Stanford CME 241 (Winter 2021) - Assignment 4

Job Hopping and Wages-Utility-Maximization

Chih-Hsuan 'Carolyn' Kao (chkao831@stanford.edu)

Mar 3rd, 2021

In [1]:

```
from dataclasses import dataclass, field
import numpy as np
from typing import Generic, List, Mapping, Set, TypeVar
```

In [2]:

```
@dataclass
class WageMaximizer():

    def __init__(

        self,
        gamma: float,
        alpha: float,
        employed_states: List[int],
        employed_wages: List[float],
        probabilities: List[float],
        unemployed_wage: float

    ):
        self.gamma = gamma
        self.alpha = alpha
        # S = {s | s = 1, 2, 3, ..., n}
        self.employed_states = employed_states
        self.employed_wages = employed_wages
        # s = n+1 with corresponding w+1
        self.unemployed_wage = unemployed_wage
        self.unemployed_state = len(employed_states) + 1
        # job-offer probabilities
        self.prob = probabilities
        # A = {0:reject, 1:accept}
        self.actions = {0, 1}

    def P(self, cr_state: int, action: str, nx_state: int) -> float:

        if (action, nx_state) == (1, self.unemployed_state):
            return 0

        if (cr_state, action) == (self.unemployed_state, 0):
            return 0

        if (cr_state, action, nx_state) ==\
            (self.unemployed_state, 0, self.unemployed_state):
```

```
            return 1

        if nx_state == self.unemployed_state:
            # with probability of alpha, will lose job
            return self.alpha

        if cr_state == nx_state:
            # with probability of 1-alpha, will not lose job
            return 1 - self.alpha

        if (cr_state, action) == (self.unemployed_state, 1):
            # while unemployed, offered job prob probs p1, p2...pn
            probs = { s:p for s,p in zip(self.employed_states, self.prob)}
            return probs[nx_state]

        return 0

    def R(self, state: int, action: str) -> float:

        if (state, action) == (self.unemployed_state, 0): # unemployed -> reject
            return np.log(self.unemployed_wage)
        if (state, action) == (self.unemployed_state, 1): # unemployed -> accept
            return np.log(np.array(self.employed_wages)@np.array(self.prob))

        # employed
        wage_series = { s:r for s,r in zip(self.employed_states, self.employed_wages)}
        return np.log(wage_series[state])

    def all_states(self) -> List[int]:

        return self.employed_states + [self.unemployed_state]

    def value_iteration(self) -> np.array:

        vk = np.zeros(len(self.all_states()))

        def optimization(s: int, v: np.array) -> float:

            maximum = float("-inf")

            for a in self.actions:
                summation = sum(
                    self.P(s, a, nx) * v[j] for j, nx in enumerate(self.all_states()))
                val = self.R(s, a) + self.gamma * summation

                maximum = max(val, maximum)

            return maximum

        while True:

            optimized = vk.copy()
            for i, state in enumerate(self.all_states()):
                optimized[i] = optimization(state, vk)
            if np.linalg.norm(optimized - vk) < 1e-8:
                return {state:val for state,val in zip(solver.all_states
(), optimized)}

            vk = optimized

    def optimal_policy(self) -> Mapping[int, str]:

        pi = {}
        v_star = self.value_iteration()

        def q_star(s: int, a: str) -> float:

            return (self.R(s, a) + self.gamma * sum(self.P(s, a, nx) * v_s
tar[nx]
                    for nx in self.all_states()))

        for s in self.all_states():

            maximum = float("-inf")
            action = None

            for a in self.actions:
                if q_star(s, a) > maximum:
                    maximum = q_star(s, a)
                    action = a
            pi[s] = action

        return pi
```

In [3]:

```
import random

gamma = 0.9
alpha = 0.1
employed_states = list(range(1,6))
employed_wages = []
for i in range(0,5):
    n = random.randint(2,10)
    employed_wages.append(n)
probs = np.random.dirichlet(np.ones(5),size=1).tolist()[0]
unemployed_wage: float = 1

solver = WageMaximizer(
    gamma=gamma,
    alpha=alpha,
    employed_states=employed_states,
    employed_wages=employed_wages,
    probabilities=np.array(probs),
    unemployed_wage=unemployed_wage,
)
v_star = solver.value_iteration()
v_star
```

Out[3]:

```
{1: 18.89329117290495,
 2: 18.89329117290495,
 3: 21.02731805768476,
 4: 21.02731805768476,
 5: 17.933704031884137,
 5: 17.[illegible],
 6: 19.977398413981216}
```

In [4]:

```
opt_policy = solver.optimal_policy()
print("Accept (1) or Reject (0) Decisions:")
print(opt_policy)
```

```
Accept (1) or Reject (0) Decisions:
{1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6: 1}
```

In [ ]:

```

```