Job Search V: Search with Learning

Thomas J. Sargent and John Stachurski

March 15, 2020

1 Contents

- Overview 2
- Model 3
- Take 1: Solution by VFI 4
- Take 2: A More Efficient Method 5
- Another Functional Equation 6
- Solving the RWFE 7
- Implementation 8
- Exercises 9
- Solutions 10
- Appendix A 11
- Appendix B 12
- Examples 13

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

In [1]:
   !pip install --upgrade quantecon
   !pip install interpolation

2 Overview

In this lecture, we consider an extension of the previously studied job search model of McCall [2].

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 [1], section 6.6.

Let’s start with some imports

In [2]:
```python
from numba import njit, prange, vectorize
from interpolation import mlinterp, interp
from math import gamma
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from matplotlib import cm
import scipy.optimize as op
from scipy.stats import cumfreq, beta
```

### 2.1 Model Features

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

### 3 Model

Let’s first review the basic McCall model [2] and then add the variation we want to consider.

#### 3.1 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

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

The optimal policy has the form \( 1\{w \geq \bar{w} \} \), where \( \bar{w} \) is a constant called the reservation wage.
3.2 Offer Distribution Unknown

Now let’s extend the model by considering the variation presented in [1], 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 \( \pi_0 \) on \( f \) being chosen.

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

\[
\pi_{t+1} = \frac{\pi_t f(w_{t+1})}{\pi_t f(w_{t+1}) + (1 - \pi_t)g(w_{t+1})}
\]

(2)

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

\[
P\{q = f \mid W = w\} = \frac{P\{W = w \mid q = f\}P\{q = f\}}{P\{W = w\}} \quad \text{and} \quad P\{W = w\} = \sum_{\omega \in \{f, g\}} P\{W = w \mid q = \omega\}P\{q = \omega\}
\]

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

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

(3)

Notice that the current guess \( \pi \) is a state variable, since it affects the worker’s perception of probabilities for future rewards.

3.3 Parameterization

Following section 6.6 of [1], 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
In [3]: @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)
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()

3.4 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 $\pi$ — in fact, it should be decreasing in $\pi$ because
• $f$ is a less attractive offer distribution than $g$
• larger $\pi$ means more weight on $f$ and less on $g$

Thus larger $\pi$ 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 $1w \geq \bar{w}(\pi)$ for some decreasing function $\bar{w}$.

4 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.

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.
In [5]: 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

    @njit
def ξ(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_func = lambda x, y: mlinterp((w_grid, π_grid), v, (x, y))
        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, 0.0
                for m in prange(mc_size):
                    integral_f += v_func(w_f[m], ξ(w_f[m], π))
                    integral_g += v_func(w_g[m], ξ(w_g[m], π))
                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.
        """
        v_func = lambda x, y: mlinterp((w_grid, π_grid), v, (x, y))
        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, 0.0
for m in prange(mc_size):
    integral_f += v_func(w_f[m], x(w_f[m], π))
    integral_g += v_func(w_g[m], x(w_g[m], π))
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

In [6]:
code

```python
# Serves for the \textit{value function}

* \textit{sp} is an instance of SearchProblem

```

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

    ""
    Solves for the value function
    ""

    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("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.\n')

return v_new

Let’s look at solutions computed from value function iteration

In [7]: sp = SearchProblem()
v_star = solve_model(sp)
fig, ax = plt.subplots(figsize=(6, 6))
ax.contourf(sp.pi_grid, sp.w_grid, v_star, 12, alpha=0.6, cmap=cm.jet)
cs = ax.contour(sp.pi_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 0.630835107428112.
Error at iteration 10 is 0.0943551196172443.
Error at iteration 15 is 0.017226580356112464.
Error at iteration 20 is 0.0034139125548815485.
Error at iteration 25 is 0.0006768911350825135.
Error at iteration 30 is 0.0001342193644440215.

Converged in 31 iterations.
We will also plot the optimal policy

In [8]: 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.

5 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.

6 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:

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

(4)

Together, (3) and (4) give

\[
v(w, \pi) = \max \left\{ \frac{w}{1 - \beta}, \frac{\bar{w}(\pi)}{1 - \beta} \right\}
\]

(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

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

(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.

### 7 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

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

(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

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

(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 $\pi$ now gives us

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

In other words, $Q$ is a contraction of modulus $\beta$ 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 $\omega \in b[0, 1]$.

## 8 Implementation

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

```python
In [9]: def Q_factory(sp, parallel_flag=True):
    f, g = sp.f, sp.g
    w_f, w_g = sp.w_f, sp.w_g
    beta, c = sp.beta, sp.c
    mc_size = sp.mc_size
    w_grid, pi_grid = sp.w_grid, sp.pi_grid

    @njit(parallel=parallel_flag)
    def kappa(w, pi):
        """Updates $\pi$ using Bayes' rule and the current wage observation $w$. """
        pf, pg = pi * f(w), (1 - pi) * g(w)
        pi_new = pf / (pf + pg)
        return pi_new

    @njit(parallel=parallel_flag)
    def Q(omega):
        """Updates the reservation wage function guess $\omega$ via the operator $Q$. """
        omega_func = lambda p: interp(pi_grid, omega, p)
        omega_new = np.empty_like(omega)
```
for i in prange(len(π_grid)):
    π = π_grid[i]
    integral_f, integral_g = 0.0, 0.0

for m in prange(mc_size):
    integral_f += max(w_f[m], ω_func(x(w_f[m], π)))
    integral_g += max(w_g[m], ω_func(x(w_g[m], π)))

integral = (π * integral_f + (1 - π) * integral_g) / mc_size

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 \( \bar{w} \).

### 9 Exercises

#### 9.1 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.

### 10 Solutions

#### 10.1 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

In [10]:
```python
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}.")
    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

In [11]: 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='$\omega$')
   ax.grid()
   plt.show()
11 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

In [12]: 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))

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

```python
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.ones((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
                p = np.random.uniform(0, 1)
                if p <= s:  # Randomly separate with
                    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:]
```

```python
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')
```
12 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 \( \pi \) at the time of employment.

In [13]: @njit
def empirical_dist(F_a, F_b, G_a, G_b, w_bar, \( \pi \) _grid, N=10000, T=600):
    
    """
    Simulates population for computing empirical cumulative distribution of unemployment duration and \( \pi \) 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
    ----------
    """
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
-------
accept_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 differ-
ent 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.

In [14]: 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 \( \pi \), and the probability of accepting an offer at different \( \pi \) level, and simulate and calculate the empirical cumulative distribution of the duration of unemployment and \( \pi \) 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
    \( \pi \)_grid = sp.\( \pi \)_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 subplots 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='\( l(w)\)', 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.\( \pi \)_grid, \( w\)_bar, color='k')
    axs[0, 1].fill_between(sp.\( \pi \)_grid, 0, \( w\)_bar, color='blue', alpha=0.15)
    axs[0, 1].fill_between(sp.\( \pi \)_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)
\[ II = \text{np.arange(0.01, 0.99, 0.08)} \]

\[ \Delta W = \text{np.zeros((len(W), len(II)))} \]
\[ \Delta II = \text{np.empty((len(W), len(II)))} \]

for \( i, w \) in enumerate(W):
    for \( j, \pi \) in enumerate(II):
        \( \Delta II[i, j] = \pi * (\Delta W / (\pi \cdot \Delta W + 1 - \pi) - 1) \)

\[ q = \text{axs[0, 1].quiver(II, W, \Delta II, \Delta W, scale=2, color='r', alpha=0.8)} \]

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

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

\[ \text{plt.show() \quad # part 2: simulate empirical cumulative distribution} \]

\[ \text{accept_t, accept_\pi = empirical_dist(F_a, F_b, G_a, G_b, w_bar, \pi\_grid)} \]
\[ \text{N = accept_t.shape[1]} \]

\[ \text{cfq\_t\_F = cumfreq(accept_t[0, :], numbins=100)} \]
\[ \text{cfq\_t\_F = cumfreq(accept_\pi[0, :], numbins=100)} \]

\[ \text{cfq\_t\_G = cumfreq(accept_t[1, :], numbins=100)} \]
\[ \text{cfq\_t\_G = cumfreq(accept_\pi[1, :], numbins=100)} \]

\[ \text{fig, axs = plt.subplots(2, 1, figsize=(12, 9))} \]

\[ \text{axs[0].plot(cumfreq_x(cfq\_t\_F), cfq\_t\_F\_cumcount/N, label='f\_\pi')} \]
\[ \text{axs[0].plot(cumfreq_x(cfq\_t\_G), cfq\_t\_G\_cumcount/N, label='g\_\pi')} \]

\[ \text{axs[0].grid(linestyle='--')} \]
\[ \text{axs[0].legend(loc=4)} \]
\[ \text{axs[0].title.set_text('CDF of duration of unemployment')} \]
\[ \text{axs[0].set(xlabel='time', ylabel='Prob(time)')} \]

\[ \text{axs[1].plot(cumfreq_x(cfq\_\pi\_F), cfq\_\pi\_F\_cumcount/N, label='f\_\pi')} \]
\[ \text{axs[1].plot(cumfreq_x(cfq\_\pi\_G), cfq\_\pi\_G\_cumcount/N, label='g\_\pi')} \]

\[ \text{axs[1].grid(linestyle='--')} \]
We now provide some examples that provide insights about how the model works.

13 Examples

13.1 Example 1 (Baseline)

\( F \sim \text{Beta}(1, 1), G \sim \text{Beta}(3, 1.2), c=0.3. \)

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

Recall the following formula from this lecture

\[
\frac{\pi_{t+1}}{\pi_t} = \frac{l(w_{t+1})}{\pi_tl(w_{t+1}) + (1-\pi_t)} \begin{cases} > 1 & \text{if } l(w_{t+1}) > 1 \\ \leq 1 & \text{if } l(w_{t+1}) \leq 1 \end{cases}
\]

The formula implies that the direction of motion of \( \pi_t \) is determined by the relationship between \( l(w) \) 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,
- \( \pi_{t-1} \) is close to either 0 or 1, which means the prior 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 \( \pi \) 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 \( \pi \) 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 \( \pi \) 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 \( \pi \) 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.
In [15]: job_search_example()
13.2 Example 2

\[ F \sim \text{Beta}(1, 1), \ G \sim \text{Beta}(1.2, 1.2), \ c=0.3. \]

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 \( \bar{w}(\pi) \) is increasing in \( \pi \).

In [16]: job_search_example(1, 1, 1.2, 1.2, 0.3)
13.3 Example 3

\[ F \sim \text{Beta}(1, 1), \ G \sim \text{Beta}(2, 2), \ c=0.3. \]

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

In [17]: job_search_example(1, 1, 2, 2, 0.3)
CDF of duration of unemployment

CDF of $\tau$ at time worker accepts wage and leaves unemployment

$F_{\tau}(\tau)$ for $f$ generates and $g$ generates
13.4 Example 4

$F \sim \text{Beta}(1, 1)$, $G \sim \text{Beta}(3, 1.2)$, 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 $\pi$ 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.

In [18]: job_search_example(1, 1, 3, 1.2, c=0.8)
13.5 Example 5

\( F \sim \text{Beta}(1, 1), \ G \sim \text{Beta}(3, 1.2), \) 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.

In [19]: job_search_example(1, 1, 3, 1.2, c=0.1)
References
