C

Effectuation-Reinforcement-Learning-Modell

C.1 Abbildung des Zustandsraums in *OpenAI Gym*

Merkmal	Key	Space Class	Beispiel
E_t	'pv_entrepreneur'	MultiBinary	MultiBinary(5)(array ([0,1,0,1,0], dtype =int8))
C_t	'pv_customer'	MultiBinary	MultiBinary(5)(array ([0,1,0,1,0], dtype =int8))
$Cost_t$ I_t	'negotiation_info'	Box	spaces.Box(low=np.array([0,0]), high=np.array([MAX_COST,MAX_INVESTMENT]),dtype=np. ↪ float16
$Class_t$	' classification_result '	Discrete	Discrete(2) (0: real customer, 1: non-customer)

C.2 Bildung des Zustandsraums und der NK-Landschaft

```
# coding=utf-8
import numpy as np
import pandas as pd
from itertools import product, combinations_with_replacement


# number of decisions in the decision string
N = 10
# number of interdependent decisions
K = 8
# number of possible states of the decisions
A = 2
# maximal means
MAX_MEANS = 7500


# create the possible decision strings
def create_possible_decisions(N, A):
    possible_decisions = np.array(list(product(np.arange(A), repeat=N)))
    return possible_decisions


# building the contribution matrix which has dimensions N x A^(K+1)
```

```
def create_combinations_K(A, K):
    combinations_K = list(product(np.arange(A), repeat=(K + 1)))
    return combinations_K


def decision_config(N, possible_decisions, MAX_AL, fitness_array):
    #decision_configuration = np.zeros((N, 3 * len(possible_decisions)), dtype=object)
    dec_conf = pd.DataFrame(columns=['pv_entrepreneur', 'pv_customer', 'cost', '
        ↪ investment'])
    np.random.seed(0)
    for idx, i_val in enumerate(possible_decisions):
        i = 0
        for jdx, j_val in enumerate(possible_decisions):
            if jdx != idx:
                if np.sum(i != j for i, j in np.column_stack((i_val, j_val))) == 1:
                    cost = np.random.randint(0, MAX_AL)
                    if cost == 0:
                        investment = np.random.randint(cost, int(1.5*(cost+1)))
                    else:
                        investment = np.random.randint(cost, int(1.5*(cost)))
                    # algorithm for creating DictSpace
                    dec_conf = dec_conf.append({'pv_entrepreneur': idx, 'pv_customer':
                        ↪ jdx, 'cost': cost, 'investment': investment, 'mf_performance':
                        ↪  fitness_array[idx]}, ignore_index=True)
                    i += 1
    print(dec_conf)
    return dec_conf


# create the contribution matrix for the calculation of the fitness values of the
    ↪ possible decision strings
def create_contribution_matrix(N, K, combinations_K, possible_decisions):
    np.random.seed(0)
    contribution_matrix = pd.DataFrame(np.random.rand(N, len(combinations_K)), columns=[
        ↪ str(i) for i in combinations_K], dtype='float32')
    # create a N-dimensional matrix with A^N elements where each index represents a
        ↪ string combination, e.g. for A=2 und N=3 -> index() (0,1,1)
    a = np.empty((len(possible_decisions), N))
    for idx, i_val in enumerate(possible_decisions):
        for jdx, j_val in enumerate(i_val):
            k_list = [i_val[jdx]]
            for k in np.arange(K):
                k_list.append(i_val[(jdx + N - k - 1) % (N)])
            c = np.array(k_list)
            b = contribution_matrix.loc[jdx, str(tuple(c))]
            a[idx, jdx] = b
    return a


# starts the simulation
def simulate_nk_model(N, K, A, MAX_AL):
    possible_decisions = create_possible_decisions(N, A)
    combinations_K = create_combinations_K(A, K)
    # possible decision strings as labels for several Tensors
    labels = [str(i) for i in possible_decisions]
    contribution_matrix = create_contribution_matrix(N, K, combinations_K,
        ↪ possible_decisions)
    fitness_array = pd.Series(np.sum(contribution_matrix, axis=1) / N, index=labels)
    decision_configuration = decision_config(N, possible_decisions, MAX_AL, fitness_array
        ↪ )

    test_results = 2
    col5 = np.repeat(np.arange(0.0, test_results), len(decision_configuration))
    decision_configuration = pd.concat([decision_configuration] * test_results,
        ↪ ignore_index=True)
    decision_configuration.insert(5, 'classification_result', col5, True)

    possible_decisions_mapping = pd.DataFrame({'Numeric Value': np.arange(0.0, len(
        ↪ possible_decisions)), 'Decision String': possible_decisions.tolist()})
```

```
83
84      print(possible_decisions_mapping)
85      print(decision_configuration)
86
87
88  simulate_nk_model(N, K, A, MAX_MEANS)
```

C.3 Aufbau der Umgebung des Reinforcement-Learning-Problems in *OpenAI Gym* (Modul `EffectuationEnv`)

```
1   import random
2   import json
3   import gym
4   from gym import spaces
5   import pandas as pd
6   import numpy as np
7   import math as m
8
9
10  # see KfW report (2017) - mean seed capital of entrepreneurs: 7500 Euro
11  INITIAL_MEANS = 7500
12  MAX_COST = INITIAL_MEANS
13  MAX_INVESTMENT = MAX_COST * 1.5
14  CLASS_REWARD_CLASS0a0 = 1
15  CLASS_REWARD_CLASS0a1 = 0
16  CLASS_REWARD_CLASS1a0 = 0
17  CLASS_REWARD_CLASS1a1 = 0.5
18  # default value = 1
19  OMEGA_1 = 1
20  OMEGA_2 = 1
21  OMEGA_3 = 1
22  # default value for PHI = CHI = 0.5
23  PHI = 0.5
24  CHI = 0.5
25  PSI = 0.307 / (1-PHI)
26
27
28  class EffectuationEnv(gym.Env):
29      """An effectuation environment for OpenAI gym"""
30      metadata = {'render.modes': ['human']}
31
32      def __init__(self, df):
33          super(EffectuationEnv, self).__init__()
34          self.df_marketfit = df.iloc[:,-2]
35          self.df = df.drop(df.columns[-2], axis=1)
36          self.means_ratio = 0
37          # action 0: classify customer as real customer, action 1: classify customer as
                ↪ non-customer
38          self.action_space = spaces.Discrete(2)
39          # Observations contain: the product vectors of the entrepreneur and customer,
                ↪ costs for changing the pv, possible investment volume,  the result of the
                ↪ classification
40          self.observation_space = spaces.Dict({
41              'pv_entrepreneur': spaces.MultiBinary(10),
42              'pv_customer': spaces.MultiBinary(10),
43              'negotiation_info': spaces.Box(
44                  low=np.array([0, 0]),
45                  high=np.array([MAX_COST, MAX_INVESTMENT]),
46                  dtype=np.float16),
47              'classicifaction_result': spaces.Discrete(2)})
48
49
50      def _means_calculation(self, next_obs, action):
51          current_cost = self.df.at[self.current_state, 'cost']
52          current_investment = self.df.at[self.current_state, 'investment']
53          # set the classicifaction result to the value of the next state
54          classification_result = next_obs['classification_result']
```

```
55          if action == 0:
56              if classification_result == 0:
57                  self.means = self.means - current_cost + current_investment
58              else:
59                  self.means = self.means - current_cost
60          else:
61              self.means = self.means
62
63
64      def _take_action(self, action):
65          # get the next customer based on the action taken and the transition
                ↪ probabilities
66          if action == 0:
67              next_customers = self.df[self.df.at[self.current_state, 'pv_customer'] ==
                    ↪ self.df['pv_entrepreneur']]
68              class_res_0 = next_customers[next_customers['classification_result'] == 0]
69              class_res_1 = next_customers[next_customers['classification_result'] == 1]
70              prob_class_res_0 = ((1 - PHI) * PSI) / (CHI * (1 - PSI) + (1 - PHI) * PSI)
71          else:
72              next_customers = self.df[self.df.at[self.current_state, 'pv_entrepreneur'] ==
                    ↪  self.df['pv_entrepreneur']]
73              class_res_0 = next_customers[next_customers['classification_result'] == 0]
74              class_res_1 = next_customers[next_customers['classification_result'] == 1]
75              prob_class_res_0 = (PHI * PSI) / (PHI * PSI + (1 - CHI) * (1-PSI))
76          if random.uniform(0, 1) < prob_class_res_0:
77              next_customers = class_res_0
78          else:
79              next_customers = class_res_1
80          idx = random.choice(next_customers.index.values)
81          next_obs = self.df.iloc[idx]
82          # calculate the means
83          self.means_old = self.means
84          self._means_calculation(next_obs, action)
85          # Append additional data
86          obs = np.append(next_obs, [self.means], axis=0)
87          self.old_state = self.current_state
88          # set the current state to the index of the next observation
89          self.current_state = idx
90          return obs
91
92
93      def _reward_calculation(self, action):
94          means_ratio_reward =  1 + ((self.means - self.means_old)  / INITIAL_MEANS)
95          if action == 0:
96              if self.df.at[self.current_state, 'classification_result'] == 0:
97                  classification_reward = CLASS_REWARD_CLASS0a0
98              else:
99                  classification_reward = CLASS_REWARD_CLASS1a0
100         else:
101             if self.df.at[self.current_state, 'classification_result'] == 0:
102                 classification_reward = CLASS_REWARD_CLASS0a1
103             else:
104                 classification_reward = CLASS_REWARD_CLASS1a1
105         market_fit_reward = self.df_marketfit.iloc[self.current_state]
106
107
108         def _means_reward_calculation(means_ratio_reward):
109             # avoid float overflow
110             if means_ratio_reward < 0.05:
111                 means_ratio_reward = 0
112             elif means_ratio_reward > 20:
113                 means_ratio_reward = 1
114             else:
115                 means_ratio_reward = (m.exp(means_ratio_reward)) / (m.exp(1 /
                        ↪ means_ratio_reward) + m.exp(means_ratio_reward))
116             return means_ratio_reward
117
118
```

```
119             reward = (OMEGA_1 * _means_reward_calculation(means_ratio_reward) + OMEGA_2 *
                    ↪ classification_reward + OMEGA_3 * market_fit_reward) / (OMEGA_1 + OMEGA_2
                    ↪ + OMEGA_3)
120             return reward
121
122
123         def step(self, action):
124             # get the observation of the next state
125             obs = self._take_action(action)
126             # increase the step counter
127             self.current_step += 1
128             # calculate the the reward for doing the chosen action
129             reward = self._reward_calculation(action)
130             done = False
131             return obs, reward, done, {'state': self.current_state}
132
133
134         def reset(self):
135             # reset the state of the environment to an initial state
136             self.means = INITIAL_MEANS
137             # set the current step to a random row index within the data frame
138             self.current_state = random.randint(0, len(self.df) - 1)
139             # store the last state for calculation purposes
140             self.old_state = self.current_state
141             # reset the step
142             self.current_step = 0
143             # get the initial observation when starting the simulation
144             initial_obs = self.df.iloc[self.current_state]
145             obs = np.append(initial_obs, [self.means], axis=0)
146             return obs
147
148
149         def render(self, mode='human', close=False):
150             print(f'Means: {self.means}')
```

C.4 Lernalgorithmus des Agenten gemäß des Q-Learning-Prinzips

```
1  import gym
2  from statistics import mean
3  import numpy as np
4  import pandas as pd
5  import random
6  import datetime
7  from env.effectuation_env_non_episodic import EffectuationEnv
8
9
10 """Initialize Environment"""
11 # Greed
12 epsilon = 1
13 # Minimum Greed
14 epsilon_min = 0.05
15 # Decay multiplied with epsilon after each episode
16 epsilon_decay = 0.9
17 # Amount of learning episodes
18 episodes = 10000
19 # Maximum timesteps per episode
20 max_steps = 100
21 # learning rate
22 learning_rate = 0.6
23 # discount factor
24 gamma = 0.2
25
26 # Initialize environment to which the q-learning algorithm will be applied
27 env = EffectuationEnv(df)
28
29 # Initalize the state and action space
30 state_space = len(df)
31 action_space = env.action_space.n
```

```
32
33  # Initialize the Q-Table with zeros
34  qtable = np.zeros((state_space, action_space))
35
36  # Initialize the list for storing the received rewards per episode
37  return_per_episode = []
38
39  # Initialize the list for storing the MSE
40  q_loss_per_episode = [np.copy(qtable)]
41  q_err_lst = [0]
42
43  # Initialize the data frame for storing the reward distribution
44  reward_dist = pd.DataFrame(columns=['episode', 'reward'])
45  rewards_avg = []
46
47  # Initialize the imported effectuation environment
48  env.reset()
49  state = env.current_state
50
51  # the path in the environment for the agent
52  for episode in range(cfg.episodes):
53      score = 0
54      qtable_old = np.copy(qtable)
55
56      for i in range(cfg.max_steps):
57          # With the probability of (1 - epsilon) take the best action in our Q-table
58          if random.uniform(0, 1) > cfg.epsilon:
59              action = np.argmax(qtable[state, :])
60          # Else take a random action
61          # Take action which is not the best with probability epsilon
62          else:
63              action = env.action_space.sample()
64
65          # Step the agent forward
66          next_obs, reward, done, next_state = env.step(action)
67          next_state = next_state['state']
68
69          # Add up the score
70          score += reward
71
72          # update the q-table
73          qtable[state, action] = (1 - cfg.learning_rate) * qtable[state, action] + (cfg.
                ↪ learning_rate) * (reward + cfg.gamma * np.max(qtable[next_state,:]))
74
75          state = next_state
76
77      if episode == (cfg.episodes - 1):
78          print('Ep 1: {}'.format(episode))
79          print("q-Table 1: {}".format(sum(qtable)))
80
81      # Calculate the MSE
82      q_err = np.mean((qtable - qtable_old)**2)
83
84      q_err_lst.append(q_err)
85
86      return_per_episode.append(score / cfg.max_steps)
87
88      # Reducing our epsilon each episode (Exploration-Exploitation trade-off)
89      if cfg.epsilon >= cfg.epsilon_min:
90          cfg.epsilon *= cfg.epsilon_decay
91
92
93  print('Average Returns: {}'.format(return_per_episode))
94  print('MSE: {}'.format(q_err_lst))
```