SARSA Learning Algorithm

AIM

To implement the SARSA algorithm to teach an agent how to choose actions that lead to the best outcomes in a given environment. The aim is to evaluate the agent's performance and visualize its learning process.

PROBLEM STATEMENT

The goal is to create a system that helps an agent learn the best actions to take in an environment to maximize rewards over time. We will use the SARSA (State-Action-Reward-State-Action) algorithm, which is a type of reinforcement learning. The agent will learn from its experiences by interacting with the environment and adjusting its strategy accordingly.

SARSA LEARNING ALGORITHM

Step 1:

Initialize the Q-value table with zeros for all state-action pairs.

Step 2:

Set the parameters: learning rate (alpha), discount factor (gamma), and exploration rate (epsilon).

Step 3:

For each episode, reset the environment and observe the initial state.

Step 4:

Choose an action based on the current state's Q-values using an epsilon-greedy policy.

Step 5:

Take the chosen action, observe the reward and the next state.

Step 6:

Choose the next action using the same epsilon-greedy policy.

Step 7:

Update the Q-value for the current state-action pair using the SARSA update rule.

Step 8:

Repeat steps 4-7 until the episode ends, then repeat for multiple episodes.

SARSA LEARNING FUNCTION

Name: Meetha Prabhu

Register Number: 212222240065

Import necessary packages

```
import warnings ; warnings.filterwarnings('ignore')

import itertools
import gym, gym_walk
import numpy as np
from tabulate import tabulate
from pprint import pprint
from tqdm import tqdm_notebook as tqdm

from itertools import cycle, count

import random
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.pylab as pylab
SEEDS = (12, 34, 56, 78, 90)

%matplotlib inline
```

Function to create plots (size wise)

```
plt.style.use('fivethirtyeight')
params = {
    'figure.figsize': (5, 5),
    'font.size': 24,
    'legend.fontsize': 20,
    'axes.titlesize': 28,
    'axes.labelsize': 24,
    'xtick.labelsize': 20,
    'ytick.labelsize': 20
}
pylab.rcParams.update(params)
np.set_printoptions(suppress=True)
```

Value Iteration and Printing Functions

```
def value_iteration(P, gamma=1.0, theta=1e-10):
    V = np.zeros(len(P), dtype=np.float64)
    while True:
        Q = np.zeros((len(P), len(P[0])), dtype=np.float64)
        for s in range(len(P)):
            for a in range(len(P[s])):
                for prob, next_state, reward, done in P[s][a]:
                    Q[s][a] += prob * (reward + gamma * V[next_state] * (not
done))
        if np.max(np.abs(V - np.max(Q, axis=1))) < theta:
            break
        V = np.max(Q, axis=1)
    pi = lambda s: {s:a for s, a in enumerate(np.argmax(Q, axis=1))}[s]
    return Q, V, pi

def print_policy(pi, P, action_symbols=('<', 'v', '>', '^'), n_cols=4,
title='Policy:'):
    print(title)
    arrs = {k:v for k,v in enumerate(action_symbols)}
    for s in range(len(P)):
        a = pi(s)
        print("| ", end="")
        if np.all([done for action in P[s].values() for _, _, _, done in
action]):
            print("".rjust(9), end=" ")
        else:
            print(str(s).zfill(2), arrs[a].rjust(6), end=" ")
        if (s + 1) % n_cols == 0: print("|")

def print_state_value_function(V, P, n_cols=4, prec=3, title='State-value
function:'):
    print(title)
    for s in range(len(P)):
        v = V[s]
        print("| ", end="")
        if np.all([done for action in P[s].values() for _, _, _, done in
action]):
            print("".rjust(9), end=" ")
        else:
            print(str(s).zfill(2), '{}'.format(np.round(v, prec)).rjust(6),
end=" ")
```

```
        if (s + 1) % n_cols == 0: print("|")

def print_action_value_function(Q,
                                optimal_Q=None,
                                action_symbols=('<', '>'),
                                prec=3,
                                title='Action-value function:'):
    vf_types=('',) if optimal_Q is None else ('', '*', 'err')
    headers = ['s',] + [' '.join(i) for i in
list(itertools.product(vf_types, action_symbols))]
    print(title)
    states = np.arange(len(Q))[..., np.newaxis]
    arr = np.hstack((states, np.round(Q, prec)))
    if not (optimal_Q is None):
        arr = np.hstack((arr, np.round(optimal_Q, prec), np.round(optimal_Q-
Q, prec)))
    print(tabulate(arr, headers, tablefmt="fancy_grid"))
```

Policy Metrics Fucntion

```
def get_policy_metrics(env, gamma, pi, goal_state, optimal_Q,
                       n_episodes=100, max_steps=200):
    random.seed(123); np.random.seed(123) ; env.seed(123)
    reached_goal, episode_reward, episode_regret = [], [], []
    for _ in range(n_episodes):
        state, done, steps = env.reset(), False, 0
        episode_reward.append(0.0)
        episode_regret.append(0.0)
        while not done and steps < max_steps:
            action = pi(state)
            regret = np.max(optimal_Q[state]) - optimal_Q[state][action]
            episode_regret[-1] += regret

            state, reward, done, _ = env.step(action)
            episode_reward[-1] += (gamma**steps * reward)

            steps += 1

        reached_goal.append(state == goal_state)
    results = np.array((np.sum(reached_goal)/len(reached_goal)*100,
                        np.mean(episode_reward),
                        np.mean(episode_regret)))
    return results

def get_metrics_from_tracks(env, gamma, goal_state, optimal_Q, pi_track,
coverage=0.1):
    total_samples = len(pi_track)
    n_samples = int(total_samples * coverage)
    samples_e = np.linspace(0, total_samples, n_samples, endpoint=True,
dtype=np.int)
```

```
    metrics = []
    for e, pi in enumerate(tqdm(pi_track)):
        if e in samples_e:
            metrics.append(get_policy_metrics(
                env,
                gamma=gamma,
                pi=lambda s: pi[s],
                goal_state=goal_state,
                optimal_Q=optimal_Q))
        else:
            metrics.append(metrics[-1])
    metrics = np.array(metrics)
    success_rate_ma, mean_return_ma, mean_regret_ma =
np.apply_along_axis(moving_average, axis=0, arr=metrics).T
    return success_rate_ma, mean_return_ma, mean_regret_ma

def rmse(x, y, dp=4):
    return np.round(np.sqrt(np.mean((x - y)**2)), dp)

def moving_average(a, n=100) :
    ret = np.cumsum(a, dtype=float)
    ret[n:] = ret[n:] - ret[:-n]
    return ret[n - 1:] / n
```

Function to plot the value function

```
def plot_value_function(title, V_track, V_true=None, log=False,
limit_value=0.05, limit_items=5):
    np.random.seed(123)
    per_col = 25
    linecycler = cycle(["-","--",":","-."])
    legends = []

    valid_values = np.argwhere(V_track[-1] > limit_value).squeeze()
    items_idxs = np.random.choice(valid_values,
                                  min(len(valid_values), limit_items),
                                  replace=False)
    # draw the true values first
    if V_true is not None:
        for i, state in enumerate(V_track.T):
            if i not in items_idxs:
                continue
            if state[-1] < limit_value:
                continue

            label = 'v*({})'.format(i)
            plt.axhline(y=V_true[i], color='k', linestyle='-', linewidth=1)
            plt.text(int(len(V_track)*1.02), V_true[i]+.01, label)

    # then the estimates
    for i, state in enumerate(V_track.T):
        if i not in items_idxs:
            continue
        if state[-1] < limit_value:
            continue
        line_type = next(linecycler)
        label = 'V({})'.format(i)
        p, = plt.plot(state, line_type, label=label, linewidth=3)
        legends.append(p)

    legends.reverse()

    ls = []
    for loc, idx in enumerate(range(0, len(legends), per_col)):
        subset = legends[idx:idx+per_col]
        l = plt.legend(subset, [p.get_label() for p in subset],
                       loc='center right', bbox_to_anchor=(1.25, 0.5))
        ls.append(l)
    [plt.gca().add_artist(l) for l in ls[:-1]]
    if log: plt.xscale('log')
    plt.title(title)
    plt.ylabel('State-value function')
    plt.xlabel('Episodes (log scale)' if log else 'Episodes')
    plt.show()
```

Decay schedule

```
def decay_schedule(init_value, min_value, decay_ratio, max_steps,
log_start=-2, log_base=10):
    decay_steps = int(max_steps * decay_ratio)
    rem_steps = max_steps - decay_steps
    values = np.logspace(log_start, 0, decay_steps, base=log_base,
endpoint=True)[::-1]
    values = (values - values.min()) / (values.max() - values.min())
    values = (init_value - min_value) * values + min_value
    values = np.pad(values, (0, rem_steps), 'edge')
    return values
```

SLIPPERY WALK SEVEN

```
env = gym.make('SlipperyWalkSeven-v0')
init_state = env.reset()
goal_state = 8
gamma = 0.99
n_episodes = 3000
P = env.env.P
n_cols, svf_prec, err_prec, avf_prec=9, 4, 2, 3
action_symbols=('<', '>')
```

```
limit_items, limit_value = 5, 0.0
cu_limit_items, cu_limit_value, cu_episodes = 10, 0.0, 100
```

Alpha and Epsilon schedules

```
plt.plot(decay_schedule(0.5, 0.01, 0.5, n_episodes),
         '-', linewidth=2,
         label='Alpha schedule')
plt.plot(decay_schedule(1.0, 0.1, 0.9, n_episodes),
         ':', linewidth=2,
         label='Epsilon schedule')
plt.legend(loc=1, ncol=1)
plt.figsize=(5,5)
plt.title('Alpha and epsilon schedules')
plt.xlabel('Episodes')
plt.ylabel('Hyperparameter values')
plt.xticks(rotation=45)

plt.show()
```

Optimal value functions and policy

```
optimal_Q, optimal_V, optimal_pi = value_iteration(P, gamma=gamma)
print_state_value_function(optimal_V, P, n_cols=n_cols, prec=svf_prec,
title='Optimal state-value function:')
print()

print_action_value_function(optimal_Q,
                            None,
                            action_symbols=action_symbols,
                            prec=avf_prec,
                            title='Optimal action-value function:')
print()
print_policy(optimal_pi, P, action_symbols=action_symbols, n_cols=n_cols)
success_rate_op, mean_return_op, mean_regret_op = get_policy_metrics(
    env, gamma=gamma, pi=optimal_pi, goal_state=goal_state,
optimal_Q=optimal_Q)
print('Reaches goal {:.2f}%. Obtains an average return of {:.4f}. Regret of
{:.4f}'.format(
    success_rate_op, mean_return_op, mean_regret_op))
```

Trajectory Function

```
def generate_trajectory(select_action, Q, epsilon, env, max_steps=200):
    done, trajectory = False, []
    while not done:
        state = env.reset()
        for t in count():
            action = select_action(state, Q, epsilon)
            next_state, reward, done, _ = env.step(action)
            experience = (state, action, reward, next_state, done)
            trajectory.append(experience)
            if done:
                break
            if t >= max_steps - 1:
                trajectory = []
                break
            state = next_state
    return np.array(trajectory, object)
```

Monte - carlo control Function

```
def mc_control(env,
               gamma=1.0,
               init_alpha=0.5,
               min_alpha=0.01,
               alpha_decay_ratio=0.5,
               init_epsilon=1.0,
               min_epsilon=0.1,
               epsilon_decay_ratio=0.9,
               n_episodes=3000,
               max_steps=200,
               first_visit=True):
    nS, nA = env.observation_space.n, env.action_space.n
    discounts = np.logspace(0,
                            max_steps,
                            num=max_steps,
                            base=gamma,
                            endpoint=False)
    alphas = decay_schedule(init_alpha,
                           min_alpha,
                           alpha_decay_ratio,
                           n_episodes)
    epsilons = decay_schedule(init_epsilon,
                              min_epsilon,
                              epsilon_decay_ratio,
                              n_episodes)
    pi_track = []
    Q = np.zeros((nS, nA), dtype=np.float64)
    Q_track = np.zeros((n_episodes, nS, nA), dtype=np.float64)
    select_action = lambda state, Q, epsilon: np.argmax(Q[state]) \
        if np.random.random() > epsilon \
        else np.random.randint(len(Q[state]))

    for e in tqdm(range(n_episodes), leave=False):

        trajectory = generate_trajectory(select_action
        trajectory = generate_trajectory(select_action,
                                         Q,
                                         epsilons[e],
                                         env,
                                         max_steps)
        visited = np.zeros((nS, nA), dtype=bool)
        for t, (state, action, reward, _, _) in enumerate(trajectory):
            if visited[state][action] and first_visit:
                continue
            visited[state][action] = True

            n_steps = len(trajectory[t:])
            G = np.sum(discounts[:n_steps] * trajectory[t:, 2])
            Q[state][action] = Q[state][action] + alphas[e] * (G - Q[state]
[action])

        Q_track[e] = Q
        pi_track.append(np.argmax(Q, axis=1))

    V = np.max(Q, axis=1)
    pi = lambda s: {s:a for s, a in enumerate(np.argmax(Q, axis=1))}[s]
    return Q, V, pi, Q_track, pi_track

Q_mcs, V_mcs, Q_track_mcs = [], [], []
for seed in tqdm(SEEDS, desc='All seeds', leave=True):
    random.seed(seed); np.random.seed(seed) ; env.seed(seed)
    Q_mc, V_mc, pi_mc, Q_track_mc, pi_track_mc = mc_control(env,
gamma=gamma, n_episodes=n_episodes)
    Q_mcs.append(Q_mc) ; V_mcs.append(V_mc) ; Q_track_mcs.append(Q_track_mc)
Q_mc, V_mc, Q_track_mc = np.mean(Q_mcs, axis=0), np.mean(V_mcs, axis=0),
np.mean(Q_track_mcs, axis=0)
del Q_mcs ; del V_mcs ; del Q_track_mcs
```

Printing policies

```
print('Name: Meetha Prabhu  Register Number: 212222240065            ')
print_state_value_function(V_mc, P, n_cols=n_cols,
                           prec=svf_prec, title='State-value function found
by FVMC:')
print_state_value_function(optimal_V, P, n_cols=n_cols,
                           prec=svf_prec, title='Optimal state-value
function:')
print_state_value_function(V_mc - optimal_V, P, n_cols=n_cols,
                           prec=err_prec, title='State-value function
errors:')
print('State-value function RMSE: {}'.format(rmse(V_mc, optimal_V)))
print()
print_action_value_function(Q_mc,
                            optimal_Q,
                            action_symbols=action_symbols,
                            prec=avf_prec,
                            title='FVMC action-value function:')
print('Action-value function RMSE: {}'.format(rmse(Q_mc, optimal_Q)))
print()
print_policy(pi_mc, P, action_symbols=action_symbols, n_cols=n_cols)
success_rate_mc, mean_return_mc, mean_regret_mc = get_policy_metrics(
    env, gamma=gamma, pi=pi_mc, goal_state=goal_state, optimal_Q=optimal_Q)
print('Reaches goal {:.2f}%. Obtains an average return of {:.4f}. Regret of
{:.4f}'.format(
    success_rate_mc, mean_return_mc, mean_regret_mc))
```

SARSA Function

```
from tqdm import tqdm
import numpy as np
def sarsa(env, gamma=1.0, init_alpha=0.5, min_alpha=0.01,
alpha_decay_ratio=0.5,init_epsilon=0.1, min_epsilon=0.1,
epsilon_decay_ratio=0.9, n_episodes=3000):
  nS, nA=env.observation_space.n, env.action_space.n
  pi_track=[]
  Q=np.zeros((nS,nA), dtype=np.float64)
  Q_track=np.zeros((n_episodes,nS,nA), dtype=np.float64)

  select_action=lambda state, Q, epsilon: np.argmax(Q[state]) if
np.random.random() > epsilon else np.random.randint(len(Q[state]))

  alphas=decay_schedule(init_alpha,  min_alpha, alpha_decay_ratio,
n_episodes)

  epsilons=decay_schedule(init_epsilon, min_epsilon, epsilon_decay_ratio,
n_episodes)

  for e in  tqdm(range(n_episodes),leave=False):
    state, done=env.reset(),False
    action=select_action(state, Q, epsilons[e])
    while not done:
      next_state, reward, done, _=env.step(action)
      next_action=select_action(next_state, Q, epsilons[e])
      td_target = reward+gamma * Q[next_state][next_action] * (not done)

      td_error=td_target-Q[state][action]
      Q[state][action]=Q[state][action]+alphas[e] * td_error
      state, action=next_state, next_action
      Q_track[e] = Q
      pi_track.append(np.argmax(Q,axis=1))
    V = np.max(Q, axis=1)
    pi=lambda s:{s:a for s, a in enumerate(np.argmax(Q, axis=1))}[s]
  return Q, V, pi, Q_track, pi_track

Q_sarsas, V_sarsas, Q_track_sarsas = [], [], []
for seed in tqdm(SEEDS, desc='All seeds', leave=True):
    random.seed(seed); np.random.seed(seed) ; env.seed(seed)
    random.seed(seed); np.random.seed(seed) ; env.seed(seed)
    Q_sarsa, V_sarsa, pi_sarsa, Q_track_sarsa, pi_track_sarsa = sarsa(env,
gamma=gamma, n_episodes=n_episodes)
    Q_sarsas.append(Q_sarsa) ; V_sarsas.append(V_sarsa) ;
Q_track_sarsas.append(Q_track_sarsa)
Q_sarsa = np.mean(Q_sarsas, axis=0)
V_sarsa = np.mean(V_sarsas, axis=0)
Q_track_sarsa = np.mean(Q_track_sarsas, axis=0)
del Q_sarsas ; del V_sarsas ; del Q_track_sarsas
```

Printing policies

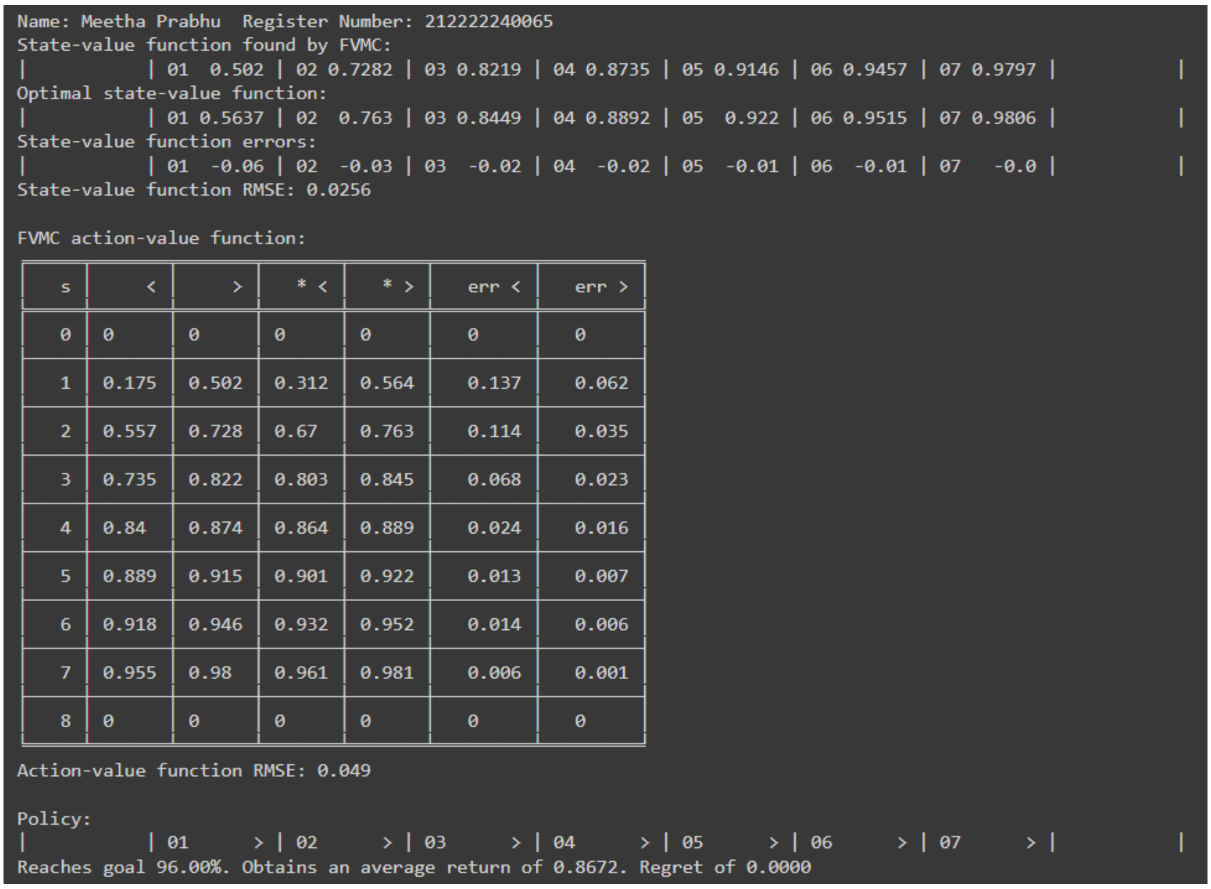
```
print('Name: Meetha Prabhu   Register Number: 212222240065             ')
print_state_value_function(V_sarsa, P, n_cols=n_cols,
                           prec=svf_prec, title='State-value function found
by Sarsa:')
print_state_value_function(optimal_V, P, n_cols=n_cols,
                           prec=svf_prec, title='Optimal state-value
function:')
print_state_value_function(V_sarsa - optimal_V, P, n_cols=n_cols,
                           prec=err_prec, title='State-value function
errors:')
print('State-value function RMSE: {}'.format(rmse(V_sarsa, optimal_V)))
print()
print_action_value_function(Q_sarsa,
                            optimal_Q,
                            action_symbols=action_symbols,
                            prec=avf_prec,
                            title='Sarsa action-value function:')
print('Action-value function RMSE: {}'.format(rmse(Q_sarsa, optimal_Q)))
print()
print_policy(pi_sarsa, P, action_symbols=action_symbols, n_cols=n_cols)
success_rate_sarsa, mean_return_sarsa, mean_regret_sarsa =
get_policy_metrics(
    env, gamma=gamma, pi=pi_sarsa, goal_state=goal_state,
optimal_Q=optimal_Q)
print('Reaches goal {:.2f}%. Obtains an average return of {:.4f}. Regret of
{:.4f}'.format(
    success_rate_sarsa, mean_return_sarsa, mean_regret_sarsa))
```

Comparision

```
plot_value_function(
    'FVMC estimates through time vs. true values  \nName: Meetha Prabhu
Reister Number: 212222240065           ',
    np.max(Q_track_mc, axis=2),
    optimal_V,
    limit_items=limit_items,
    limit_value=limit_value,
    log=False)


plot_value_function(
    'Sarsa estimates through time vs. true values       Name:Meetha Prabhu
Reister Number: 212222240065           ',
    np.max(Q_track_sarsa, axis=2),
    optimal_V,
    limit_items=limit_items,
    limit_value=limit_value,
    log=False)
```

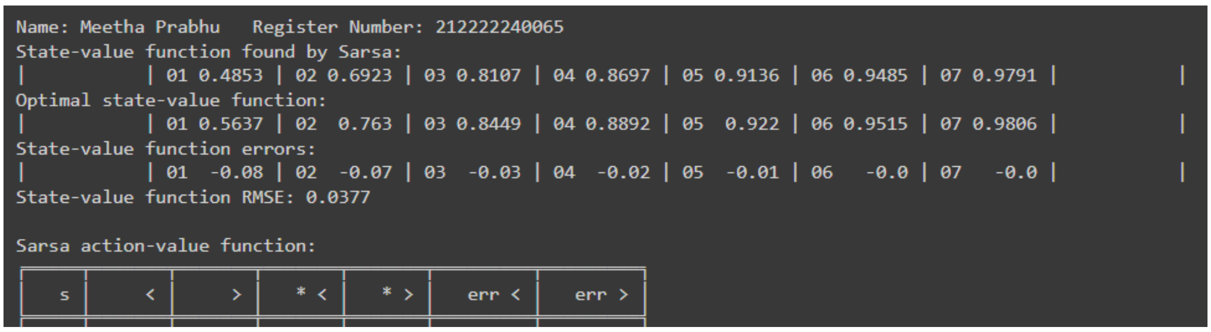
OUTPUT:

MC- control Policies

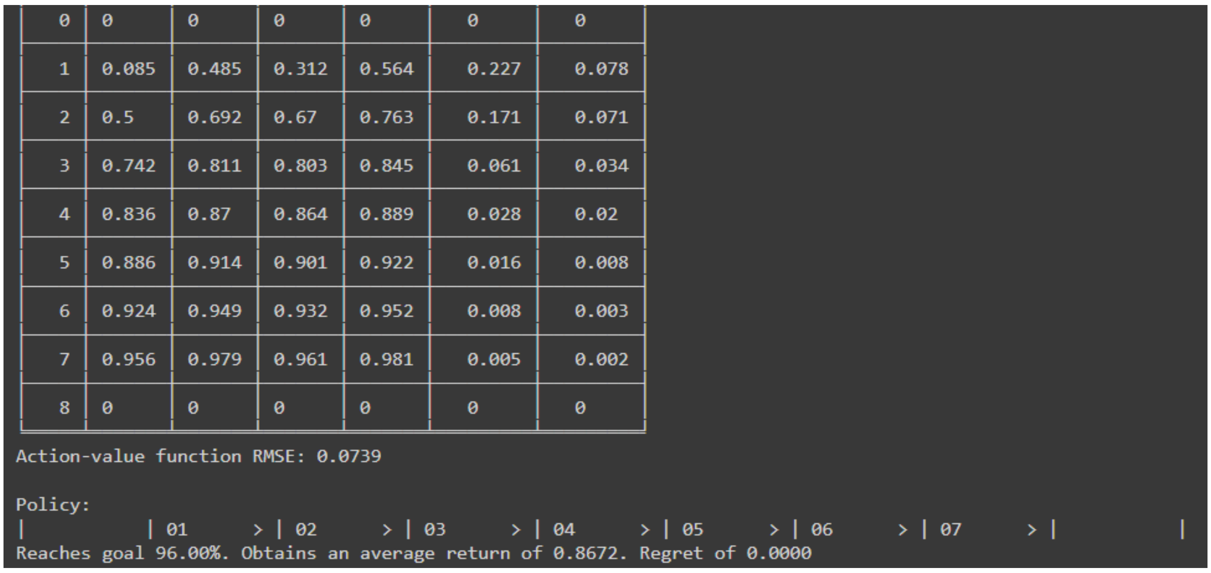


```
Name: Meetha Prabhu  Register Number: 212222240065
State-value function found by FVMC:
|           | 01  0.502 | 02 0.7282 | 03 0.8219 | 04 0.8735 | 05 0.9146 | 06 0.9457 | 07 0.9797 |           |
Optimal state-value function:
|           | 01 0.5637 | 02  0.763 | 03 0.8449 | 04 0.8892 | 05  0.922 | 06 0.9515 | 07 0.9806 |           |
State-value function errors:
|           | 01  -0.06 | 02  -0.03 | 03  -0.02 | 04  -0.02 | 05  -0.01 | 06  -0.01 | 07   -0.0 |           |
State-value function RMSE: 0.0256

FVMC action-value function:
| s |     < |     > |   * < |   * > | err < | err > |
| 0 | 0     | 0     | 0     | 0     | 0     | 0     |
| 1 | 0.175 | 0.502 | 0.312 | 0.564 | 0.137 | 0.062 |
| 2 | 0.557 | 0.728 | 0.67  | 0.763 | 0.114 | 0.035 |
| 3 | 0.735 | 0.822 | 0.803 | 0.845 | 0.068 | 0.023 |
| 4 | 0.84  | 0.874 | 0.864 | 0.889 | 0.024 | 0.016 |
| 5 | 0.889 | 0.915 | 0.901 | 0.922 | 0.013 | 0.007 |
| 6 | 0.918 | 0.946 | 0.932 | 0.952 | 0.014 | 0.006 |
| 7 | 0.955 | 0.98  | 0.961 | 0.981 | 0.006 | 0.001 |
| 8 | 0     | 0     | 0     | 0     | 0     | 0     |
Action-value function RMSE: 0.049

Policy:
|           | 01      > | 02      > | 03      > | 04      > | 05      > | 06      > | 07      > |           |
Reaches goal 96.00%. Obtains an average return of 0.8672. Regret of 0.0000
```

SARSA policies

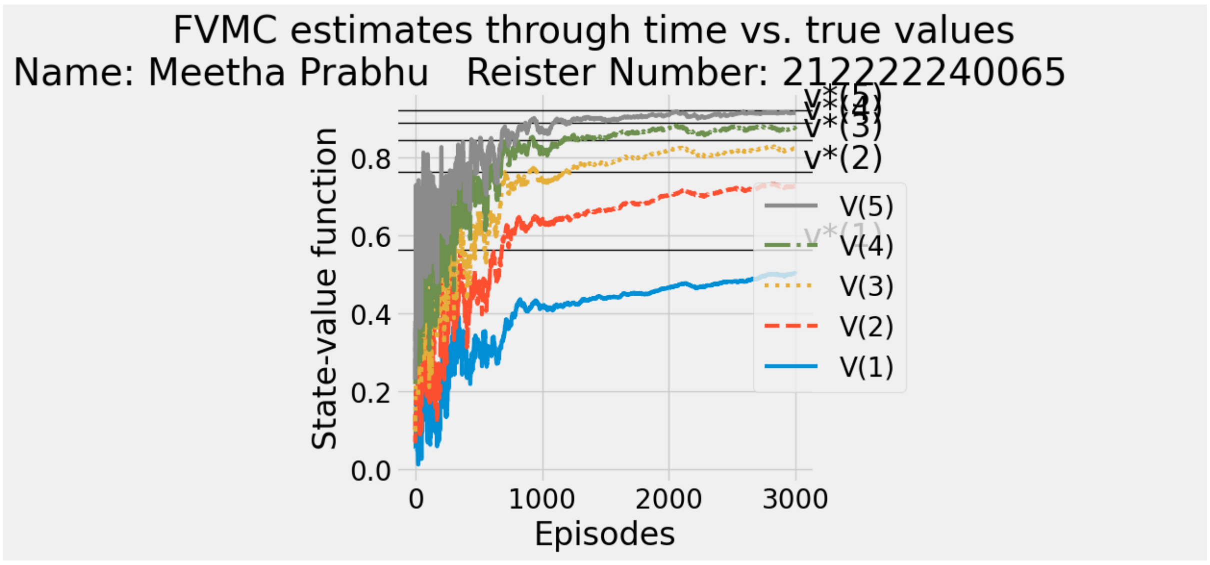


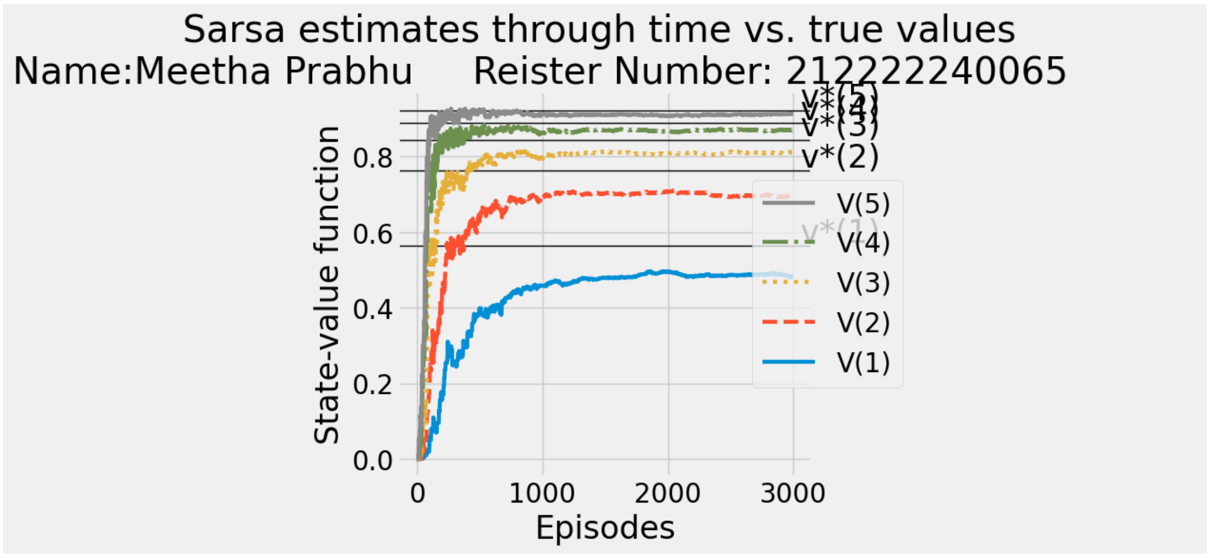
```
Name: Meetha Prabhu   Register Number: 212222240065
State-value function found by Sarsa:
|           | 01 0.4853 | 02 0.6923 | 03 0.8107 | 04 0.8697 | 05 0.9136 | 06 0.9485 | 07 0.9791 |           |
Optimal state-value function:
|           | 01 0.5637 | 02  0.763 | 03 0.8449 | 04 0.8892 | 05  0.922 | 06 0.9515 | 07 0.9806 |           |
State-value function errors:
|           | 01  -0.08 | 02  -0.07 | 03  -0.03 | 04  -0.02 | 05  -0.01 | 06   -0.0 | 07   -0.0 |           |
State-value function RMSE: 0.0377

Sarsa action-value function:
| s |     < |     > |   * < |   * > | err < | err > |
```



```
| 0 | 0     | 0     | 0     | 0     | 0     | 0     |
| 1 | 0.085 | 0.485 | 0.312 | 0.564 | 0.227 | 0.078 |
| 2 | 0.5   | 0.692 | 0.67  | 0.763 | 0.171 | 0.071 |
| 3 | 0.742 | 0.811 | 0.803 | 0.845 | 0.061 | 0.034 |
| 4 | 0.836 | 0.87  | 0.864 | 0.889 | 0.028 | 0.02  |
| 5 | 0.886 | 0.914 | 0.901 | 0.922 | 0.016 | 0.008 |
| 6 | 0.924 | 0.949 | 0.932 | 0.952 | 0.008 | 0.003 |
| 7 | 0.956 | 0.979 | 0.961 | 0.981 | 0.005 | 0.002 |
| 8 | 0     | 0     | 0     | 0     | 0     | 0     |
Action-value function RMSE: 0.0739

Policy:
|           | 01      > | 02      > | 03      > | 04      > | 05      > | 06      > | 07      > |           |
Reaches goal 96.00%. Obtains an average return of 0.8672. Regret of 0.0000
```

Comparision





RESULT:

Thus the program to implement SARSA algorithm and comparing it with Monte-

Thus the program to implement SARSA algorithm and comparing it with Monte Carlo prediction is successfully completed.