Assignment 2

April 10, 2023

0.1 Rust (1987) Replication

0.1.1 a. Estimate the replacement cost RC and the cost function parameter θ_{11} by maximum likelihood using the nested fixed-point method for computing the value functions. (More specifically, follow the Rust (1987) algorithm to replicate/find close numbers to the estimates for Group 4 buses in Table IX.) No need to compute standard errors.

[1]:
```
import numpy as np
import pandas as pd
from scipy.optimize import minimize
```

[2]:
```
df = pd.read_csv("group_4.csv")
df.head()
```

[2]:

	Bus_ID	period	state	mileage	decision
0	5297	1	1	6299	0
1	5297	2	2	10479	0
2	5297	3	3	15201	0
3	5297	4	4	20326	0
4	5297	5	4	24898	0

[3]:
```
class Rust:
    def __init__(self, beta=0.9999, theta_1=3.6, RC=10, T=90, p=0.3919, q=0.
↪5953, scale=0.001):

        """
        Initializes an instance of Rust class with the given arguments.

        Parameters:
        - beta: float, default 0.9999
        - theta_1: float, default 3.6
        - RC: int, default 10
        - T: int, default 90
        - p: float, default 0.3919 = theta_30
        - q: float, default 0.5953 = theta_31
        - scale: float, default 0.001
        """
```

```
        self.beta, self.theta_1, self.RC, self.T, self.p, self.q, self.scale =␣
↪beta, theta_1, RC, T, p, q, scale

        # Create matrix P with values of p, q, and 1-p-q on the diagonals
        self.P = np.diag(p * np.ones(T)) + np.diag(q * np.ones(T - 1), k=1) +␣
↪np.diag((1 - p - q) * np.ones(T - 2), k=2)

        # Normalize the last column of P so that the sum of each row in P is␣
↪equal to 1
        self.P[:, -1] += 1 - self.P.sum(axis=1)

        # Create an array x with values ranging from 0 to T-1
        self.x = np.arange(T, dtype=np.float64)


    @staticmethod
    def c(x, rust): # maintainance cost of the bus engine
        return rust.scale * rust.theta_1 * x


    @staticmethod
    def u(x, i, rust): # utility/payoff from the bus engine
        return - Rust.c(x, rust) if i == 0 else - Rust.c(x, rust) - rust.RC


    def solve_EV(self, tol=1e-3, maxiter=300000):
        """
        Solve for the expected value (EV) of the value function using the Rust␣
↪algorithm.
        Args:
        tol (float, optional): The tolerance level for convergence. Default is␣
↪1e-3.
        maxiter (int, optional): The maximum number of iterations. Default is␣
↪300000.
        Returns:
        numpy.ndarray: The expected value of the value function.
        """

        EV = np.zeros(self.T)  # Initialize the expected value array with zeros

        for _ in range(maxiter):  # Iterate until convergence or maximum␣
↪iterations

            # Compute expected values of continuing to wait and of replacing␣
↪the engine
            wait = Rust.u(self.x, 0, self) + self.beta * EV
```

```
            replace = Rust.u(self.x[0], 1, self) + self.beta * EV[0]

            # Calculate new expected values and update the EV array
            EV_new = np.exp(replace - EV) + np.exp(wait - EV)
            EV_new = self.P @ (np.log(EV_new) + EV)
            if np.linalg.norm(EV - EV_new) < tol:
                return EV_new
            EV = EV_new


    def conditional_probability(self):
        """
        Calculate the conditional probability of waiting versus replacing the engine given the expected value (EV) of the
        value function.

        Returns:
        Tuple of two floats: The probabilities of waiting and replacing the engine, respectively.
        """
        EV = self.solve_EV() # Calculate the expected value of the value function

        # Calculate the expected utility of waiting and of replacing the engine
        wait = Rust.u(self.x, 0, self) + self.beta * self.P @ EV
        replace = Rust.u(self.x[0], 1, self) + self.beta * EV[0]

        # Calculate the conditional probability of waiting
        P_wait = 1 / (1 + np.exp(replace - wait))

        # Return the probabilities of waiting and replacing the engine
        return P_wait, 1 - P_wait
```

[4]:
```
def log_likelihood(theta, df, p, q):
    """
    Compute the log-likelihood of the binary choice model given the parameter values `theta`, the data frame `df`, and
    the transition probabilities `p` and `q`.

    Returns:
    Float: The negative of the log-likelihood.
    """
    # Unpack the `theta` tuple into the `theta_1` and `RC` variables and create an instance of the `Rust` class
    # with these parameter values and the `p` and `q` probabilities.
    theta_1, RC = theta
    r = Rust(theta_1=theta_1, RC=RC, p=p, q=q)
```

```
    # Get the probabilities of waiting and replacing the engine
    P_wait, P_replace = r.conditional_probability()

    # Initialize a variable `logL` to zero and loop over each row in the `df`␣
↪data frame.
    logL = 0
    for decision, state in zip(df.decision, df.state.astype(np.int64)):
        # For each row, add the log-probability of the observed decision (wait␣
↪or replace) given the state to `logL`.
        logL += np.log(P_wait[state]) if decision == 0 else np.
↪log(P_replace[state])

    # Return the negative of `logL`, which is the negative of the␣
↪log-likelihood.
    return -logL
```

[5]:
```
%%time
p=0.3919
q=0.5953
results = minimize(log_likelihood, x0=(0.1, 10), args=(df, p, q)).x
```

```
CPU times: user 4min 6s, sys: 630 ms, total: 4min 6s
Wall time: 4min 6s
```

[6]:
```
results
```

[6]:
```
array([ 2.29716842, 10.20707904])
```

The first parameter in the above arrow indicates the estimate for θ_{11}, and the second parameter indicates the estimate for RC.

0.1.2 b. Provide pseudo-code for an alternative CCP method to the nested fixed-point method. Discuss one benefit and one limitation of this CCP method as compared to the full solution method you used in (a.). (To emphasize: no need to write the exact code or compute values with this method, pseudo-code is enough.)

Here is a pseudo-code for a CCP approach, alternative to Rust's nested fixed-point method and based on Hotz and Miller (1993).

Steps

1. Use frequency of the replacement decisions from the data to estimate the conditional choice probabilites (CCP)
2. Do Hotz and Miller inversion to find the estimate of expected value function using step 1.
3. Estimate expected policy function from step 2.

4. Compute the log-likelihood function, $\mathcal{L}(\theta)$, using step 3 to estimate the parameters θ.

Advantage: The main issue with the Rust approach is value function iteration: computation of the parameters suffers from curse of dimensionality if the state space gets large. Solving the model using Hotz and Miller (1993) gets around this issue as we don't solve a fixed point problem.

Limitation: As we need to discretize the state space, $H\&M$ approach can get complicated as we need enough variation in each state actions. While it's also valid in Rust, it is specifically an issue in $H\&M$ as it needs the consistency of CCP estimates.