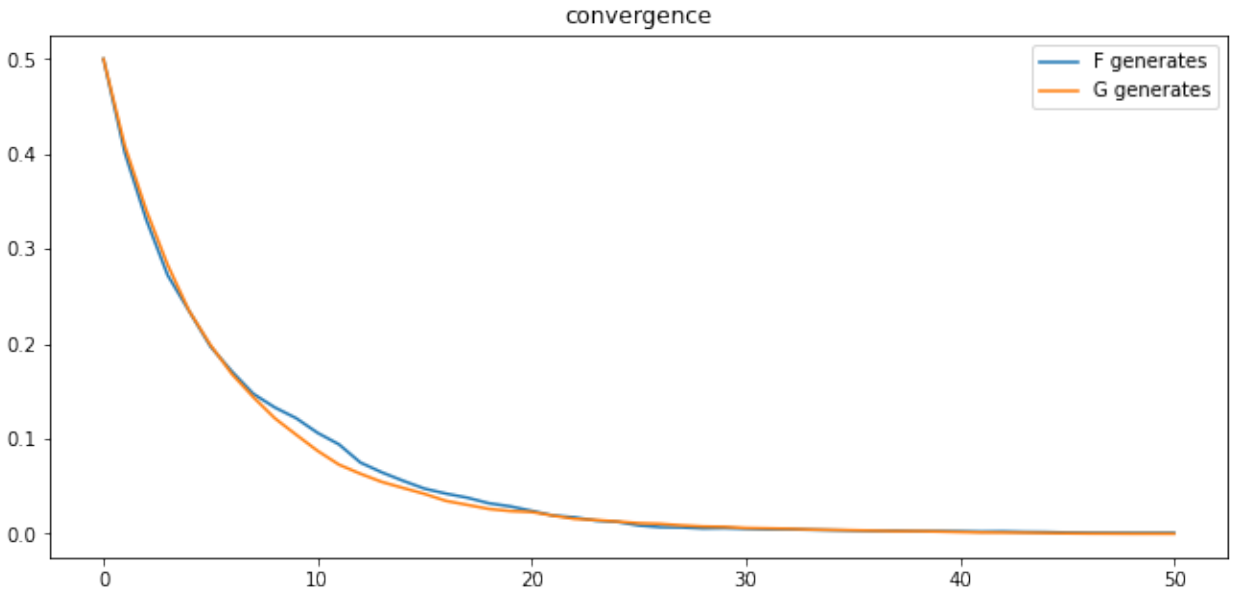
45.8.2 Rates of convergence

We study rates of convergence of π_t to 1 when nature generates the data as IID draws from F and of π_t to 0 when nature generates the data as IID draws from G.

We do this by averaging across simulated paths of $\{\pi_t\}_{t=0}^T$.

Using N simulated π_t paths, we compute $1 - \sum_{i=1}^N \pi_{i,t}$ at each t when the data are generated as draws from F and compute $\sum_{i=1}^N \pi_{i,t}$ when the data are generated as draws from G.

```
plt.plot(range(T+1), 1 - np.mean(π_paths_F, 0), label='F generates')
plt.plot(range(T+1), np.mean(π_paths_G, 0), label='G generates')
plt.legend()
plt.title("convergence");
```



From the above graph, rates of convergence appear not to depend on whether F or G generates the data.

45.8.3 Another Graph of Population Dynamics of π_t

More insights about the dynamics of $\{\pi_t\}$ can be gleaned by computing the following conditional expectations of $\frac{\pi_{t+1}}{\pi_t}$ as functions of π_t via integration with respect to the pertinent probability distribution:

$$
\begin{aligned}
E\left[\frac{\pi_{t+1}}{\pi_t}\,\middle|\, q=\omega, \pi_t\right] &= E\left[\frac{l\left(w_{t+1}\right)}{\pi_t l\left(w_{t+1}\right)+\left(1-\pi_t\right)}\,\middle|\, q=\omega, \pi_t\right], \\
&= \int_0^1 \frac{l\left(w_{t+1}\right)}{\pi_t l\left(w_{t+1}\right)+\left(1-\pi_t\right)} \omega\left(w_{t+1}\right) dw_{t+1}
\end{aligned}
$$

where $\omega = f, g$.

The following code approximates the integral above:

```
def expected_ratio(F_a=1, F_b=1, G_a=3, G_b=1.2):

    # define f and g
    f = njit(lambda x: p(x, F_a, F_b))
    g = njit(lambda x: p(x, G_a, G_b))

    l = lambda w: f(w) / g(w)
    integrand_f = lambda w, π: f(w) * l(w) / (π * l(w) + 1 - π)
    integrand_g = lambda w, π: g(w) * l(w) / (π * l(w) + 1 - π)

    π_grid = np.linspace(0.02, 0.98, 100)

    expected_rario = np.empty(len(π_grid))
    for q, inte in zip(["f", "g"], [integrand_f, integrand_g]):
        for i, π in enumerate(π_grid):
            expected_rario[i]= quad(inte, 0, 1, args=(π,))[0]
        plt.plot(π_grid, expected_rario, label=f"{q} generates")
```

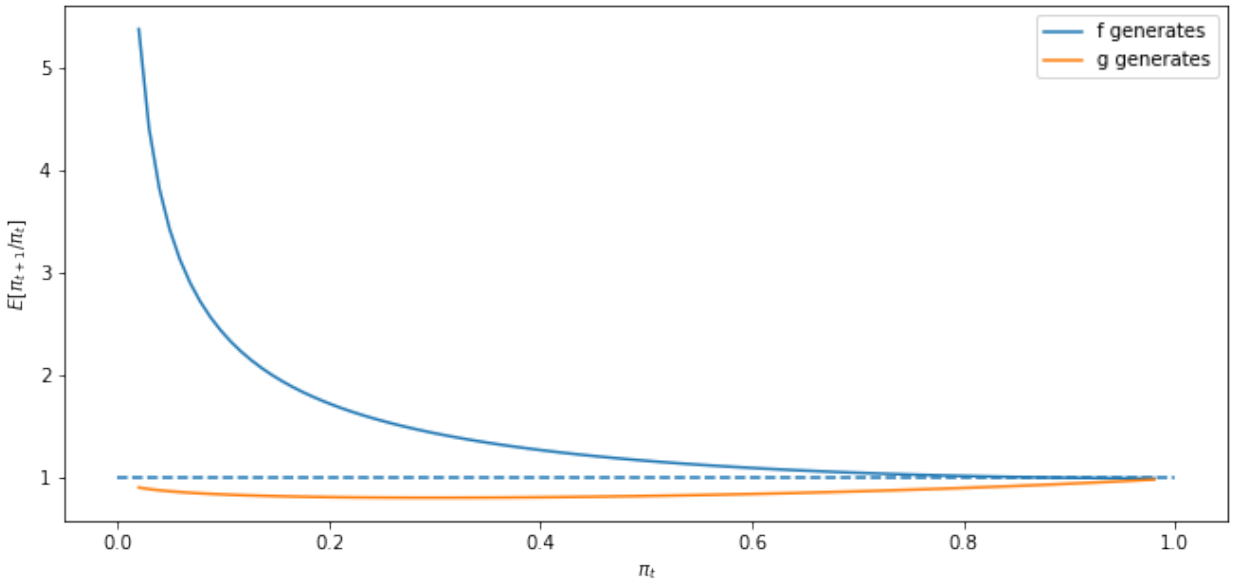
(continues on next page)

(continued from previous page)

```
    plt.hlines(1, 0, 1, linestyle="--")
    plt.xlabel("$π_t$")
    plt.ylabel("$E[\pi_{t+1}/\pi_t]$")
    plt.legend()

    plt.show()
```

First, consider the case where $F_a = F_b = 1$ and $G_a = 3, G_b = 1.2$.

```
expected_ratio()
```

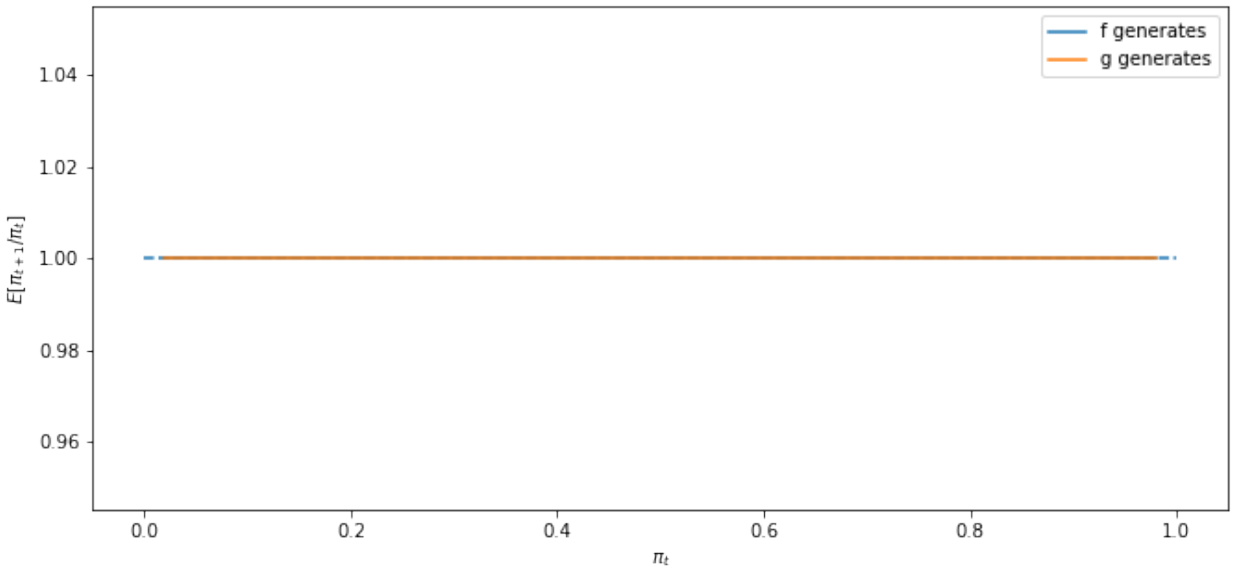


The above graphs shows that when F generates the data, π_t on average always heads north, while when G generates the data, π_t heads south.

Next, we'll look at a degenerate case in whcih f and g are identical beta distributions, and $F_a = G_a = 3, F_b = G_b = 1.2$.

In a sense, here there is nothing to learn.

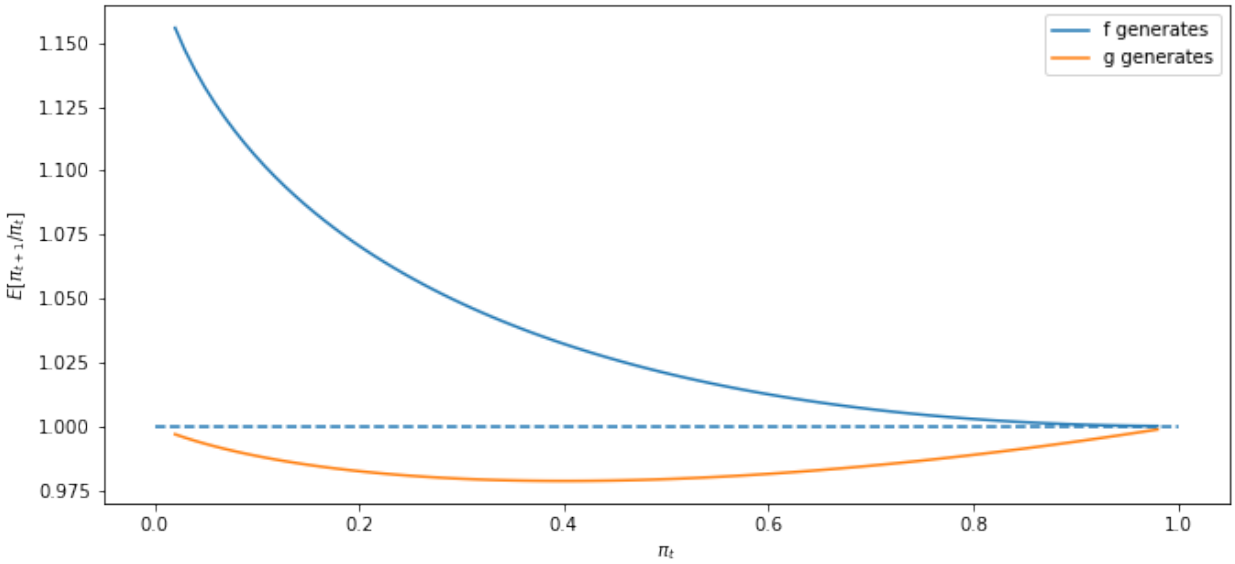
```
expected_ratio(F_a=3, F_b=1.2)
```



The above graph says that π_t is inert and would remain at its initial value.

Finally, let's look at a case in which f and g are neither very different nor identical, in particular one in which $F_a = 2, F_b = 1$ and $G_a = 3, G_b = 1.2$.

```
expected_ratio(F_a=2, F_b=1, G_a=3, G_b=1.2)
```



45.9 Sequels

We'll dig deeper into some of the ideas used here in the following lectures:

- *this lecture* describes **likelihood ratio processes** and their role in frequentist and Bayesian statistical theories
- *this lecture* returns to the subject of this lecture and studies whether the Captain's hunch that the (frequentist) decision rule that the Navy had ordered him to use can be expected to be better or worse than the rule sequential rule that Abraham Wald designed

CHAPTER

FORTYSIX

LIKELIHOOD RATIO PROCESSES AND BAYESIAN LEARNING

Contents


```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from numba import vectorize, njit
from math import gamma
```

46.1 Overview

This lecture describes the role that **likelihood ratio processes** play in **Bayesian learning**.

As in *this lecture*, we'll use a simple statistical setting from *this lecture*.

We'll focus on how a likelihood ratio process and a **prior** probability determine a **posterior** probability.

We'll derive a convenient recursion for today's posterior as a function of yesterday's posterior and today's multiplicative increment to a likelihood process.

We'll also present a useful generalization of that formula that represents today's posterior in terms of an initial prior and today's realization of the likelihood ratio process.

We'll study how, at least in our setting, a Bayesian eventually learns the probability distribution that generates the data, an outcome that rests on the asymptotic behavior of likelihood ratio processes studied in *this lecture*.

This lecture provides technical results that underly outcomes to be studied in *this lecture* and *this lecture* and *this lecture*.

46.2 The Setting

We begin by reviewing the setting in *this lecture*, which we adopt here too.

A nonnegative random variable W has one of two probability density functions, either f or g.

Before the beginning of time, nature once and for all decides whether she will draw a sequence of IID draws from f or from g.

We will sometimes let q be the density that nature chose once and for all, so that q is either f or g, permanently.

Nature knows which density it permanently draws from, but we the observers do not.

We do know both f and g, but we don't know which density nature chose.

But we want to know.

To do that, we use observations.

We observe a sequence $\{w_t\}_{t=1}^T$ of T IID draws from either f or g.

We want to use these observations to infer whether nature chose f or g.

A **likelihood ratio process** is a useful tool for this task.

To begin, we define the key component of a likelihood ratio process, namely, the time t likelihood ratio as the random variable

$$\ell(w_t) = \frac{f(w_t)}{g(w_t)}, \quad t \geq 1.$$

We assume that f and g both put positive probabilities on the same intervals of possible realizations of the random variable W.

That means that under the g density, $\ell(w_t) = \frac{f(w_t)}{g(w_t)}$ is evidently a nonnegative random variable with mean 1.

A **likelihood ratio process** for sequence $\{w_t\}_{t=1}^\infty$ is defined as

$$L(w^t) = \prod_{i=1}^t \ell(w_i),$$

where $w^t = \{w_1, \ldots, w_t\}$ is a history of observations up to and including time t.

Sometimes for shorthand we'll write $L_t = L(w^t)$.

Notice that the likelihood process satisfies the *recursion* or *multiplicative decomposition*

$$L(w^t) = \ell(w_t) L(w^{t-1}).$$

The likelihood ratio and its logarithm are key tools for making inferences using a classic frequentist approach due to Neyman and Pearson [NP33].

We'll again deploy the following Python code from *this lecture* that evaluates f and g as two different beta distributions, then computes and simulates an associated likelihood ratio process by generating a sequence w^t from *some* probability distribution, for example, a sequence of IID draws from g.

```
# Parameters in the two beta distributions.
F_a, F_b = 1, 1
G_a, G_b = 3, 1.2

@vectorize
def p(x, a, b):
```

(continues on next page)

(continued from previous page)

```
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * x** (a-1) * (1 - x) ** (b-1)

# The two density functions.
f = njit(lambda x: p(x, F_a, F_b))
g = njit(lambda x: p(x, G_a, G_b))
```

```
@njit
def simulate(a, b, T=50, N=500):
    '''
    Generate N sets of T observations of the likelihood ratio,
    return as N x T matrix.

    '''

    l_arr = np.empty((N, T))

    for i in range(N):

        for j in range(T):
            w = np.random.beta(a, b)
            l_arr[i, j] = f(w) / g(w)

    return l_arr
```

We'll also use the following Python code to prepare some informative simulations

```
l_arr_g = simulate(G_a, G_b, N=50000)
l_seq_g = np.cumprod(l_arr_g, axis=1)
```

```
l_arr_f = simulate(F_a, F_b, N=50000)
l_seq_f = np.cumprod(l_arr_f, axis=1)
```

46.3 Likelihood Ratio Process and Bayes' Law

Let π_t be a Bayesian posterior defined as

$$\pi_t = \text{Prob}(q = f|w^t)$$

The likelihood ratio process is a principal actor in the formula that governs the evolution of the posterior probability π_t, an instance of **Bayes' Law**.

Bayes' law implies that $\{\pi_t\}$ obeys the recursion

$$\pi_t = \frac{\pi_{t-1} l_t(w_t)}{\pi_{t-1} l_t(w_t) + 1 - \pi_{t-1}} \tag{1}$$

with π_0 being a Bayesian prior probability that $q = f$, i.e., a personal or subjective belief about q based on our having seen no data.

Below we define a Python function that updates belief π using likelihood ratio ℓ according to recursion (1)

```
@njit
def update(π, l):
    "Update π using likelihood l"

    # Update belief
    π = π * l / (π * l + 1 - π)

    return π
```

Formula (1) can be generalized by iterating on it and thereby deriving an expression for the time t posterior π_{t+1} as a function of the time 0 prior π_0 and the likelihood ratio process $L(w^{t+1})$ at time t.

To begin, notice that the updating rule

$$\pi_{t+1} = \frac{\pi_t \ell\left(w_{t+1}\right)}{\pi_t \ell\left(w_{t+1}\right) + \left(1 - \pi_t\right)}$$

implies

$$\begin{aligned}\frac{1}{\pi_{t+1}} &= \frac{\pi_t \ell\left(w_{t+1}\right) + \left(1 - \pi_t\right)}{\pi_t \ell\left(w_{t+1}\right)} \\ &= 1 - \frac{1}{\ell\left(w_{t+1}\right)} + \frac{1}{\ell\left(w_{t+1}\right)} \frac{1}{\pi_t}. \\ \Rightarrow \frac{1}{\pi_{t+1}} - 1 &= \frac{1}{\ell\left(w_{t+1}\right)} \left(\frac{1}{\pi_t} - 1\right).\end{aligned}$$

Therefore

$$\frac{1}{\pi_{t+1}} - 1 = \frac{1}{\prod_{i=1}^{t+1} \ell\left(w_i\right)} \left(\frac{1}{\pi_0} - 1\right) = \frac{1}{L\left(w^{t+1}\right)} \left(\frac{1}{\pi_0} - 1\right).$$

Since $\pi_0 \in (0, 1)$ and $L\left(w^{t+1}\right) > 0$, we can verify that $\pi_{t+1} \in (0, 1)$.

After rearranging the preceding equation, we can express π_{t+1} as a function of $L\left(w^{t+1}\right)$, the likelihood ratio process at $t+1$, and the initial prior π_0

$$\pi_{t+1} = \frac{\pi_0 L\left(w^{t+1}\right)}{\pi_0 L\left(w^{t+1}\right) + 1 - \pi_0}. \tag{2}$$

Formula (2) generalizes formula (1).

Formula (2) can be regarded as a one step revision of prior probability π_0 after seeing the batch of data $\{w_i\}_{i=1}^{t+1}$.

Formula (2) shows the key role that the likelihood ratio process $L\left(w^{t+1}\right)$ plays in determining the posterior probability π_{t+1}.

Formula (2) is the foundation for the insight that, because of how the likelihood ratio process behaves as $t \to +\infty$, the likelihood ratio process dominates the initial prior π_0 in determining the limiting behavior of π_t.

To illustrate this insight, below we will plot graphs showing **one** simulated path of the likelihood ratio process L_t along with two paths of π_t that are associated with the *same* realization of the likelihood ratio process but *different* initial prior probabilities π_0.

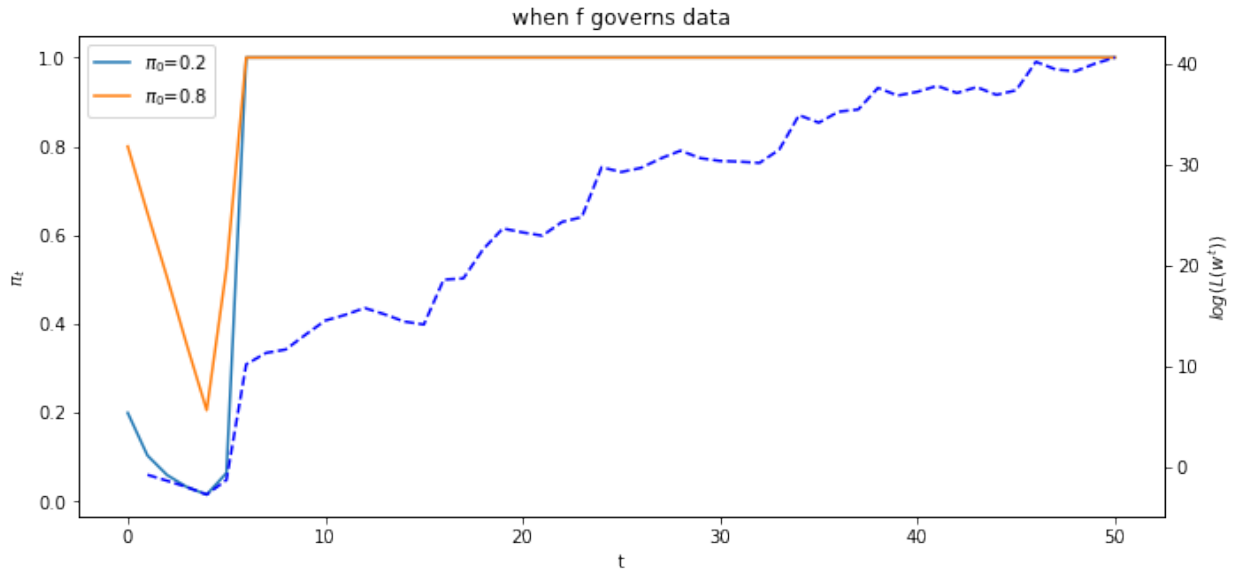
First, we tell Python two values of π_0.

```
π1, π2 = 0.2, 0.8
```

Next we generate paths of the likelihood ratio process L_t and the posterior π_t for a history of IID draws from density f.

```
T = l_arr_f.shape[1]
π_seq_f = np.empty((2, T+1))
π_seq_f[:, 0] = π1, π2

for t in range(T):
    for i in range(2):
        π_seq_f[i, t+1] = update(π_seq_f[i, t], l_arr_f[0, t])
```

```
fig, ax1 = plt.subplots()

for i in range(2):
    ax1.plot(range(T+1), π_seq_f[i, :], label=f"$\pi_0$={π_seq_f[i, 0]}")

ax1.set_ylabel("$\pi_t$")
ax1.set_xlabel("t")
ax1.legend()
ax1.set_title("when f governs data")

ax2 = ax1.twinx()
ax2.plot(range(1, T+1), np.log(l_seq_f[0, :]), '--', color='b')
ax2.set_ylabel("$log(L(w^{t}))$")

plt.show()
```



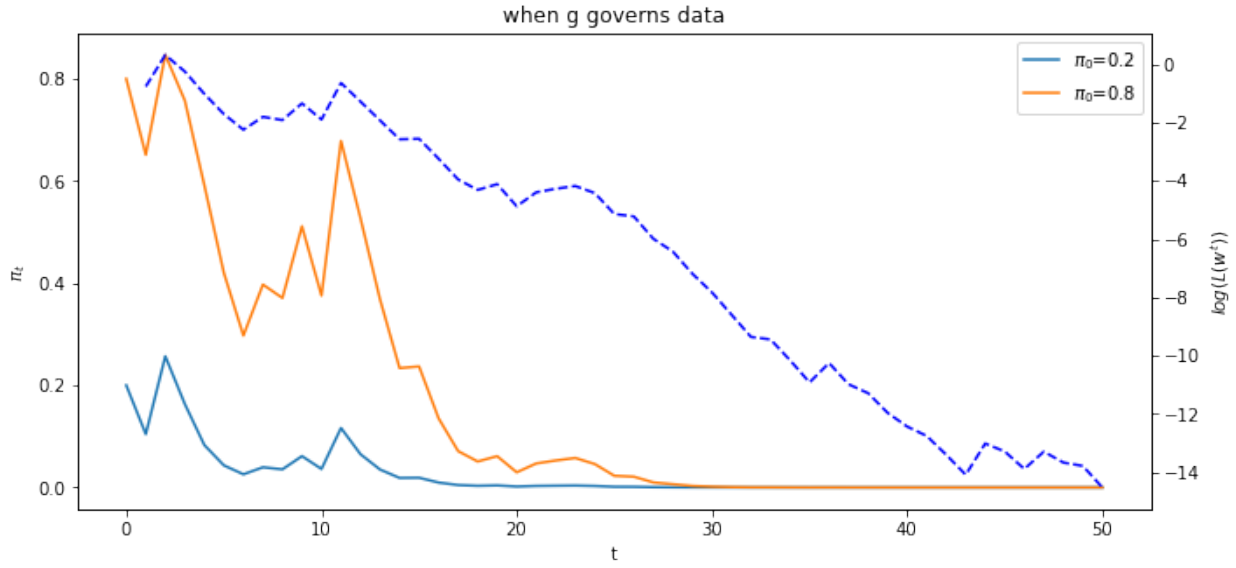
The dotted line in the graph above records the logarithm of the likelihood ratio process $\log L(w^t)$.

Please note that there are two different scales on the y axis.

Now let's study what happens when the history consists of IID draws from density g

```
T = l_arr_g.shape[1]
π_seq_g = np.empty((2, T+1))
π_seq_g[:, 0] = π1, π2

for t in range(T):
    for i in range(2):
        π_seq_g[i, t+1] = update(π_seq_g[i, t], l_arr_g[0, t])
```

```
fig, ax1 = plt.subplots()

for i in range(2):
    ax1.plot(range(T+1), π_seq_g[i, :], label=f"$\pi_0$={π_seq_g[i, 0]}")

ax1.set_ylabel("$\pi_t$")
ax1.set_xlabel("t")
ax1.legend()
ax1.set_title("when g governs data")

ax2 = ax1.twinx()
ax2.plot(range(1, T+1), np.log(l_seq_g[0, :]), '--', color='b')
ax2.set_ylabel("$log(L(w^{t}))$")

plt.show()
```



Below we offer Python code that verifies that nature chose permanently to draw from density f.

```
π_seq = np.empty((2, T+1))
π_seq[:, 0] = π1, π2

for i in range(2):
    πL = π_seq[i, 0] * l_seq_f[0, :]
    π_seq[i, 1:] = πL / (πL + 1 - π_seq[i, 0])
```

```
np.abs(π_seq - π_seq_f).max() < 1e-10
```

```
True
```

We thus conclude that the likelihood ratio process is a key ingredient of the formula (2) for a Bayesian's posteior probabilty that nature has drawn history w^t as repeated draws from density g.

46.4 Sequels

This lecture has been devoted to building some useful infrastructure.

We'll build on results highlighted in this lectures to understand inferences that are the foundations of results described in *this lecture* and *this lecture* and *this lecture*.

CHAPTER

FORTYSEVEN

BAYESIAN VERSUS FREQUENTIST DECISION RULES

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!conda install -y quantecon
!conda install -y -c conda-forge interpolation
```

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from numba import njit, prange, float64, int64
from numba.experimental import jitclass
from interpolation import interp
from math import gamma
from scipy.optimize import minimize
```

47.1 Overview

This lecture follows up on ideas presented in the following lectures:

- *A Problem that Stumped Milton Friedman*
- *Exchangeability and Bayesian Updating*
- *Likelihood Ratio Processes*

In *A Problem that Stumped Milton Friedman* we described a problem that a Navy Captain presented to Milton Friedman during World War II.

The Navy had instructed the Captain to use a decision rule for quality control that the Captain suspected could be dominated by a better rule.

(The Navy had ordered the Captain to use an instance of a **frequentist decision rule**.)

Milton Friedman recognized the Captain's conjecture as posing a challenging statistical problem that he and other members of the US Government's Statistical Research Group at Columbia University proceeded to try to solve.

One of the members of the group, the great mathematician Abraham Wald, soon solved the problem.

A good way to formulate the problem is to use some ideas from Bayesian statistics that we describe in this lecture *Exchangeability and Bayesian Updating* and in this lecture *Likelihood Ratio Processes*, which describes the link between Bayesian updating and likelihood ratio processes.

The present lecture uses Python to generate simulations that evaluate expected losses under **frequentist** and **Bayesian** decision rules for an instance of the Navy Captain's decision problem.

The simulations validate the Navy Captain's hunch that there is a better rule than the one the Navy had ordered him to use.

47.2 Setup

To formalize the problem of the Navy Captain whose questions posed the problem that Milton Friedman and Allan Wallis handed over to Abraham Wald, we consider a setting with the following parts.

- Each period a decision maker draws a non-negative random variable Z from a probability distribution that he does not completely understand. He knows that two probability distributions are possible, f_0 and f_1, and that which ever distribution it is remains fixed over time. The decision maker believes that before the beginning of time, nature once and for all selected either f_0 or f_1 and that the probability that it selected f_0 is probability π^*.
- The decision maker observes a sample $\{z_i\}_{i=0}^t$ from the the distribution chosen by nature.

The decision maker wants to decide which distribution actually governs Z and is worried by two types of errors and the losses that they impose on him.

- a loss $\bar{L}_1$ from a **type I error** that occurs when he decides that $f = f_1$ when actually $f = f_0$
- a loss $\bar{L}_0$ from a **type II error** that occurs when he decides that $f = f_0$ when actually $f = f_1$

The decision maker pays a cost c for drawing another z

We mainly borrow parameters from the quantecon lecture *A Problem that Stumped Milton Friedman* except that we increase both $\bar{L}_0$ and $\bar{L}_1$ from 25 to 100 to encourage the frequentist Navy Captain to take more draws before deciding.

We set the cost c of taking one more draw at 1.25.

We set the probability distributions f_0 and f_1 to be beta distributions with $a_0 = b_0 = 1$, $a_1 = 3$, and $b_1 = 1.2$, respectively.

Below is some Python code that sets up these objects.

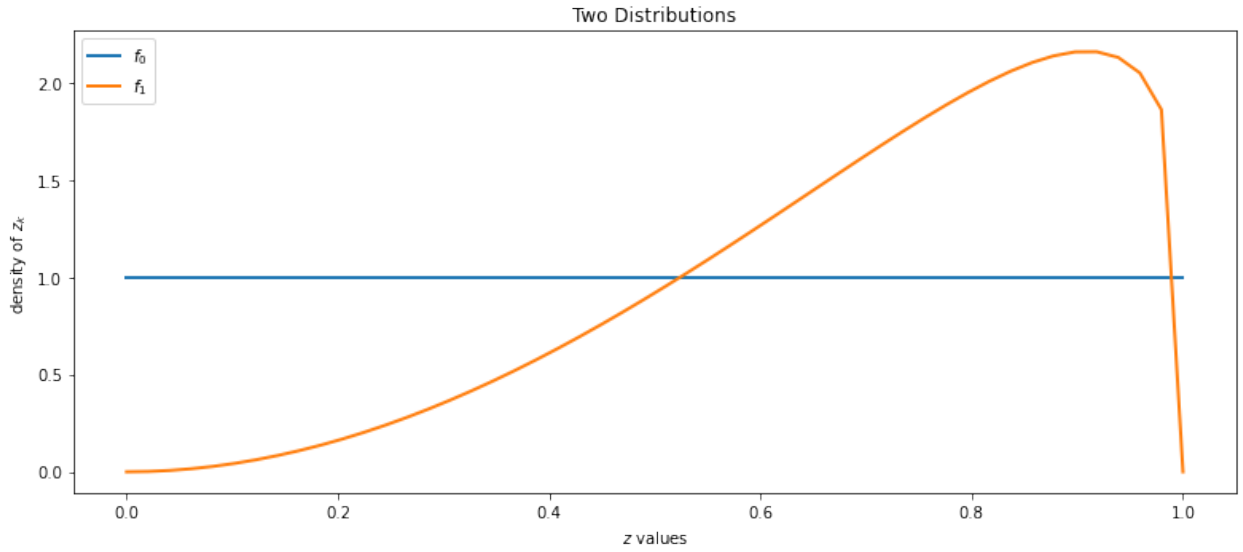
```
@njit
def p(x, a, b):
    "Beta distribution."

    r = gamma(a + b) / (gamma(a) * gamma(b))

    return r * x**(a-1) * (1 - x)**(b-1)
```

We start with defining a `jitclass` that stores parameters and functions we need to solve problems for both the Bayesian and frequentist Navy Captains.

```
wf_data = [
    ('c', float64),           # unemployment compensation
    ('a0', float64),          # parameters of beta distribution
    ('b0', float64),
    ('a1', float64),
    ('b1', float64),
    ('L0', float64),          # cost of selecting f0 when f1 is true
    ('L1', float64),          # cost of selecting f1 when f0 is true
    ('π_grid', float64[:]),   # grid of beliefs π
    ('π_grid_size', int64),
    ('mc_size', int64),       # size of Monto Carlo simulation
    ('z0', float64[:]),       # sequence of random values
    ('z1', float64[:])        # sequence of random values
]
```

```
@jitclass(wf_data)
class WaldFriedman:

    def __init__(self,
                 c=1.25,
                 a0=1,
                 b0=1,
                 a1=3,
                 b1=1.2,
                 L0=100,
                 L1=100,
                 π_grid_size=200,
                 mc_size=1000):

        self.c, self.π_grid_size = c, π_grid_size
        self.a0, self.b0, self.a1, self.b1 = a0, b0, a1, b1
        self.L0, self.L1 = L0, L1
        self.π_grid = np.linspace(0, 1, π_grid_size)
        self.mc_size = mc_size

        self.z0 = np.random.beta(a0, b0, mc_size)
        self.z1 = np.random.beta(a1, b1, mc_size)

    def f0(self, x):

        return p(x, self.a0, self.b0)

    def f1(self, x):

        return p(x, self.a1, self.b1)

    def κ(self, z, π):
        """
        Updates π using Bayes' rule and the current observation z
        """

        a0, b0, a1, b1 = self.a0, self.b0, self.a1, self.b1

        π_f0, π_f1 = π * p(z, a0, b0), (1 - π) * p(z, a1, b1)
        π_new = π_f0 / (π_f0 + π_f1)

        return π_new
```

```
wf = WaldFriedman()

grid = np.linspace(0, 1, 50)

plt.figure()

plt.title("Two Distributions")
plt.plot(grid, wf.f0(grid), lw=2, label="$f_0$")
plt.plot(grid, wf.f1(grid), lw=2, label="$f_1$")

plt.legend()
plt.xlabel("$z$ values")
plt.ylabel("density of $z_k$")

plt.tight_layout()
plt.show()
```



Above, we plot the two possible probability densities f_0 and f_1

47.3 Frequentist Decision Rule

The Navy told the Captain to use a frequentist decision rule.

In particular, it gave him a decision rule that the Navy had designed by using frequentist statistical theory to minimize an expected loss function.

That decision rule is characterized by a sample size t and a cutoff d associated with a likelihood ratio.

Let $L\left(z^t\right) = \prod_{i=0}^{t} \frac{f_0(z_i)}{f_1(z_i)}$ be the likelihood ratio associated with observing the sequence $\{z_i\}_{i=0}^{t}$.

The decision rule associated with a sample size t is:

- decide that f_0 is the distribution if the likelihood ratio is greater than d

To understand how that rule was engineered, let null and alternative hypotheses be

- null: H_0: $f = f_0$,
- alternative H_1: $f = f_1$.

Given sample size t and cutoff d, under the model described above, the mathematical expectation of total loss is

$$\bar{V}_{fre}(t,d) = ct + \pi^* PFA \times \bar{L}_1 + (1-\pi^*)(1-PD) \times \bar{L}_0 \tag{1}$$

$$\text{where} \quad PFA = \Pr\{L(z^t) < d \mid q = f_0\}$$
$$PD = \Pr\{L(z^t) < d \mid q = f_1\}$$

Here

- PFA denotes the probability of a **false alarm**, i.e., rejecting H_0 when it is true
- PD denotes the probability of a **detection error**, i.e., not rejecting H_0 when H_1 is true

For a given sample size t, the pairs (PFA, PD) lie on a **receiver operating characteristic curve** and can be uniquely pinned down by choosing d.

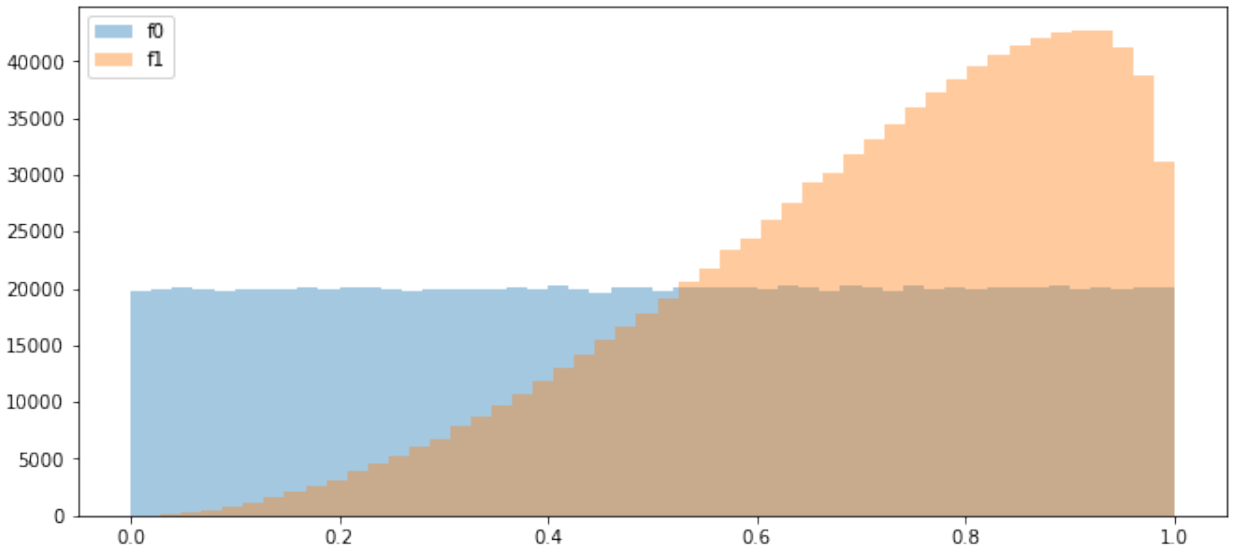
To see some receiver operating characteristic curves, please see this lecture *Likelihood Ratio Processes*.

To solve for $\bar{V}_{fre}(t,d)$ numerically, we first simulate sequences of z when either f_0 or f_1 generates data.

```
N = 10000
T = 100
```

```
z0_arr = np.random.beta(wf.a0, wf.b0, (N, T))
z1_arr = np.random.beta(wf.a1, wf.b1, (N, T))
```

```
plt.hist(z0_arr.flatten(), bins=50, alpha=0.4, label='f0')
plt.hist(z1_arr.flatten(), bins=50, alpha=0.4, label='f1')
plt.legend()
plt.show()
```



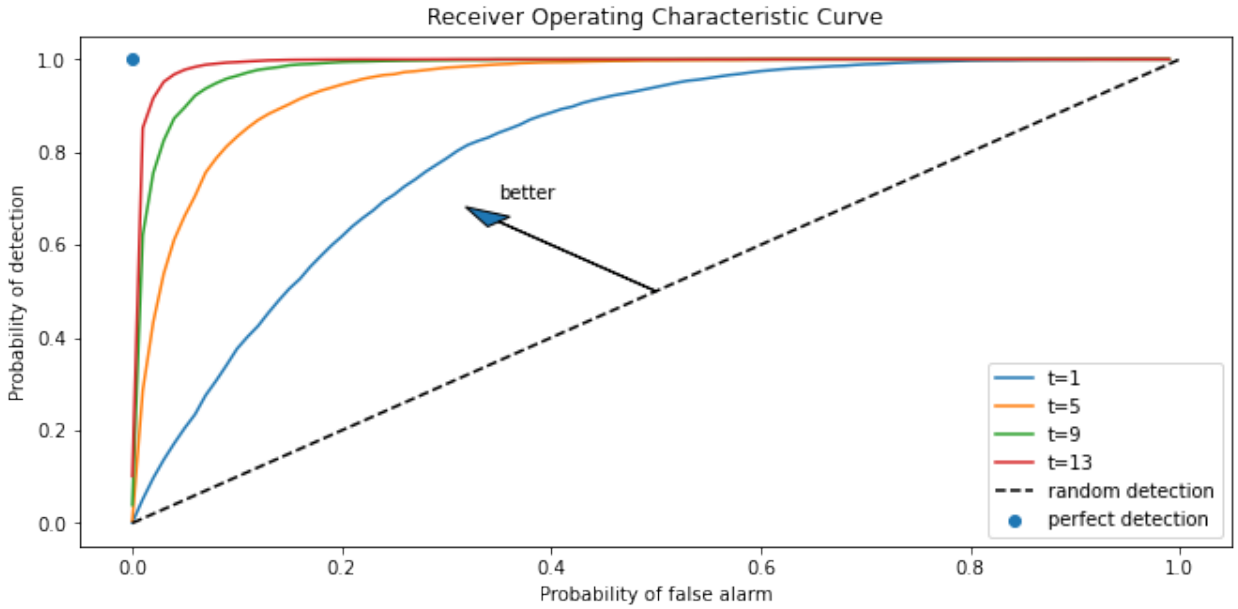
We can compute sequences of likelihood ratios using simulated samples.

```
l = lambda z: wf.f0(z) / wf.f1(z)
```

```
l0_arr = l(z0_arr)
l1_arr = l(z1_arr)

L0_arr = np.cumprod(l0_arr, 1)
L1_arr = np.cumprod(l1_arr, 1)
```

With an empirical distribution of likelihood ratios in hand, we can draw **receiver operating characteristic curves** by enumerating (PFA, PD) pairs given each sample size t.

```
PFA = np.arange(0, 100, 1)

for t in range(1, 15, 4):
    percentile = np.percentile(L0_arr[:, t], PFA)
    PD = [np.sum(L1_arr[:, t] < p) / N for p in percentile]

    plt.plot(PFA / 100, PD, label=f"t={t}")

plt.scatter(0, 1, label="perfect detection")
plt.plot([0, 1], [0, 1], color='k', ls='--', label="random detection")

plt.arrow(0.5, 0.5, -0.15, 0.15, head_width=0.03)
plt.text(0.35, 0.7, "better")
plt.xlabel("Probability of false alarm")
plt.ylabel("Probability of detection")
plt.legend()
plt.title("Receiver Operating Characteristic Curve")
plt.show()
```



Our frequentist minimizes the expected total loss presented in equation (1) by choosing (t, d).

Doing that delivers an expected loss

$$\bar{V}_{fre} = \min_{t,d} \bar{V}_{fre}(t, d).$$

We first consider the case in which $\pi^* = \Pr\{\text{nature selects } f_0\} = 0.5$.

We can solve the minimization problem in two steps.

First, we fix t and find the optimal cutoff d and consequently the minimal $\bar{V}_{fre}(t)$.

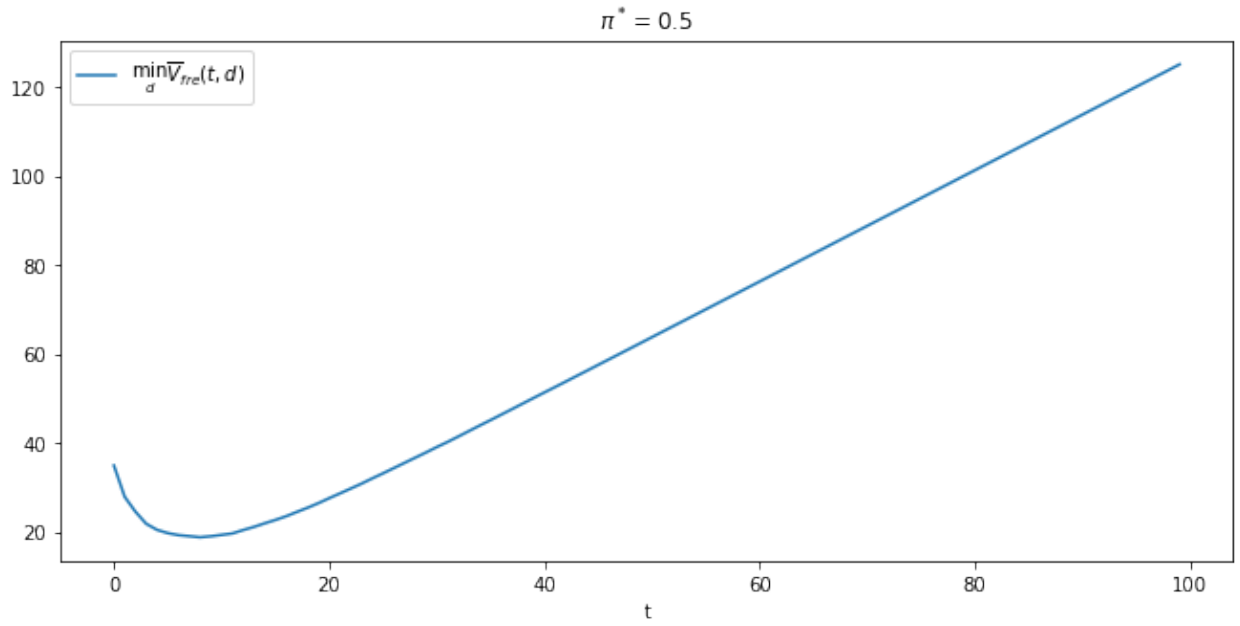
Here is Python code that does that and then plots a useful graph.

```
@njit
def V_fre_d_t(d, t, L0_arr, L1_arr, π_star, wf):

    N = L0_arr.shape[0]

    PFA = np.sum(L0_arr[:, t-1] < d) / N
    PD = np.sum(L1_arr[:, t-1] < d) / N

    V = π_star * PFA *wf. L1 + (1 - π_star) * (1 - PD) * wf.L0

    return V
```

```
def V_fre_t(t, L0_arr, L1_arr, π_star, wf):

    res = minimize(V_fre_d_t, 1, args=(t, L0_arr, L1_arr, π_star, wf), method='Nelder-
↪Mead')
    V = res.fun
    d = res.x

    PFA = np.sum(L0_arr[:, t-1] < d) / N
    PD = np.sum(L1_arr[:, t-1] < d) / N

    return V, PFA, PD
```

```
def compute_V_fre(L0_arr, L1_arr, π_star, wf):

    T = L0_arr.shape[1]

    V_fre_arr = np.empty(T)
    PFA_arr = np.empty(T)
    PD_arr = np.empty(T)

    for t in range(1, T+1):
        V, PFA, PD = V_fre_t(t, L0_arr, L1_arr, π_star, wf)
        V_fre_arr[t-1] = wf.c * t + V
        PFA_arr[t-1] = PFA
        PD_arr[t-1] = PD

    return V_fre_arr, PFA_arr, PD_arr
```

```
π_star = 0.5
V_fre_arr, PFA_arr, PD_arr = compute_V_fre(L0_arr, L1_arr, π_star, wf)

plt.plot(range(T), V_fre_arr, label='$\min_{d} \overline{V}_{fre}(t,d)$')
plt.xlabel('t')
plt.title('$\pi^*=0.5$')
plt.legend()
plt.show()
```




```
t_optimal = np.argmin(V_fre_arr) + 1
```

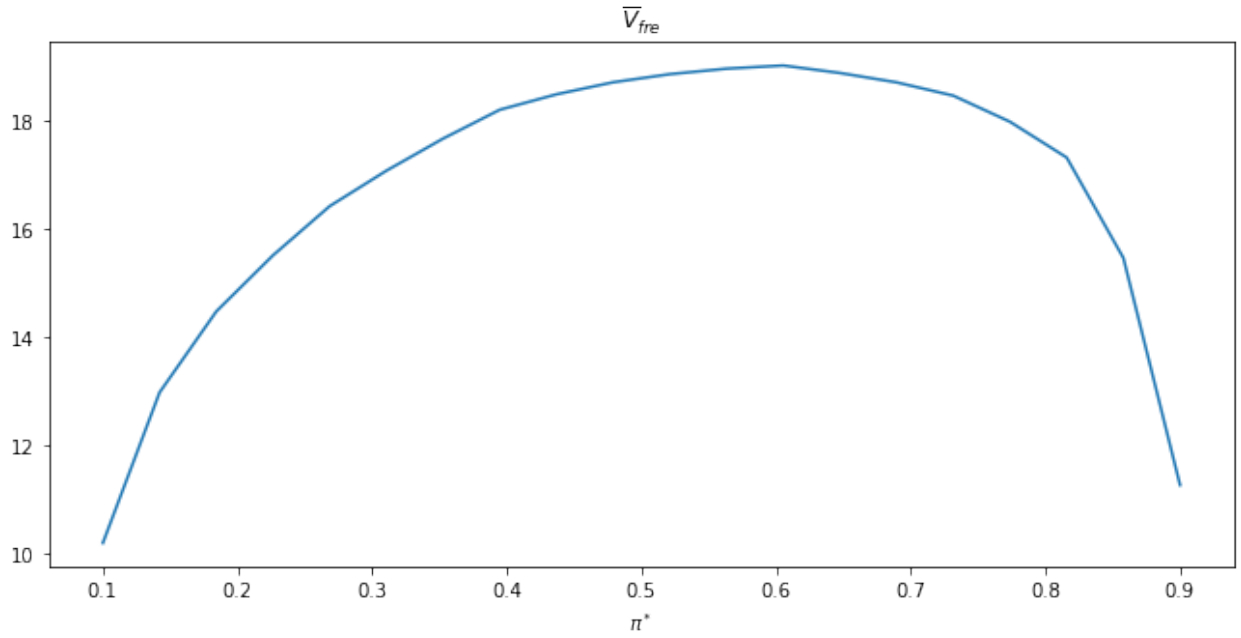
```
msg = f"The above graph indicates that minimizing over t tells the frequentist to␣
 ↪draw {t_optimal} observations and then decide."
print(msg)
```

```
The above graph indicates that minimizing over t tells the frequentist to draw 9␣
 ↪observations and then decide.
```

Let's now change the value of π^* and watch how the decision rule changes.

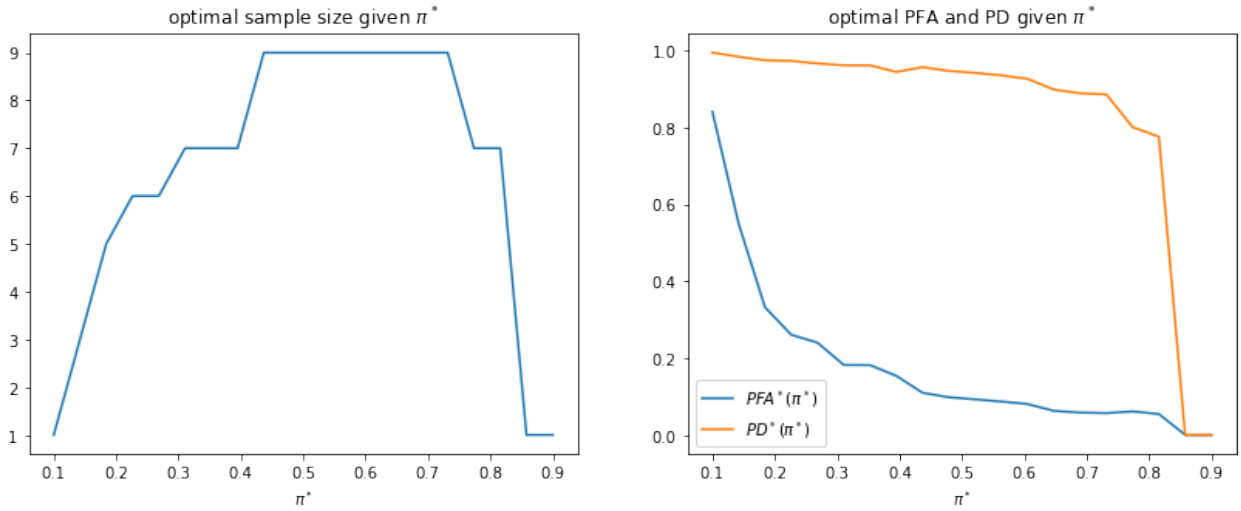
```
n_π = 20
π_star_arr = np.linspace(0.1, 0.9, n_π)

V_fre_bar_arr = np.empty(n_π)
t_optimal_arr = np.empty(n_π)
PFA_optimal_arr = np.empty(n_π)
PD_optimal_arr = np.empty(n_π)

for i, π_star in enumerate(π_star_arr):
    V_fre_arr, PFA_arr, PD_arr = compute_V_fre(L0_arr, L1_arr, π_star, wf)
    t_idx = np.argmin(V_fre_arr)

    V_fre_bar_arr[i] = V_fre_arr[t_idx]
    t_optimal_arr[i] = t_idx + 1
    PFA_optimal_arr[i] = PFA_arr[t_idx]
    PD_optimal_arr[i] = PD_arr[t_idx]
```

```
plt.plot(π_star_arr, V_fre_bar_arr)
plt.xlabel('$\pi^*$')
plt.title('$\overline{V}_{fre}$')

plt.show()
```



The following shows how optimal sample size t and targeted (PFA, PD) change as π^* varies.

```
fig, axs = plt.subplots(1, 2, figsize=(14, 5))

axs[0].plot(π_star_arr, t_optimal_arr)
axs[0].set_xlabel('$\pi^*$')
axs[0].set_title('optimal sample size given $\pi^*$')

axs[1].plot(π_star_arr, PFA_optimal_arr, label='$PFA^*(\pi^*)$')
axs[1].plot(π_star_arr, PD_optimal_arr, label='$PD^*(\pi^*)$')
axs[1].set_xlabel('$\pi^*$')
axs[1].legend()
axs[1].set_title('optimal PFA and PD given $\pi^*$')

plt.show()
```



47.4 Bayesian Decision Rule

In *A Problem that Stumped Milton Friedman*, we learned how Abraham Wald confirmed the Navy Captain's hunch that there is a better decision rule.

We presented a Bayesian procedure that instructed the Captain to makes decisions by comparing his current Bayesian posterior probability π with two cutoff probabilities called α and β.

To proceed, we borrow some Python code from the quantecon lecture *A Problem that Stumped Milton Friedman* that computes α and β.

```
@njit(parallel=True)
def Q(h, wf):

    c, π_grid = wf.c, wf.π_grid
    L0, L1 = wf.L0, wf.L1
    z0, z1 = wf.z0, wf.z1
    mc_size = wf.mc_size

    κ = wf.κ

    h_new = np.empty_like(π_grid)
    h_func = lambda p: interp(π_grid, h, p)

    for i in prange(len(π_grid)):
        π = π_grid[i]

        # Find the expected value of J by integrating over z
        integral_f0, integral_f1 = 0, 0
        for m in range(mc_size):
            π_0 = κ(z0[m], π)  # Draw z from f0 and update π
            integral_f0 += min((1 - π_0) * L0, π_0 * L1, h_func(π_0))

            π_1 = κ(z1[m], π)  # Draw z from f1 and update π
            integral_f1 += min((1 - π_1) * L0, π_1 * L1, h_func(π_1))

        integral = (π * integral_f0 + (1 - π) * integral_f1) / mc_size

        h_new[i] = c + integral

    return h_new
```

```
@njit
def solve_model(wf, tol=1e-4, max_iter=1000):
    """
    Compute the continuation value function

    * wf is an instance of WaldFriedman
    """

    # Set up loop
    h = np.zeros(len(wf.π_grid))
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        h_new = Q(h, wf)
```

(continues on next page)

(continued from previous page)

```
        error = np.max(np.abs(h - h_new))
        i += 1
        h = h_new

    if i == max_iter:
        print("Failed to converge!")

    return h_new
```

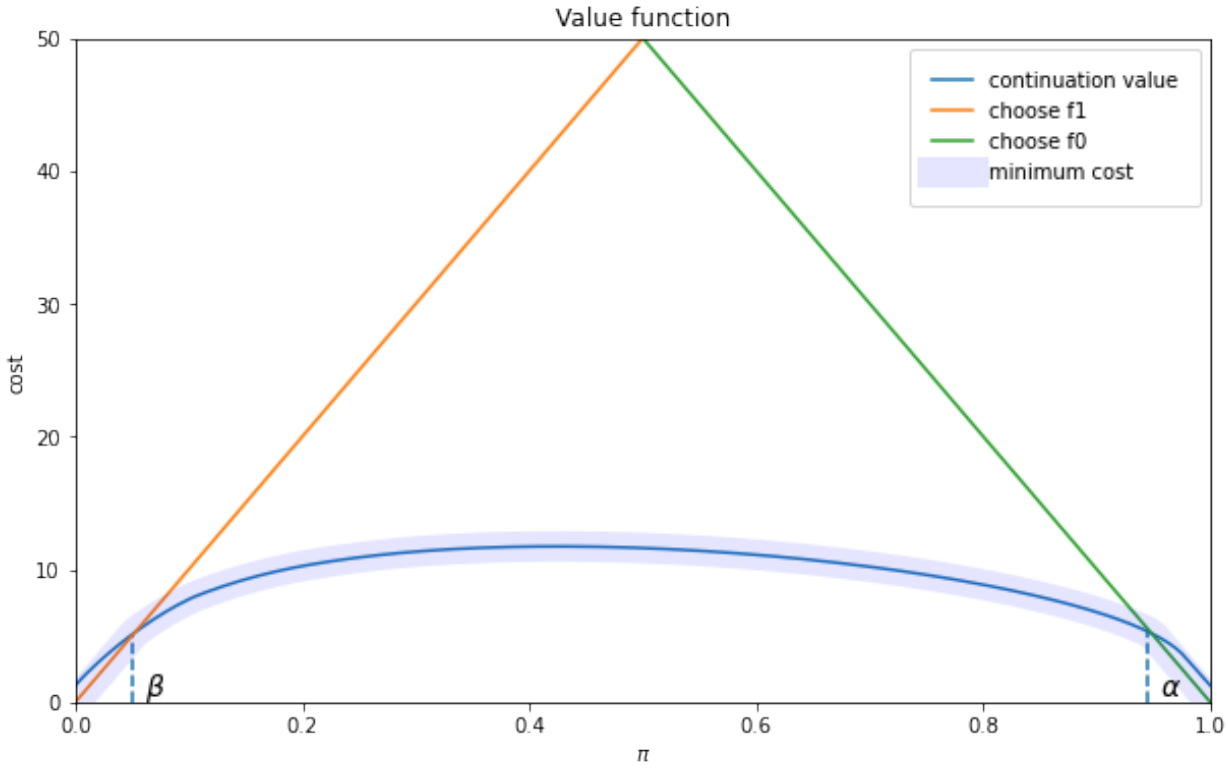
```
h_star = solve_model(wf)
```

```
@njit
def find_cutoff_rule(wf, h):

    """
    This function takes a continuation value function and returns the
    corresponding cutoffs of where you transition between continuing and
    choosing a specific model
    """

    π_grid = wf.π_grid
    L0, L1 = wf.L0, wf.L1

    # Evaluate cost at all points on grid for choosing a model
    payoff_f0 = (1 - π_grid) * L0
    payoff_f1 = π_grid * L1

    # The cutoff points can be found by differencing these costs with
    # The Bellman equation (J is always less than or equal to p_c_i)
    β = π_grid[np.searchsorted(
                              payoff_f1 - np.minimum(h, payoff_f0),
                              1e-10)
               - 1]
    α = π_grid[np.searchsorted(
                              np.minimum(h, payoff_f1) - payoff_f0,
                              1e-10)
               - 1]

    return (β, α)

β, α = find_cutoff_rule(wf, h_star)
cost_L0 = (1 - wf.π_grid) * wf.L0
cost_L1 = wf.π_grid * wf.L1

fig, ax = plt.subplots(figsize=(10, 6))

ax.plot(wf.π_grid, h_star, label='continuation value')
ax.plot(wf.π_grid, cost_L1, label='choose f1')
ax.plot(wf.π_grid, cost_L0, label='choose f0')
ax.plot(wf.π_grid,
        np.amin(np.column_stack([h_star, cost_L0, cost_L1]),axis=1),
        lw=15, alpha=0.1, color='b', label='minimum cost')

ax.annotate(r"$\beta$", xy=(β + 0.01, 0.5), fontsize=14)
ax.annotate(r"$\alpha$", xy=(α + 0.01, 0.5), fontsize=14)
```

(continues on next page)

(continued from previous page)

```
plt.vlines(β, 0, β * wf.L0, linestyle="--")
plt.vlines(α, 0, (1 - α) * wf.L1, linestyle="--")

ax.set(xlim=(0, 1), ylim=(0, 0.5 * max(wf.L0, wf.L1)), ylabel="cost",
       xlabel="$\pi$", title="Value function")

plt.legend(borderpad=1.1)
plt.show()
```



The above figure portrays the value function plotted against the decision maker's Bayesian posterior.

It also shows the probabilities α and β.

The Bayesian decision rule is:

- accept H_0 if $\pi \geq \alpha$
- accept H_1 if $\pi \leq \beta$
- delay deciding and draw another z if $\beta \leq \pi \leq \alpha$

We can calculate two "objective" loss functions under this situation conditioning on knowing for sure that nature has selected f_0, in the first case, or f_1, in the second case.

1. under f_0,

$$V^0(\pi) = \begin{cases} 0 & \text{if } \alpha \leq \pi, \\ c + EV^0(\pi') & \text{if } \beta \leq \pi < \alpha, \\ \bar{L}_1 & \text{if } \pi < \beta. \end{cases}$$

2. under f_1

$$V^1(\pi) = \begin{cases} \bar{L}_0 & \text{if } \alpha \leq \pi, \\ c + EV^1(\pi') & \text{if } \beta \leq \pi < \alpha, \\ 0 & \text{if } \pi < \beta. \end{cases}$$

where $\pi' = \frac{\pi f_0(z')}{\pi f_0(z') + (1-\pi) f_1(z')}$.

Given a prior probability π_0, the expected loss for the Bayesian is

$$\bar{V}_{Bayes}(\pi_0) = \pi^* V^0(\pi_0) + (1 - \pi^*) V^1(\pi_0).$$

Below we write some Python code that computes $V^0(\pi)$ and $V^1(\pi)$ numerically.

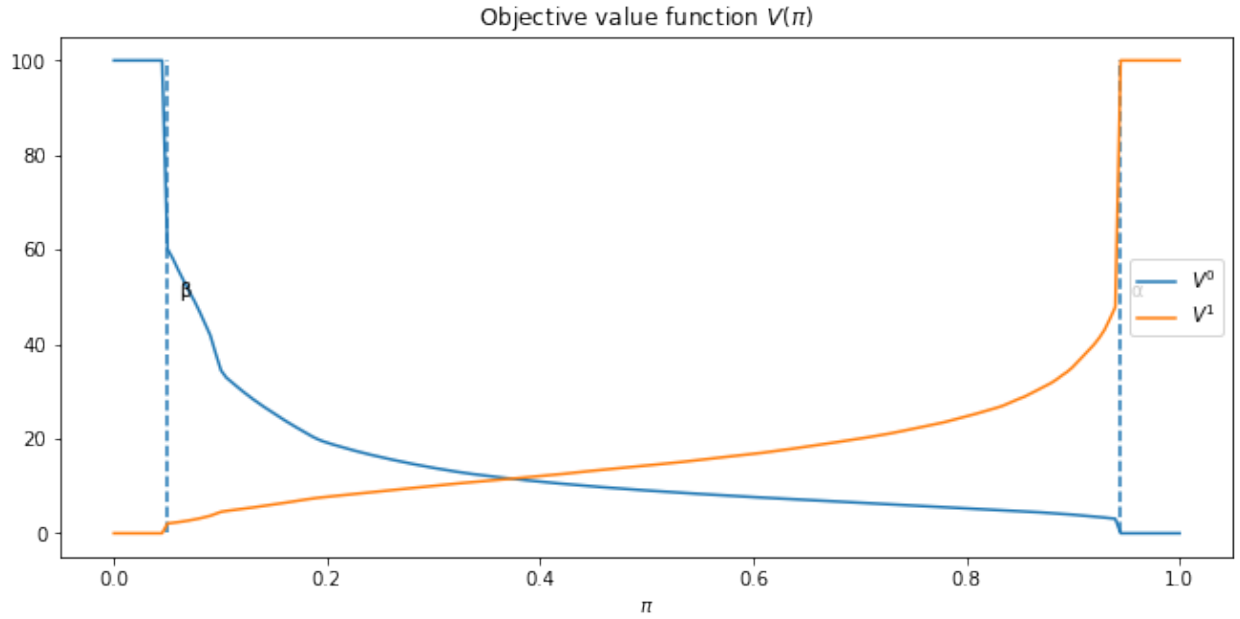
```
@njit(parallel=True)
def V_q(wf, flag):
    V = np.zeros(wf.π_grid_size)
    if flag == 0:
        z_arr = wf.z0
        V[wf.π_grid < β] = wf.L1
    else:
        z_arr = wf.z1
        V[wf.π_grid >= α] = wf.L0

    V_old = np.empty_like(V)

    while True:
        V_old[:] = V[:]
        V[(β <= wf.π_grid) & (wf.π_grid < α)] = 0

        for i in prange(len(wf.π_grid)):
            π = wf.π_grid[i]

            if π >= α or π < β:
                continue

            for j in prange(len(z_arr)):
                π_next = wf.κ(z_arr[j], π)
                V[i] += wf.c + interp(wf.π_grid, V_old, π_next)

            V[i] /= wf.mc_size

        if np.abs(V - V_old).max() < 1e-5:
            break

    return V
```

```
V0 = V_q(wf, 0)
V1 = V_q(wf, 1)

plt.plot(wf.π_grid, V0, label='$V^0$')
plt.plot(wf.π_grid, V1, label='$V^1$')
plt.vlines(β, 0, wf.L0, linestyle='--')
plt.text(β+0.01, wf.L0/2, 'β')
plt.vlines(α, 0, wf.L0, linestyle='--')
plt.text(α+0.01, wf.L0/2, 'α')
```

(continues on next page)

(continued from previous page)

```
plt.xlabel('$\pi$')
plt.title('Objective value function $V(\pi)$')
plt.legend()
plt.show()
```



Given an assumed value for $\pi^* = \Pr\{\text{nature selects } f_0\}$, we can then compute $\bar{V}_{Bayes}(\pi_0)$.

We can then determine an initial Bayesian prior π_0^* that minimizes this objective concept of expected loss.

The figure 9 below plots four cases corresponding to $\pi^* = 0.25, 0.3, 0.5, 0.7$.

We observe that in each case π_0^* equals π^*.

```
def compute_V_baye_bar(π_star, V0, V1, wf):

    V_baye = π_star * V0 + (1 - π_star) * V1
    π_idx = np.argmin(V_baye)
    π_optimal = wf.π_grid[π_idx]
    V_baye_bar = V_baye[π_idx]

    return V_baye, π_optimal, V_baye_bar
```

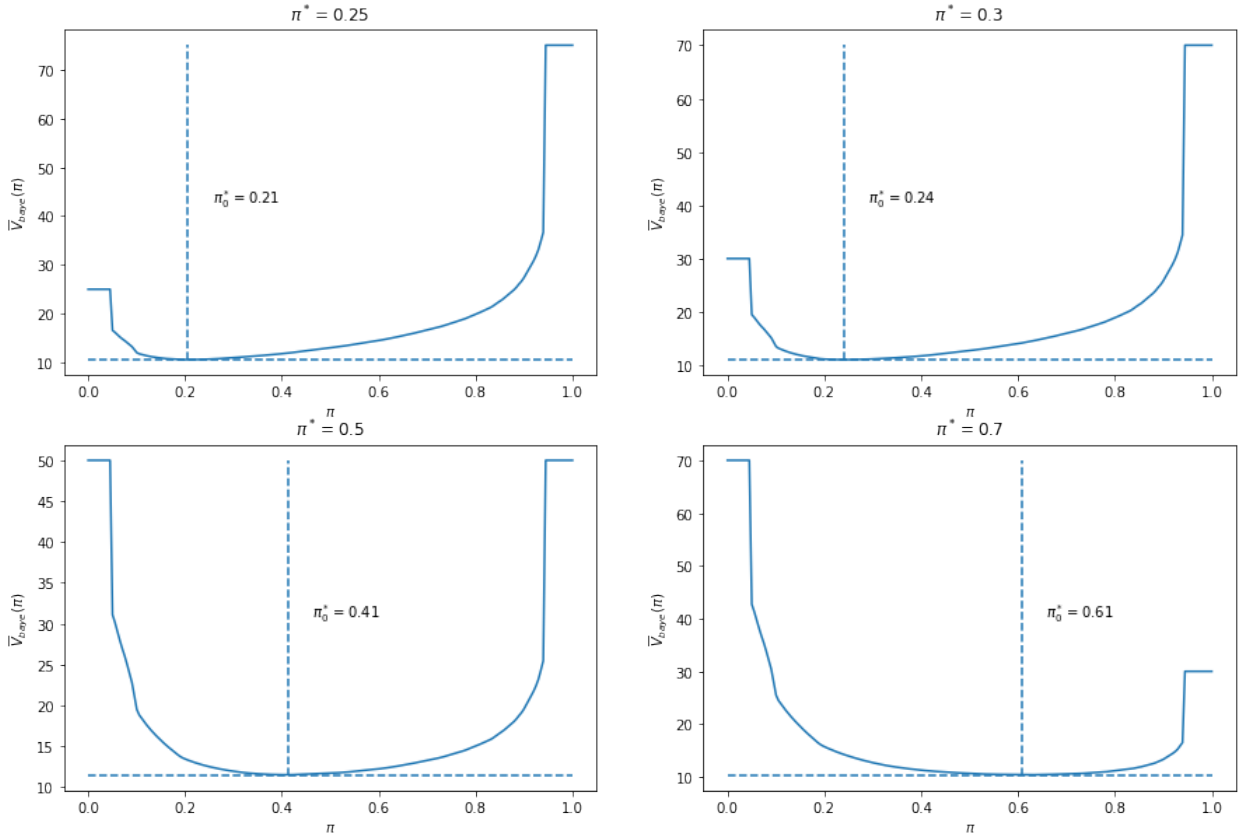
```
π_star_arr = [0.25, 0.3, 0.5, 0.7]

fig, axs = plt.subplots(2, 2, figsize=(15, 10))

for i, π_star in enumerate(π_star_arr):
    row_i = i // 2
    col_i = i % 2

    V_baye, π_optimal, V_baye_bar = compute_V_baye_bar(π_star, V0, V1, wf)

    axs[row_i, col_i].plot(wf.π_grid, V_baye)
    axs[row_i, col_i].hlines(V_baye_bar, 0, 1, linestyle='--')
    axs[row_i, col_i].vlines(π_optimal, V_baye_bar, V_baye.max(), linestyle='--')
```

(continues on next page)

(continued from previous page)

```
    axs[row_i, col_i].text(π_optimal+0.05, (V_baye_bar + V_baye.max()) / 2,
                           '${\pi_0^*}=$'+f'{π_optimal:0.2f}')
    axs[row_i, col_i].set_xlabel('$\pi$')
    axs[row_i, col_i].set_ylabel('$\overline{V}_{baye}(\pi)$')
    axs[row_i, col_i].set_title('$\pi^*=$' + f'{π_star}')

fig.suptitle('$\overline{V}_{baye}(\pi)=\pi^*V^0(\pi) + (1-\pi^*)V^1(\pi)$',␣
 ↪fontsize=16)
plt.show()
```

$$\overline{V}_{baye}(\pi) = \pi^* V^0(\pi) + (1 - \pi^*) V^1(\pi)$$



This pattern of outcomes holds more generally.

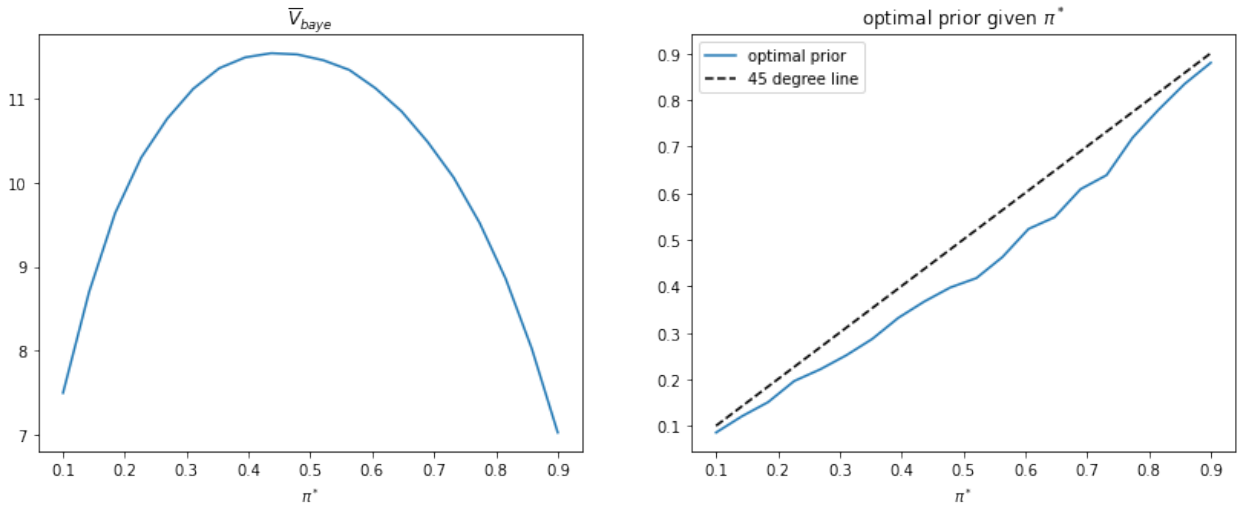
Thus, the following Python code generates the associated graph that verifies the equality of π_0^* to π^* holds for all π^*.

```
π_star_arr = np.linspace(0.1, 0.9, n_π)
V_baye_bar_arr = np.empty_like(π_star_arr)
π_optimal_arr = np.empty_like(π_star_arr)

for i, π_star in enumerate(π_star_arr):

    V_baye, π_optimal, V_baye_bar = compute_V_baye_bar(π_star, V0, V1, wf)

    V_baye_bar_arr[i] = V_baye_bar
    π_optimal_arr[i] = π_optimal
```

(continues on next page)

(continued from previous page)

```
fig, axs = plt.subplots(1, 2, figsize=(14, 5))

axs[0].plot(π_star_arr, V_baye_bar_arr)
axs[0].set_xlabel('$\pi^*$')
axs[0].set_title('$\overline{V}_{baye}$')

axs[1].plot(π_star_arr, π_optimal_arr, label='optimal prior')
axs[1].plot([π_star_arr.min(), π_star_arr.max()],
            [π_star_arr.min(), π_star_arr.max()],
            c='k', linestyle='--', label='45 degree line')
axs[1].set_xlabel('$\pi^*$')
axs[1].set_title('optimal prior given $\pi^*$')
axs[1].legend()

plt.show()
```



47.5 Was the Navy Captain's hunch correct?

We now compare average (i.e., frequentist) losses obtained by the frequentist and Bayesian decision rules.

As a starting point, let's compare average loss functions when $\pi^* = 0.5$.

```
π_star = 0.5
```

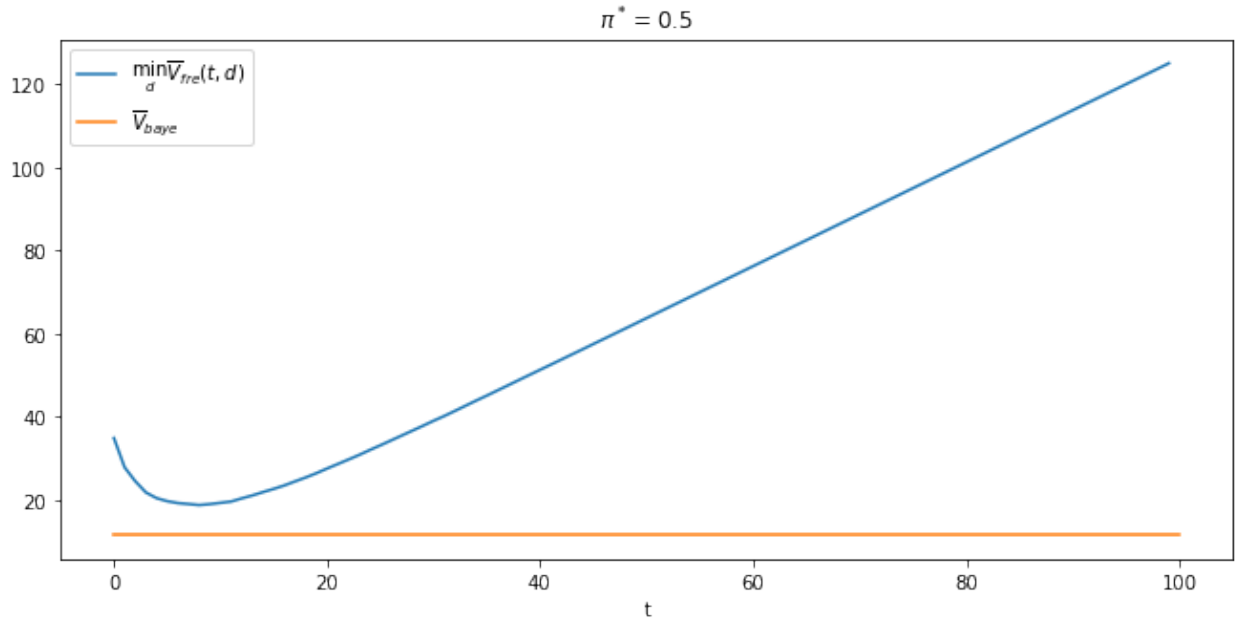
```
# frequentist
V_fre_arr, PFA_arr, PD_arr = compute_V_fre(L0_arr, L1_arr, π_star, wf)

# bayesian
V_baye = π_star * V0 + π_star * V1
V_baye_bar = V_baye.min()
```

```
plt.plot(range(T), V_fre_arr, label='$\min_{d} \overline{V}_{fre}(t,d)$')
plt.plot([0, T], [V_baye_bar, V_baye_bar], label='$\overline{V}_{baye}$')
plt.xlabel('t')
plt.title('$\pi^*=0.5$')
```

(continues on next page)

(continued from previous page)

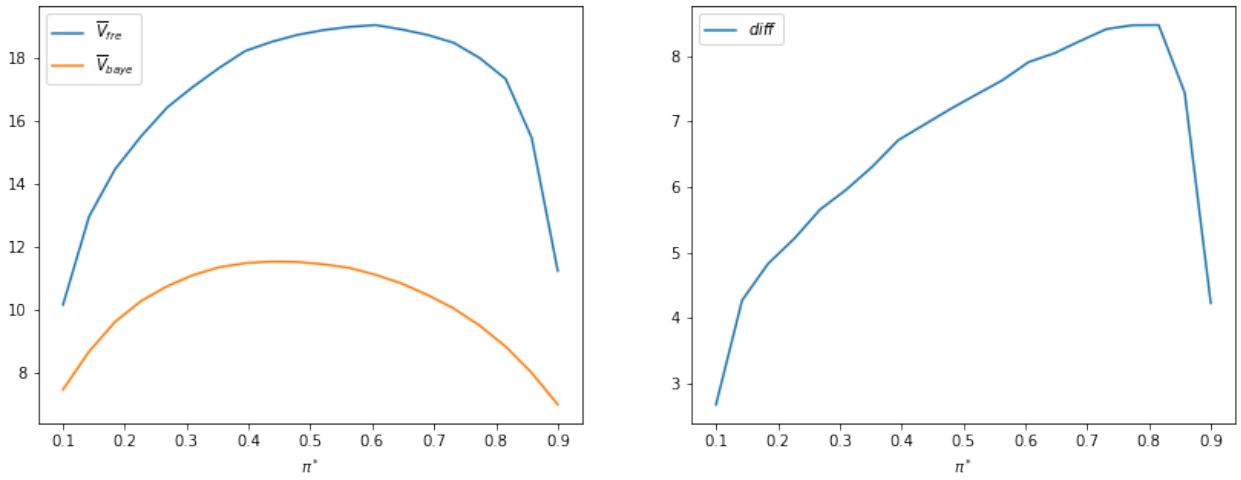
```
plt.legend()
plt.show()
```



Evidently, there is no sample size t at which the frequentist decision rule attains a lower loss function than does the Bayesian rule.

Furthermore, the following graph indicates that the Bayesian decision rule does better on average for all values of π^*.

```
fig, axs = plt.subplots(1, 2, figsize=(14, 5))

axs[0].plot(π_star_arr, V_fre_bar_arr, label='$\overline{V}_{fre}$')
axs[0].plot(π_star_arr, V_baye_bar_arr, label='$\overline{V}_{baye}$')
axs[0].legend()
axs[0].set_xlabel('$\pi^*$')

axs[1].plot(π_star_arr, V_fre_bar_arr - V_baye_bar_arr, label='$diff$')
axs[1].legend()
axs[1].set_xlabel('$\pi^*$')

plt.show()
```



The right panel of the above graph plots the difference $\bar{V}_{fre} - \bar{V}_{Bayes}$.

It is always positive.

47.6 More details

We can provide more insights by focusing on the case in which $\pi^* = 0.5 = \pi_0$.

```
π_star = 0.5
```

Recall that when $\pi^* = 0.5$, the frequentist decision rule sets a sample size `t_optimal` **ex ante**.

For our parameter settings, we can compute its value:

```
t_optimal
```

```
9
```

For convenience, let's define `t_idx` as the Python array index corresponding to `t_optimal` sample size.

```
t_idx = t_optimal - 1
```

47.6.1 Distribution of Bayesian decision rule's times to decide

By using simulations, we compute the frequency distribution of time to deciding for the Bayesian decision rule and compare that time to the frequentist rule's fixed t.

The following Python code creates a graph that shows the frequency distribution of Bayesian times to decide of Bayesian decision maker, conditional on distribution $q = f_0$ or $q = f_1$ generating the data.

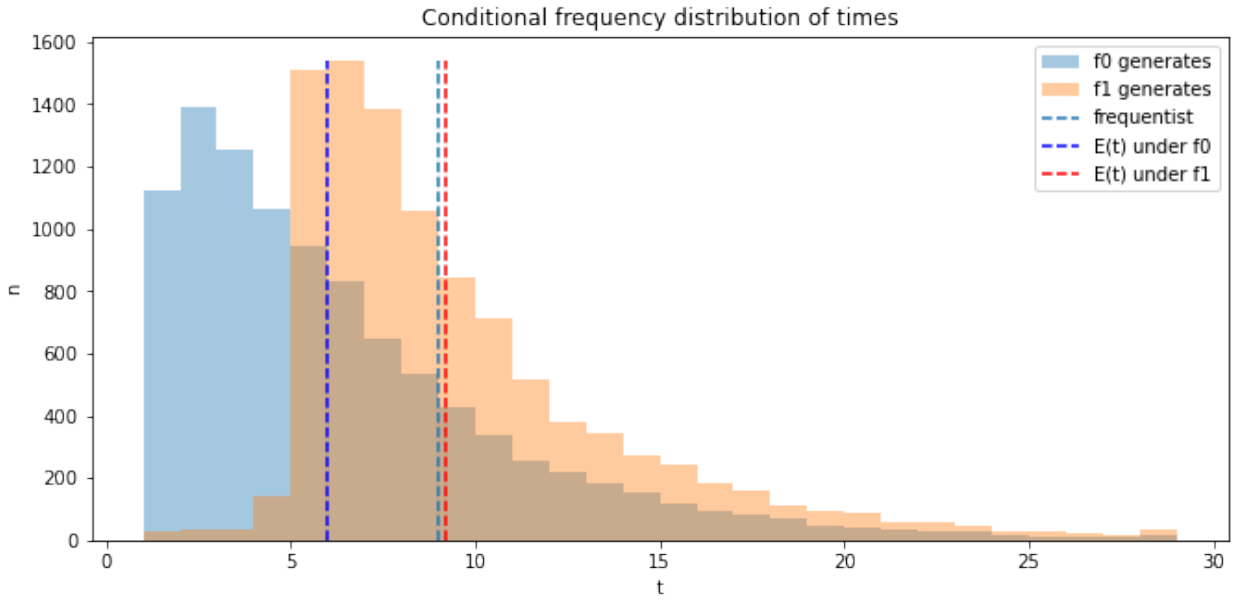
The blue and red dotted lines show averages for the Bayesian decision rule, while the black dotted line shows the frequentist optimal sample size t.

On average the Bayesian rule decides **earlier** than the frequentist rule when $q = f_0$ and **later** when $q = f_1$.

```
@njit(parallel=True)
def check_results(L_arr, α, β, flag, π0):

    N, T = L_arr.shape

    time_arr = np.empty(N)
    correctness = np.empty(N)

    π_arr = π0 * L_arr / (π0 * L_arr + 1 - π0)

    for i in prange(N):
        for t in range(T):
            if (π_arr[i, t] < β) or (π_arr[i, t] > α):
                time_arr[i] = t + 1
                correctness[i] = (flag == 0 and π_arr[i, t] > α) or (flag == 1 and π_
↪arr[i, t] < β)
                break

    return time_arr, correctness
```

```
time_arr0, correctness0 = check_results(L0_arr, α, β, 0, π_star)
time_arr1, correctness1 = check_results(L1_arr, α, β, 1, π_star)

# unconditional distribution
time_arr_u = np.concatenate((time_arr0, time_arr1))
correctness_u = np.concatenate((correctness0, correctness1))
```

```
n1 = plt.hist(time_arr0, bins=range(1, 30), alpha=0.4, label='f0 generates')[0]
n2 = plt.hist(time_arr1, bins=range(1, 30), alpha=0.4, label='f1 generates')[0]
plt.vlines(t_optimal, 0, max(n1.max(), n2.max()), linestyle='--', label='frequentist')
plt.vlines(np.mean(time_arr0), 0, max(n1.max(), n2.max()),
           linestyle='--', color='b', label='E(t) under f0')
plt.vlines(np.mean(time_arr1), 0, max(n1.max(), n2.max()),
           linestyle='--', color='r', label='E(t) under f1')
plt.legend();

plt.xlabel('t')
plt.ylabel('n')
plt.title('Conditional frequency distribution of times')

plt.show()
```



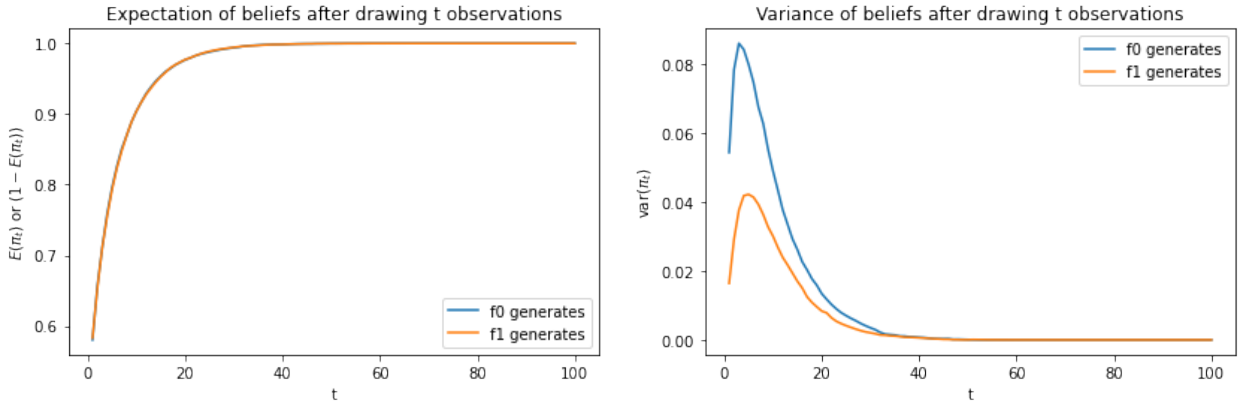
Later we'll figure out how these distributions ultimately affect objective expected values under the two decision rules.

To begin, let's look at simulations of the Bayesian's beliefs over time.

We can easily compute the updated beliefs at any time t using the one-to-one mapping from L_t to π_t given π_0 described in this lecture *Likelihood Ratio Processes*.

```
π0_arr = π_star * L0_arr / (π_star * L0_arr + 1 - π_star)
π1_arr = π_star * L1_arr / (π_star * L1_arr + 1 - π_star)
```

```
fig, axs = plt.subplots(1, 2, figsize=(14, 4))

axs[0].plot(np.arange(1, π0_arr.shape[1]+1), np.mean(π0_arr, 0), label='f0 generates')
axs[0].plot(np.arange(1, π1_arr.shape[1]+1), 1 - np.mean(π1_arr, 0), label='f1␣
↪generates')
axs[0].set_xlabel('t')
axs[0].set_ylabel('$E(\pi_t)$ or ($1 - E(\pi_t)$)')
axs[0].set_title('Expectation of beliefs after drawing t observations')
axs[0].legend()

axs[1].plot(np.arange(1, π0_arr.shape[1]+1), np.var(π0_arr, 0), label='f0 generates')
axs[1].plot(np.arange(1, π1_arr.shape[1]+1), np.var(π1_arr, 0), label='f1 generates')
axs[1].set_xlabel('t')
axs[1].set_ylabel('var($\pi_t$)')
axs[1].set_title('Variance of beliefs after drawing t observations')
axs[1].legend()

plt.show()
```



The above figures compare averages and variances of updated Bayesian posteriors after t draws.

The left graph compares $E(\pi_t)$ under f_0 to $1 - E(\pi_t)$ under f_1: they lie on top of each other.

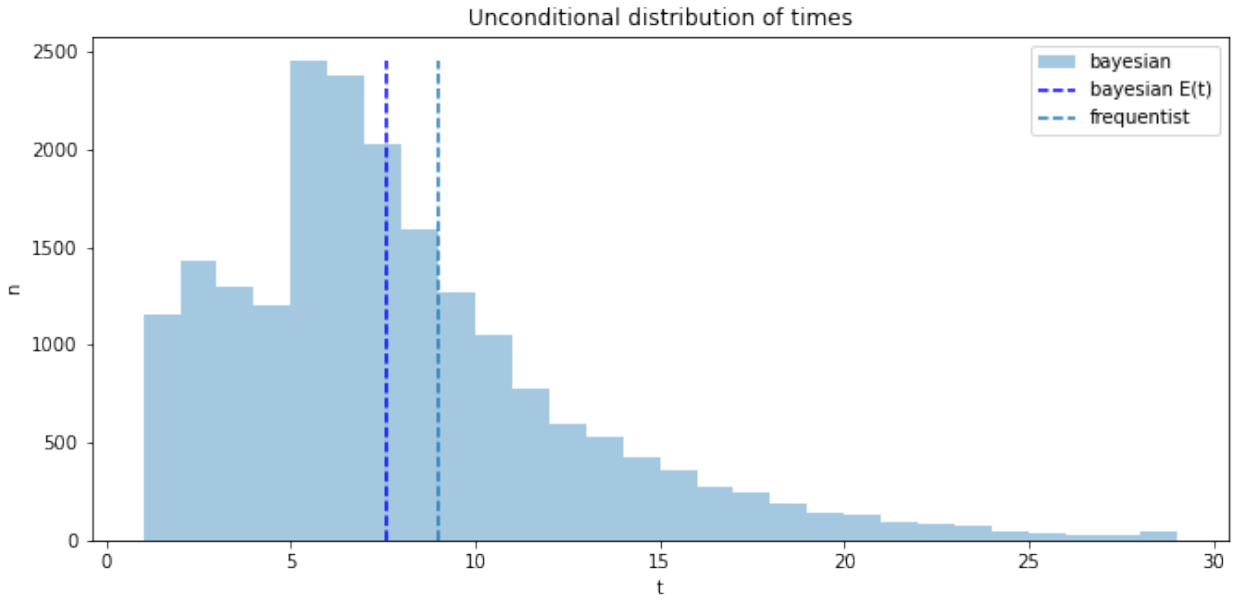
However, as the right hand size graph shows, there is significant difference in variances when t is small: the variance is lower under f_1.

The difference in variances is the reason that the Bayesian decision maker waits longer to decide when f_1 generates the data.

The code below plots outcomes of constructing an unconditional distribution by simply pooling the simulated data across the two possible distributions f_0 and f_1.

The pooled distribution describes a sense in which on average the Bayesian decides earlier, an outcome that seems at least partly to confirm the Navy Captain's hunch.

```
n = plt.hist(time_arr_u, bins=range(1, 30), alpha=0.4, label='bayesian')[0]
plt.vlines(np.mean(time_arr_u), 0, n.max(), linestyle='--',
           color='b', label='bayesian E(t)')
plt.vlines(t_optimal, 0, n.max(), linestyle='--', label='frequentist')
plt.legend()

plt.xlabel('t')
plt.ylabel('n')
plt.title('Unconditional distribution of times')

plt.show()
```



47.6.2 Probability of making correct decisions

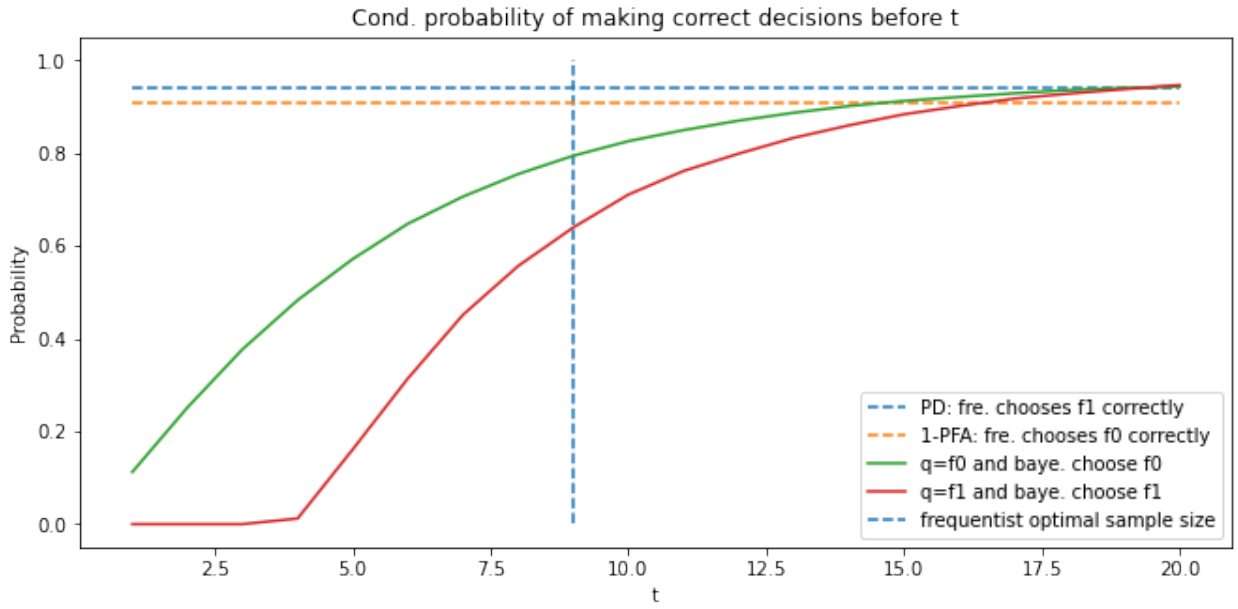
Now we use simulations to compute the fraction of samples in which the Bayesian and the frequentist decision rules decide correctly.

For the frequentist rule, the probability of making the correct decision under f_1 is the optimal probability of detection given t that we defined earlier, and similarly it equals 1 minus the optimal probability of a false alarm under f_0.

Below we plot these two probabilities for the frequentist rule, along with the conditional probabilities that the Bayesian rule decides before t *and* that the decision is correct.

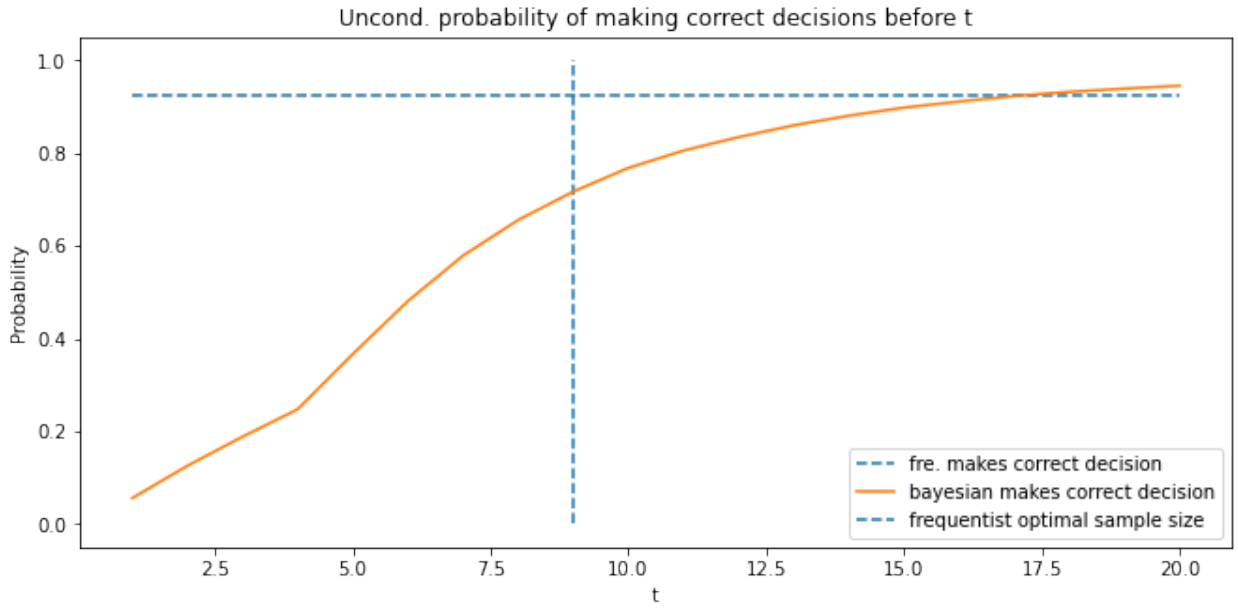
```
# optimal PFA and PD of frequentist with optimal sample size
V, PFA, PD = V_fre_t(t_optimal, L0_arr, L1_arr, π_star, wf)
```

```
plt.plot([1, 20], [PD, PD], linestyle='--', label='PD: fre. chooses f1 correctly')
plt.plot([1, 20], [1-PFA, 1-PFA], linestyle='--', label='1-PFA: fre. chooses f0␣
↪correctly')
plt.vlines(t_optimal, 0, 1, linestyle='--', label='frequentist optimal sample size')

N = time_arr0.size
T_arr = np.arange(1, 21)
plt.plot(T_arr, [np.sum(correctness0[time_arr0 <= t] == 1) / N for t in T_arr],
         label='q=f0 and baye. choose f0')
plt.plot(T_arr, [np.sum(correctness1[time_arr1 <= t] == 1) / N for t in T_arr],
         label='q=f1 and baye. choose f1')
plt.legend(loc=4)

plt.xlabel('t')
plt.ylabel('Probability')
plt.title('Cond. probability of making correct decisions before t')

plt.show()
```



By averaging using π^*, we also plot the unconditional distribution.

```
plt.plot([1, 20], [(PD + 1 - PFA) / 2, (PD + 1 - PFA) / 2],
         linestyle='--', label='fre. makes correct decision')
plt.vlines(t_optimal, 0, 1, linestyle='--', label='frequentist optimal sample size')

N = time_arr_u.size
plt.plot(T_arr, [np.sum(correctness_u[time_arr_u <= t] == 1) / N for t in T_arr],
         label="bayesian makes correct decision")
plt.legend()

plt.xlabel('t')
plt.ylabel('Probability')
plt.title('Uncond. probability of making correct decisions before t')

plt.show()
```



47.6.3 Distribution of likelihood ratios at frequentist's t

Next we use simulations to construct distributions of likelihood ratios after t draws.

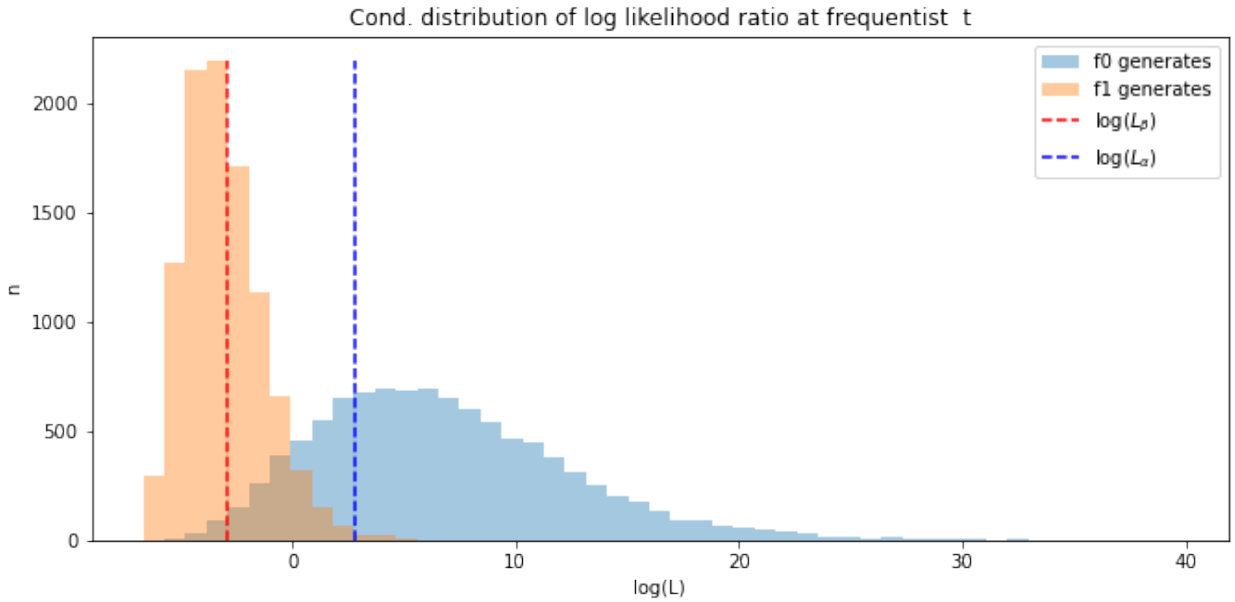
To serve as useful reference points, we also show likelihood ratios that correspond to the Bayesian cutoffs α and β.

In order to exhibit the distribution more clearly, we report logarithms of likelihood ratios.

The graphs below reports two distributions, one conditional on f_0 generating the data, the other conditional on f_1 generating the data.

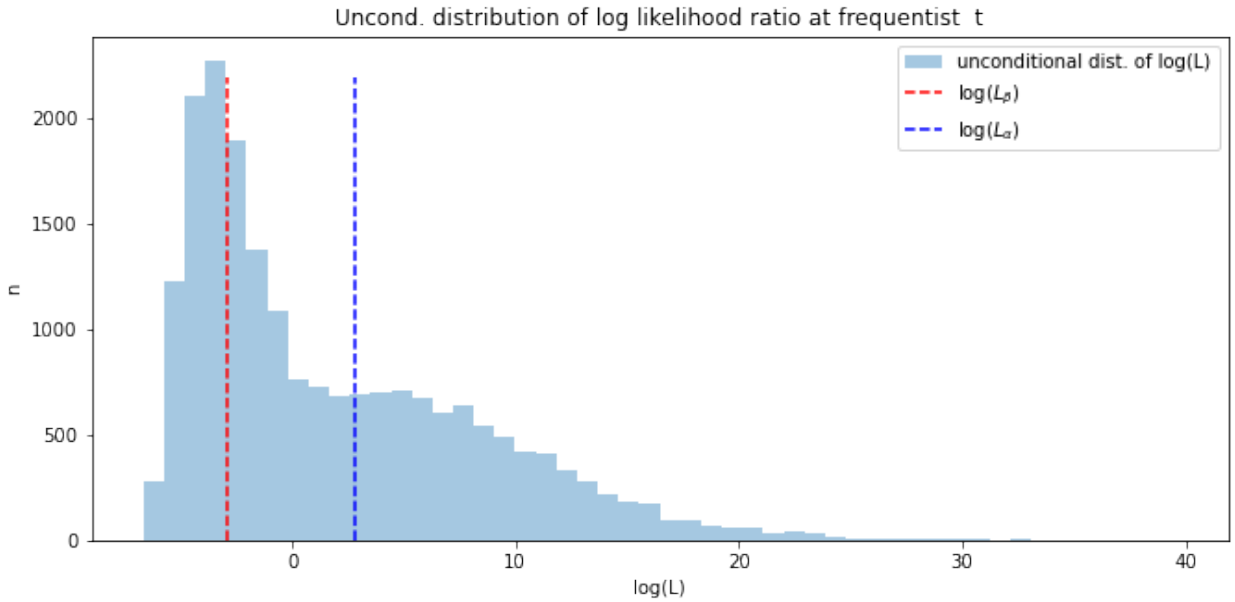
```
Lα = (1 - π_star) *  α / (π_star - π_star * α)
Lβ = (1 - π_star) *  β / (π_star - π_star * β)
```

```
L_min = min(L0_arr[:, t_idx].min(), L1_arr[:, t_idx].min())
L_max = max(L0_arr[:, t_idx].max(), L1_arr[:, t_idx].max())
bin_range = np.linspace(np.log(L_min), np.log(L_max), 50)
n0 = plt.hist(np.log(L0_arr[:, t_idx]), bins=bin_range, alpha=0.4, label='f0 generates
↪')[0]
n1 = plt.hist(np.log(L1_arr[:, t_idx]), bins=bin_range, alpha=0.4, label='f1 generates
↪')[0]

plt.vlines(np.log(Lβ), 0, max(n0.max(), n1.max()), linestyle='--', color='r', label=
↪'log($L_β$)')
plt.vlines(np.log(Lα), 0, max(n0.max(), n1.max()), linestyle='--', color='b', label=
↪'log($L_α$)')
plt.legend()

plt.xlabel('log(L)')
plt.ylabel('n')
plt.title('Cond. distribution of log likelihood ratio at frequentist  t')

plt.show()
```



The next graph plots the unconditional distribution of Bayesian times to decide, constructed as earlier by pooling the two conditional distributions.

```
plt.hist(np.log(np.concatenate([L0_arr[:, t_idx], L1_arr[:, t_idx]])),
         bins=50, alpha=0.4, label='unconditional dist. of log(L)')
plt.vlines(np.log(Lβ), 0, max(n0.max(), n1.max()), linestyle='--', color='r', label=
 ↪'log($L_β$)')
plt.vlines(np.log(Lα), 0, max(n0.max(), n1.max()), linestyle='--', color='b', label=
 ↪'log($L_α$)')
plt.legend()

plt.xlabel('log(L)')
plt.ylabel('n')
plt.title('Uncond. distribution of log likelihood ratio at frequentist  t')

plt.show()
```


Uncond. distribution of log likelihood ratio at frequentist t
unconditional dist. of log(L)
log(L_β)
log(L_α)
n
2000
1500
1000
500
0
0
10
20
30
40
log(L)