RL Programming Assignment #1

ee21b124 Shivam, ee21b144 Sujal

March 2025

Note

1. Python version used is 3.11.9
2. In each environment, τ is decayed after every 100 episodes for Q-learning, and ϵ is decayed after every 100 episodes for SARSA.
3. Reward function used in each environment is same as what given its documentation.
4. Modifying the reward function to incorporate additional factors such as position, velocity, or other state variables can enhance the learning process by minimizing regret.
5. However, such changes may add complexity to interpreting performance metrics, such as episode return over time. Therefore, we are using the standard reward functions provided by environments like CartPole, MountainCar and MiniGrid to simplify the analysis and visualization of results, especially in toy environments where the primary focus is on algorithmic performance
6. The range of hyperparameter values for fine tuning was chosen based on what seems as a reasonable choice as taught in class.
7. All the code files, CSV tables etc. used are shared on github and a zipped folder is submitted on moodle.
8. Github link: `https://github.com/Sujal-Burad/DA6400_PA1`

1 Cartpole

1.1 SARSA

1.1.1 Code Snippets:

Agent code-

```python
def learn(self, num_episodes, num_steps):
    """
        Description: This function implements the SARSA algorithm.
        Args:
            num_episodes : The number of episodes.
            num_steps    : The number of steps.
        Returns:
            reward_list  : The list of rewards.
    """
    reward_list = []
    for episode in range(num_episodes):
        # initialize the environment
        state, _ = self.env.reset(seed=self.seed)
        # discretize the state
        state_discrete = discretize_state(state, self.bins)
        # initialize the reward
        total_reward = 0

        epsilon = self.epsilon

        if episode % 100 == 0:
            # Decaying epsilon
            epsilon = epsilon * self.epsilon_decay


        for step in range(num_steps):
            # get the action
            action = self.policy.get_action(epsilon, state_discrete)

            # take the action
            next_state, reward, done = self.env.step(action)[:3]


            # discretize the next state
            next_state_discrete = discretize_state(next_state, self.bins)
            # get the next_action
            next_state_action = self.policy.get_action(epsilon, next_state_discrete)
            # update the q-table
            td_error = self.compute_td_error(state_discrete, action, next_state_discrete, next_state_action, reward)
            self.update_q_table(state_discrete, action, td_error)
            # update the state
            state = next_state
            # update the state_discrete
            state_discrete = next_state_discrete
            # update the total reward
            total_reward += reward
            # check if the episode is finished
            if done or step == num_steps - 1:
                # print the total reward
                print("Episode: {}/{}, Total Reward: {}".format(episode + 1, num_episodes, total_reward))
                # append the total reward
                reward_list.append(total_reward)
                break
    return reward_list
```

Hyperparameter tuning code-

```python
# Define hyperparameters and their ranges
parameters_dict = {
    'algorithm': {'values': ['sarsa']},
    'alpha': {'min': 0.01, 'max': 1.0},
    'epsilon': {'min': 0.01, 'max': 0.2},
    'epsilon_decay': {'min': 0.9, 'max': 0.99},
    'gamma': {'value': 0.99},
    'num_episodes': {'value': 10000},
    'num_steps': {'value': 500},
    'num_bins': {'value': 20},
    'seed': {'values': [10, 20, 30, 40, 50]},
    'render': {'value': False}
}

# Create sweep configuration
sweep_config = {
    'method': 'bayes',
    'metric': {
        'name': 'Episode_return',
        'goal': 'maximize'
    },
    'parameters': parameters_dict
}

# Initialize the sweep
sweep_id = wandb.sweep(sweep_config, project="sarsa_cartpole_hyperparameter_finetuning_minimizing_regret")
```

1.1.2 Plots:

1) Plot for 5 runs(or seeds) as obtained in *wandb.ai* [Figure 1] and obtained plot for 5 runs(or seeds) consisting mean and variance[Figure 2]-

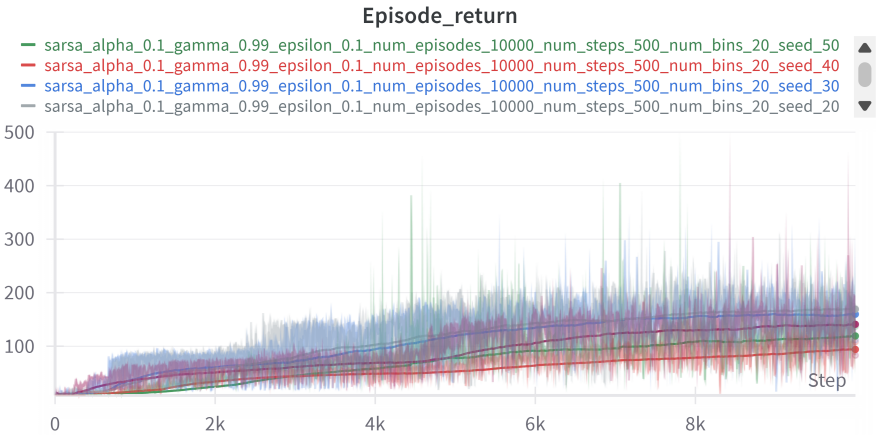

Figure 1: SARSA plot for Cartpole for 5 runs

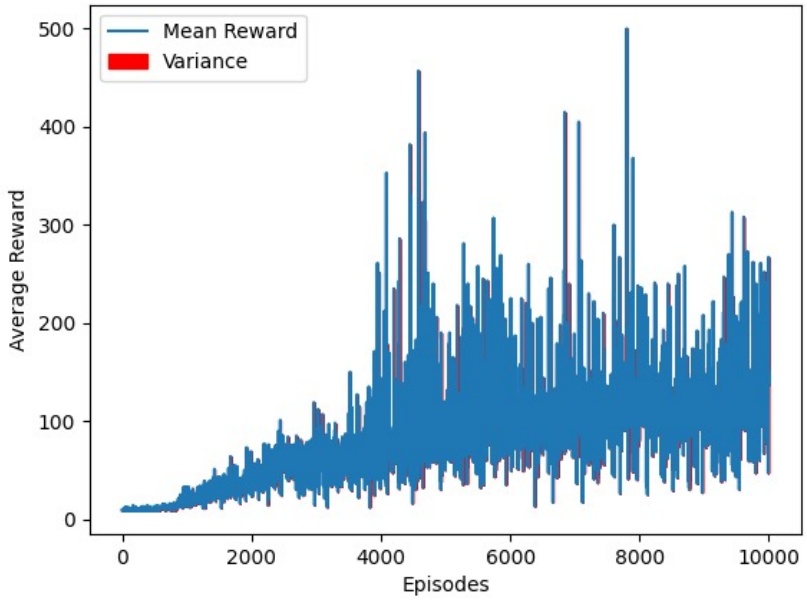

Figure 2: SARSA plot for Cartpole with mean and variance for 5 runs

2) Plot obtained by sweeping for hyperparameters in *wandb.ai*[Figure 3]-

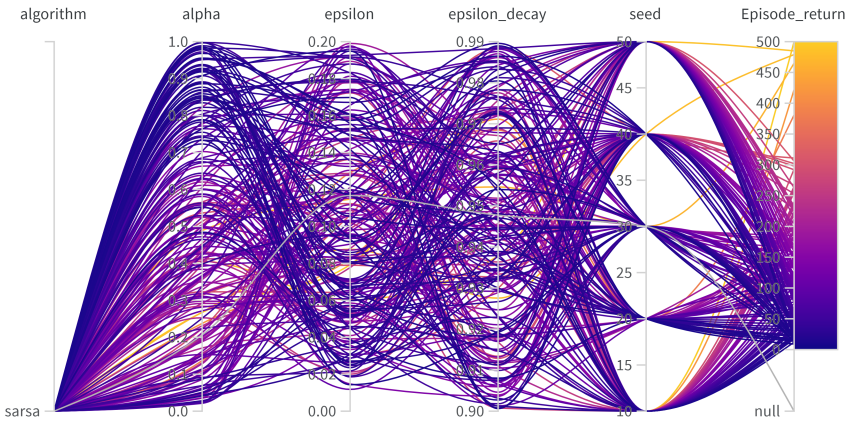

Figure 3: SARSA plot for Hypertuning

1.1.3 Top 3 best Hyperparameters:

3 best hyperparameters are obtained with the help of CSV file of the sweep plot available to download on *wandb.ai*. CSV file is sorted in descending order to obtain following best hyperparameters:-

Alpha	Epsilon	Epsilon Decay	Seed	Episode Return
0.267192	0.079249	0.972740	10	500
0.405514	0.060197	0.937756	50	485
0.267896	0.087639	0.927528	40	479

1.2 Q-Learning

1.2.1 Code Snippets:

Agent code-

```python
def learn(self, num_episodes, num_steps):
    """
        Description: This function implements the Q-Learning Procedure.
        Args:
            num_episodes : The number of episodes.
            num_steps    : The number of steps.
        Returns:
            reward_list  : The list of rewards.
    """
    reward_list = []
    for episode in range(num_episodes):
        # initialize the environment
        state, _ = self.env.reset(seed=self.seed)
        # discretize the state
        state_discrete = discretize_state(state, self.bins)
        # initialize the reward
        total_reward = 0

        tau = self.tau

        if episode % 100 == 0:
            # Decaying tau
            tau = tau * self.tau_decay


        for step in range(num_steps):
            # get the action
            action = self.policy.get_action(tau, state_discrete)

            # take the action
            next_state, reward, done = self.env.step(action)[:3]
            # discretize the next state
            next_state_discrete = discretize_state(next_state, self.bins)
            # update the q-table
            td_error = self.compute_td_error(state_discrete, action, next_state_discrete, reward)
            self.update_q_table(state_discrete, action, td_error)
            # update the state
            state = next_state
            # update the state_discrete
            state_discrete = next_state_discrete
            # update the total reward
            total_reward += reward
            # check if the episode is finished
            if done or step == num_steps - 1:
                # print the total reward
                print("Episode: {}/{}, Total Reward: {}".format(episode + 1, num_episodes, total_reward))
                # append the total reward
                reward_list.append(total_reward)
                break
    return reward_list
```

Hyperparameter tuning code-

```python
# Define hyperparameters and their ranges
parameters_dict = {
    'algorithm': {'values': ['q_learning']},
    'alpha': {'min': 0.01, 'max': 1.0},
    'tau': {'min': 0.5, 'max': 4.0},
    'tau_decay': {'min': 0.9, 'max': 0.99},
    'gamma': {'value': 0.99},
    'num_episodes': {'value': 10000},
    'num_steps': {'value': 500},
    'num_bins': {'value': 20},
    'seed': {'values': [10, 20, 30, 40, 50]},
    'render': {'value': False}
}

# Create sweep configuration
sweep_config = {
    'method': 'bayes',
    'metric': {
        'name': 'Episode_return',
        'goal': 'maximize'
    },
    'parameters': parameters_dict
}

# Initialize the sweep
sweep_id = wandb.sweep(sweep_config, project="q_learning_cartpole_hyperparameter_finetuning_minimizing_regret")
```

1.2.2 Plots:

1) Plot for 5 runs(or seeds) as obtained in *wandb.ai* [Figure 4] and obtained plot for 5 runs(or seeds) consisting mean and variance[Figure 5]-

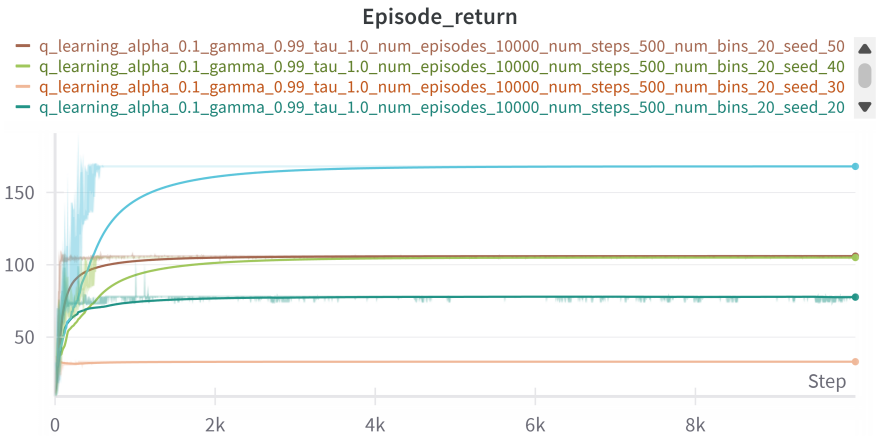

Figure 4: Q-Learning plot for Cartpole for 5 runs

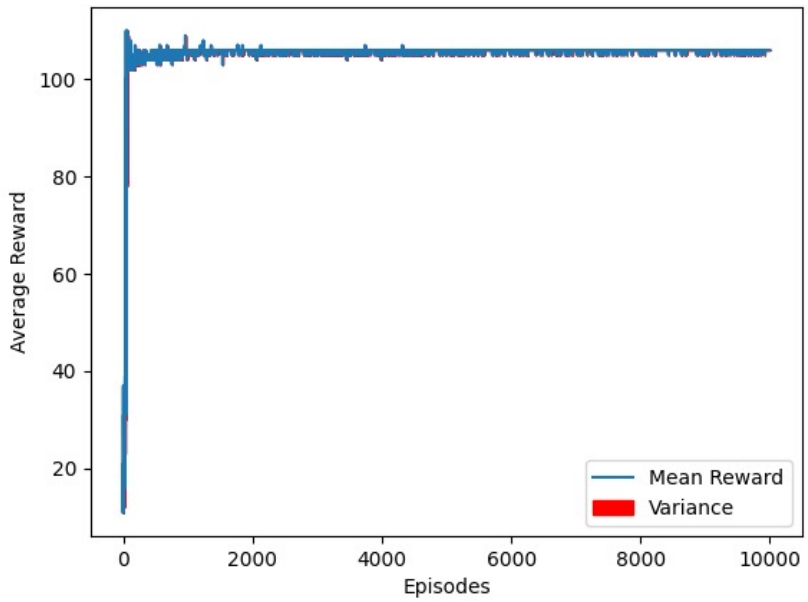

Figure 5: Q-Learning plot for Cartpole with mean and variance for 5 runs

2) Plot obtained by sweeping for hyperparameters in *wandb.ai*[Figure 6]-

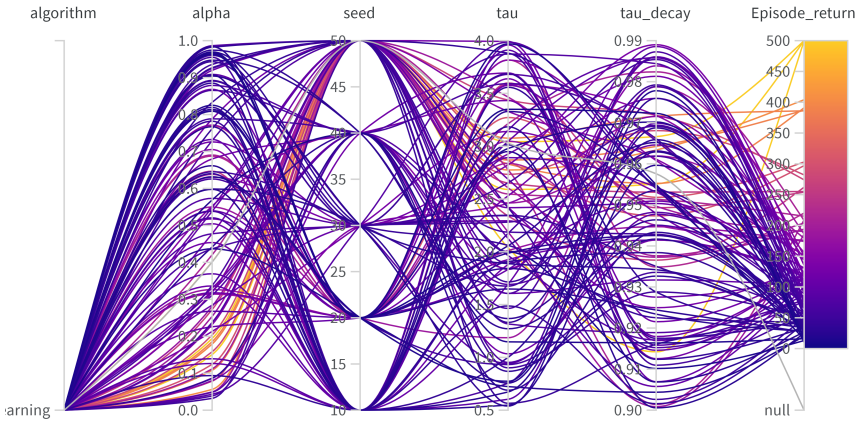

Figure 6: Q-Learning plot for Hypertuning

1.2.3 Top 3 best Hyperparameters:

Following the same steps as used previously:-

Alpha	Seed	Tau	Tau Decay	Episode Return
0.150538	50	3.042758	0.966632	500
0.194010	50	2.038783	0.914212	500
0.401267	50	2.590263	0.955087	500

1.3 Inferences and Conjectures

- In Q-learning, reward value stabilizes for lesser number of episodes as compared to SARSA.
- Even after stabilizing, in Q-learning, reward values varies less from stable value because it starts exploitation from very beginning. But in SARSA, variation is high because exploration doesn't fade out with episodes.

2 Mountain Car

2.1 SARSA

2.1.1 Code Snippets:

Agent code-

```python
def learn(self, num_episodes, num_steps):
    """
        Description: This function implements the SARSA algorithm.
        Args:
            num_episodes : The number of episodes.
            num_steps    : The number of steps.
        Returns:
            reward_list  : The list of rewards.
    """
    reward_list = []
    for episode in range(num_episodes):
        # initialize the environment
        state, _ = self.env.reset(seed=self.seed)
        # discretize the state
        state_discrete = discretize_state(state, self.bins)
        # initialize the reward
        total_reward = 0

        epsilon = self.epsilon

        if episode % 100 == 0:
            # Decaying epsilon
            epsilon = epsilon * self.epsilon_decay


        for step in range(num_steps):
            # get the action
            action = self.policy.get_action(epsilon, state_discrete)

            # take the action
            next_state, reward, done = self.env.step(action)[:3]

            # discretize the next state
            next_state_discrete = discretize_state(next_state, self.bins)
            # get the next_action
            next_state_action = self.policy.get_action(epsilon, next_state_discrete)
            # update the q-table
            td_error = self.compute_td_error(state_discrete, action, next_state_discrete, next_state_action, reward)
            self.update_q_table(state_discrete, action, td_error)
            # update the state
            state = next_state
            # update the state_discrete
            state_discrete = next_state_discrete
            # update the total reward
            total_reward += reward
            # check if the episode is finished
            if done or step == num_steps - 1:
                # print the total reward
                print("Episode: {}/{}, Total Reward: {}".format(episode + 1, num_episodes, total_reward))
                # append the total reward
                reward_list.append(total_reward)
                break
    return reward_list
```

Hyperparameter tuning code-

```python
# Define hyperparameters and their ranges
parameters_dict = {
    'algorithm': {'values': ['sarsa']},
    'alpha': {'min': 0.01, 'max': 1.0},
    'epsilon': {'min': 0.01, 'max': 0.2},
    'epsilon_decay': {'min': 0.9, 'max': 0.99},
    'gamma': {'value': 0.99},
    'num_episodes': {'value': 10000},
    'num_steps': {'value': 200},
    'num_bins': {'value': 20},
    'seed': {'values': [10, 20, 30, 40, 50]},
    'render': {'value': False}
}

# Create sweep configuration
sweep_config = {
    'method': 'bayes',
    'metric': {
        'name': 'Episode_return',
        'goal': 'maximize'
    },
    'parameters': parameters_dict
}

# Initialize the sweep
sweep_id = wandb.sweep(sweep_config, project="sarsa_mountaincar_hyperparameter_finetuning_final_minimizing_regret")
```

2.1.2 Plots:

1) Plot for 5 runs(or seeds) as obtained in *wandb.ai* [Figure 7] and obtained plot for 5 runs(or seeds) consisting mean and variance[Figure 8]-

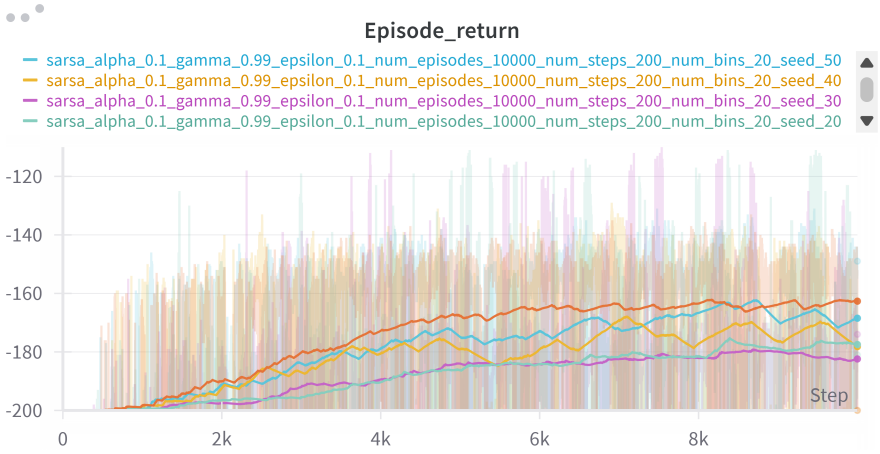

Figure 7: SARSA plot for Mountain Car for 5 runs

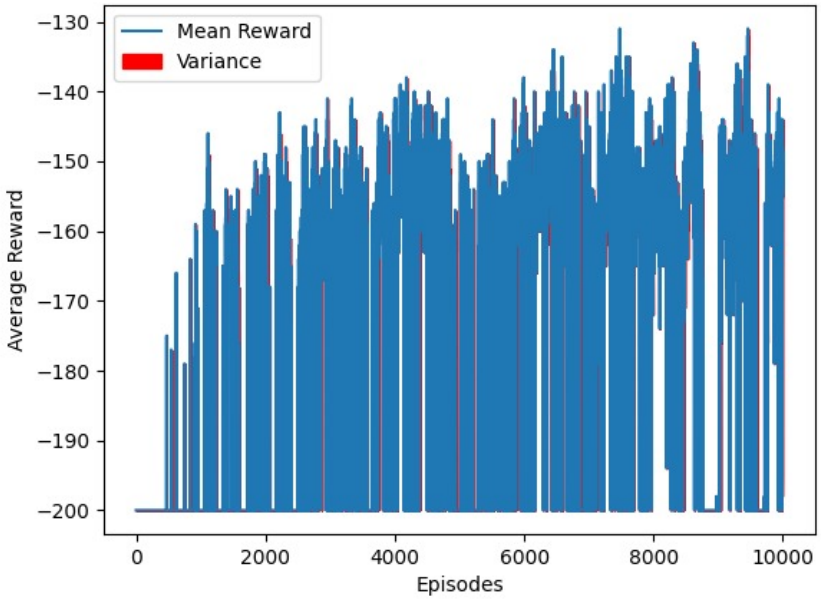

Figure 8: SARSA plot for Mountain Car with mean and variance for 5 runs

2) Plot obtained by sweeping for hyperparameters in *wandb.ai*[Figure 9]-

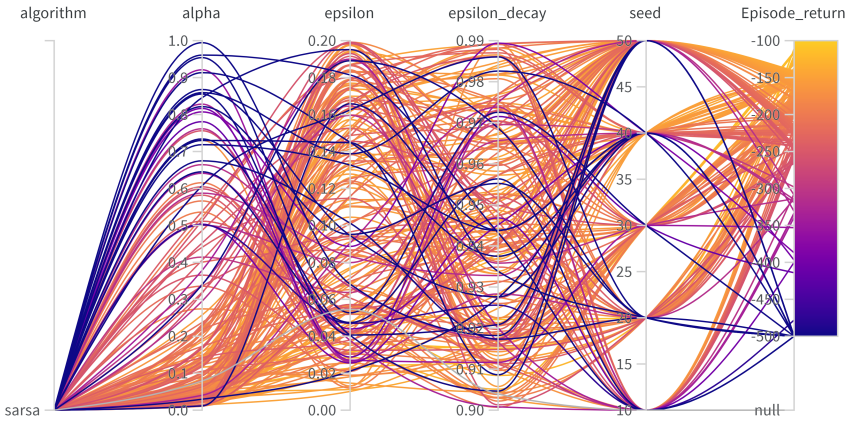

Figure 9: SARSA plot for Hypertuning

2.1.3 Top 3 best Hyperparameters:

Following are the 3 best hyperparameters obtained by sorting sweep CSV file in descending order:-

Alpha	Epsilon	Epsilon Decay	Seed	Episode Return
0.038301	0.052185	0.922067	20	−111
0.048709	0.022737	0.912021	30	−115
0.061261	0.031571	0.916006	30	−117

2.2 Q-Learning

2.2.1 Code Snippets:

Agent code-

```python
def learn(self, num_episodes, num_steps):
    """
        Description: This function implements the Q-Learning Procedure.
        Args:
            num_episodes : The number of episodes.
            num_steps    : The number of steps.
        Returns:
            reward_list  : The list of rewards.
    """
    reward_list = []
    for episode in range(num_episodes):
        # initialize the environment
        state, _ = self.env.reset(seed=self.seed)
        # discretize the state
        state_discrete = discretize_state(state, self.bins)
        # initialize the reward
        total_reward = 0

        tau = self.tau

        if episode % 100 == 0:
            # Decaying tau
            tau = tau * self.tau_decay


        for step in range(num_steps):
            # get the action
            action = self.policy.get_action(tau, state_discrete)

            # take the action
            next_state, reward, done = self.env.step(action)[:3]

            # discretize the next state
            next_state_discrete = discretize_state(next_state, self.bins)
            # update the q-table
            td_error = self.compute_td_error(state_discrete, action, next_state_discrete, reward)
            self.update_q_table(state_discrete, action, td_error)
            # update the state
            state = next_state
            # update the state_discrete
            state_discrete = next_state_discrete
            # update the total reward
            total_reward += reward
            # check if the episode is finished
            if done or step == num_steps - 1:
                # print the total reward
                print("Episode: {}/{}, Total Reward: {}".format(episode + 1, num_episodes, total_reward))
                # append the total reward
                reward_list.append(total_reward)
                break
    return reward_list
```

Hyperparameter tuning code-

```python
# Define hyperparameters and their ranges
parameters_dict = {
    'algorithm': {'values': ['q_learning']},
    'alpha': {'min': 0.01, 'max': 1.0},
    'tau': {'min': 0.5, 'max': 4.0},
    'tau_decay': {'min': 0.9, 'max': 0.99},
    'gamma': {'value': 0.99},
    'num_episodes': {'value': 10000},
    'num_steps': {'value': 200},
    'num_bins': {'value': 20},
    'seed': {'values': [10, 20, 30, 40, 50]},
    'render': {'value': False}
}

# Create sweep configuration
sweep_config = {
    'method': 'bayes',
    'metric': {
        'name': 'Episode_return',
        'goal': 'maximize'
    },
    'parameters': parameters_dict
}

# Initialize the sweep
sweep_id = wandb.sweep(sweep_config, project="q_learning_mountaincar_hyperparameter_finetuning_final_minimizing_regret")
```

2.2.2 Plots:

1) Plot for 5 runs(or seeds) as obtained in *wandb.ai* [Figure 10] and obtained plot for 5 runs(or seeds) consisting mean and variance[Figure 11]-

Figure 10: Q-Learning plot for Mountain Car for 5 runs

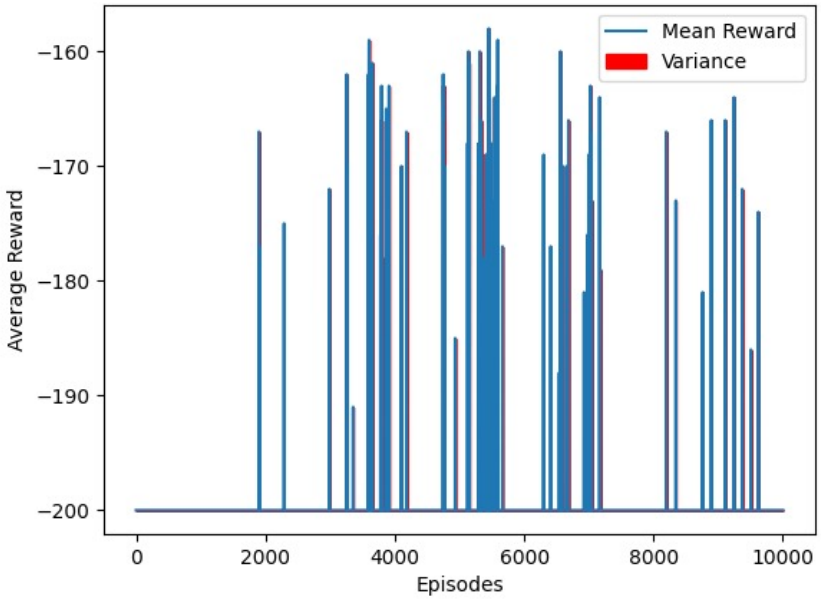

Figure 11: Q-Learning plot for Mountain Car with mean and variance for 5 runs

2) Plot obtained by sweeping for hyperparameters in *wandb.ai*[Figure 12]-

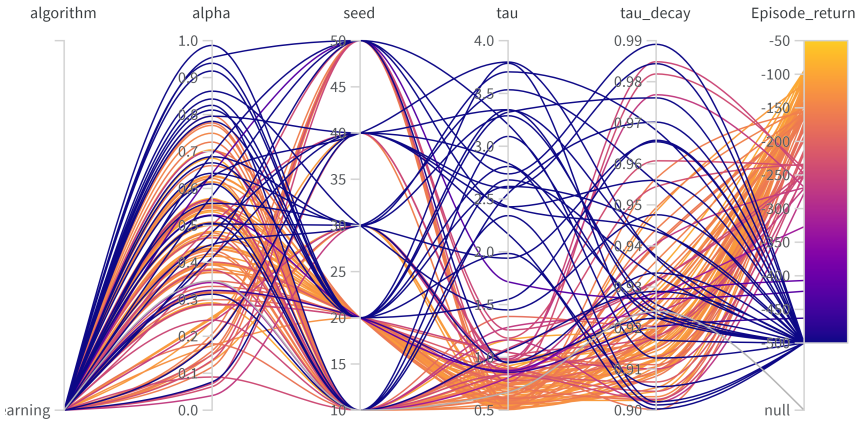

Figure 12: Q-Learning plot for Hypertuning

2.2.3 Top 3 best Hyperparameters:

Best 3 hyperparameters are:-

Alpha	Seed	Tau	Tau Decay	Episode Return
0.572766	10	0.942309	0.922622	−94
0.681833	10	0.669324	0.933374	−95
0.704362	10	0.880093	0.928751	−95

2.3 Inferences and Conjectures

- Reward of Q-learning indeed stabilizes at lesser number of episodes as compared to SARSA but it shows large variation after a number of episodes beacuse of overestimated Q-values from its off-policy max operator.
- This post-stabilization variance was not prominent in CartPole experiments, where both algorithms showed similar asymptotic stability. This suggests that such variance is not a universal trait but rather an environment-dependent phenomenon that may emerge in sparse-reward tasks (e.g., Mountain Car) or under extended training regimes.

3 MiniGrid

A plot for **total steps for each episode**, which tells the number of steps for which the episode ran, is also added.

The total steps are positively correlated with the episodic return.

Note:

- **Number of bins** is not needed here as the state space is already discrete.
- **Number of steps** is not needed here as the environment gets terminated when it reaches a certain number of steps. (Default `max_steps = 100`)

3.1 SARSA

3.1.1 Code Snippets:

Agent code-

```
def learn(self, num_episodes):
    """
        Description: This function implements the SARSA algorithm.
        Args:
            num_episodes : The number of episodes.
        Returns:
            reward_list  : The list of rewards.
    """
    reward_list = []
    total_steps_list = []
    for episode in range(num_episodes):
        # initialize the environment
        self.env.reset(seed=self.seed)

        terminated = False # When reached the goal green square or collided with an obstacle
        truncated = False # When max_steps have been done

        # initialize the reward
        total_reward = 0
        epsilon = self.epsilon

        if episode % 100 == 0:
            # Decaying epsilon
            epsilon = epsilon * self.epsilon_decay

        steps = 0


        while not (terminated or truncated):
            steps += 1
            position = (self.env.agent_pos[0] - 1) * 3 + (self.env.agent_pos[1] - 1) # 0 to 9
            direction = self.env.agent_dir
            front_cell = self.env.grid.get(*self.env.front_pos)
            not_clear = front_cell and front_cell.type != 'goal'
            if not_clear:
                immediate_cell = 1
            else:
                immediate_cell = 0

            # get the action
            action = self.policy.get_action(epsilon, (position, direction, immediate_cell))
            # take the action
            next_state, reward, terminated, truncated, _ = self.env.step(action)

            next_position = (self.env.agent_pos[0] - 1) * 3 + (self.env.agent_pos[1] - 1) # 0 to 9
            next_direction = self.env.agent_dir
            next_front_cell = self.env.grid.get(*self.env.front_pos)
            next_not_clear = next_front_cell and next_front_cell.type != 'goal'
            if not_clear:
                next_immediate_cell = 1
            else:
                next_immediate_cell = 0

            # print("next_state = ", next_state, type(next_state))
            # update the q table
            next_state_action = self.policy.get_action(epsilon, (next_position, next_direction, next_immediate_cell))
            td_error = self.compute_td_error((position, direction, immediate_cell), action, (next_position, next_direction, next_immediate_cell), next_state_action, reward)
            self.update_q_table((position, direction, immediate_cell), action, td_error)

            # update the total reward
            total_reward += reward
            # check if the episode is finished
            if terminated or truncated:
                # print the total reward
                print("Episode: {}/{}, Total Reward: {}, Total steps{}".format(episode + 1, num_episodes, total_reward, steps))
                # append the total reward
                reward_list.append(total_reward)
                total_steps_list.append(steps)
                break

    return reward_list, total_steps_list
```

Hyperparameter tuning code-

```python
# Define hyperparameters and their ranges
parameters_dict = {
    'algorithm': {'values': ['sarsa']},
    'alpha': {'min': 0.01, 'max': 1.0},
    'epsilon': {'min': 0.01, 'max': 0.2},
    'epsilon_decay': {'min': 0.9, 'max': 0.99},
    'gamma': {'value': 0.99},
    'num_episodes': {'value': 10000},
    'seed': {'values': [10, 20, 30, 40, 50]},
    'render': {'value': False}
}

# Create sweep configuration
sweep_config = {
    'method': 'bayes',
    'metric': {
        'name': 'Episode_return',
        'goal': 'maximize',
        'name': 'Total_steps',
        'goal': 'maximize'
    },
    'parameters': parameters_dict
}
# Initialize the sweep
sweep_id = wandb.sweep(sweep_config, project="sarsa_minigrid_hyperparameter_finetuning_final_minimizing_regret")
```

3.1.2 Plots:

1) Plot for 5 runs(or seeds) as obtained in *wandb.ai* is not legible thus here only the obtained plot for 5 runs(or seeds) consisting of mean and variance[Figure 13] is included-

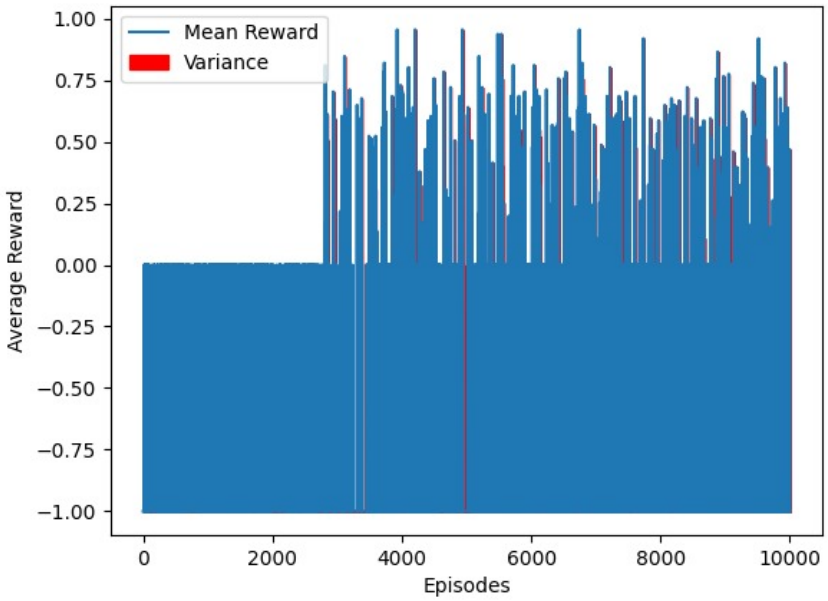

Figure 13: SARSA plot for MiniGrid with mean and variance for 5 runs

Also, the plot of total steps for each episode with episodes consisting of mean and variance for 5 runs is[Figure 14]-

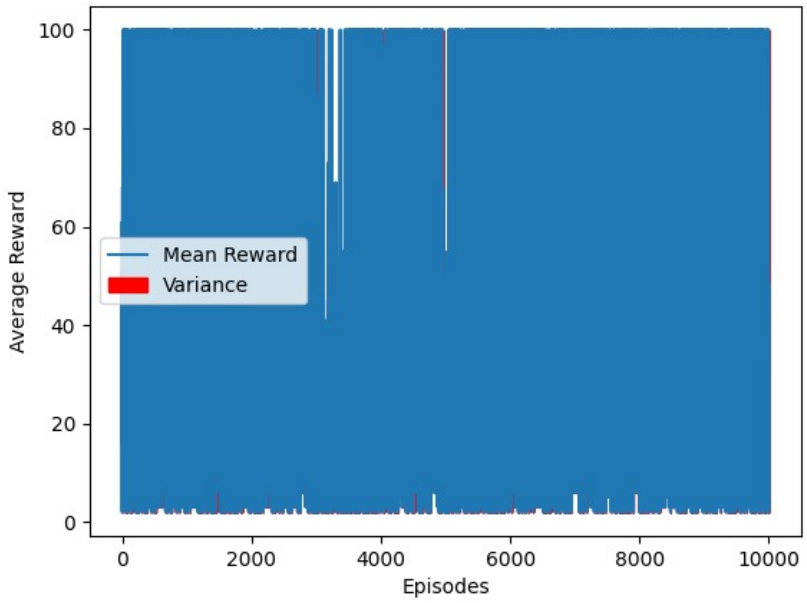

Figure 14: SARSA plot of total steps for each episode for MiniGrid with mean and variance for 5 runs

Please note that in above plot, it is Total number of steps instead of Average Reward in y-axis, apologize for typing error.

2) Plot obtained by sweeping for hyperparameters in *wandb.ai*[Figure 15]-

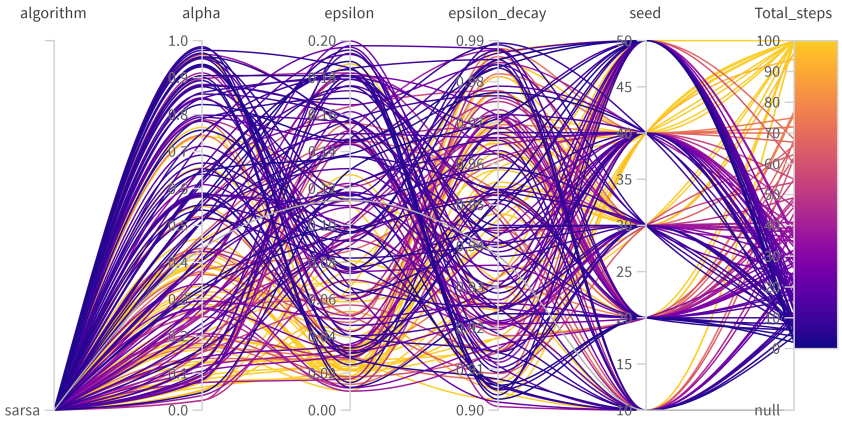

Figure 15: SARSA plot for Hypertuning

Since the total number of steps are positively correlated with the episodic return, sweep plot for it is as above and sweep plot for episodic return will be similar, thus not provided here.

3.1.3 Top 3 best Hyperparameters:

The best hyperparametrs obtained by sorting the CSV table are:-

Alpha	Epsilon	Epsilon Decay	Seed	Total Steps
0.074820	0.018512	0.984708	40	100
0.740075	0.111993	0.906773	40	100
0.312300	0.017331	0.972141	40	100

3.2 Q-Learning

3.2.1 Code Snippets:

Agent code-

```python
def learn(self, num_episodes):
    """
        Description: This function implements the Q-Learning Procedure.
        Args:
            num_episodes : The number of episodes.
        Returns:
            reward_list  : The list of rewards.
            total_steps_list : The list of steps taken.
    """
    reward_list = []
    total_steps_list = []
    for episode in range(num_episodes):
        # initialize the environment
        self.env.reset(seed=self.seed)

        terminated = False # When reached the goal green square or collided with an obstacle
        truncated = False # When max_steps have been done

        # initialize the reward
        total_reward = 0
        tau = self.tau

        if episode % 100 == 0:
            # Decaying tau
            tau = tau * self.tau_decay

        steps = 0


        while not (terminated or truncated):
            steps += 1

            position = (self.env.agent_pos[0] - 1) * 3 + (self.env.agent_pos[1] - 1) # 0 to 9
            direction = self.env.agent_dir
            front_cell = self.env.grid.get(*self.env.front_pos)
            not_clear = front_cell and front_cell.type != 'goal'
            if not_clear:
                immediate_cell = 1
            else:
                immediate_cell = 0

            # get the action
            action = self.policy.get_action(tau, (position, direction, immediate_cell))
            # take the action
            next_state, reward, terminated, truncated, _ = self.env.step(action)

            next_position = (self.env.agent_pos[0] - 1) * 3 + (self.env.agent_pos[1] - 1) # 0 to 9
            next_direction = self.env.agent_dir
            next_front_cell = self.env.grid.get(*self.env.front_pos)
            next_not_clear = next_front_cell and next_front_cell.type != 'goal'
            if not_clear:
                next_immediate_cell = 1
            else:
                next_immediate_cell = 0

            # print("next_state = ", next_state, type(next_state))
            # update the q_table
            td_error = self.compute_td_error((position, direction, immediate_cell), action, (next_position, next_direction, next_immediate_cell), reward)
            self.update_q_table((position, direction, immediate_cell), action, td_error)

            # update the total reward
            total_reward += reward
            # check if the episode is finished
            if terminated or truncated:
                # print the total reward
                print("Episode: {}/{}, Total Reward: {}, Total steps{}".format(episode + 1, num_episodes, total_reward, steps))
                # append the total reward
                reward_list.append(total_reward)
                total_steps_list.append(steps)
                break

    return reward_list, total_steps_list
```

Hyperparameter tuning code-

```python
# Define hyperparameters and their ranges
parameters_dict = {
    'algorithm': {'values': ['q_learning']},
    'alpha': {'min': 0.01, 'max': 1.0},
    'tau': {'min': 0.5, 'max': 4.0},
    'tau_decay': {'min': 0.9, 'max': 0.99},
    'gamma': {'value': 0.99},
    'num_episodes': {'value': 10000},
    'seed': {'values': [10, 20, 30, 40, 50]},
    'render': {'value': False}
}

# Create sweep configuration
sweep_config = {
    'method': 'bayes',
    'metric': {
        'name': 'Episode_return',
        'goal': 'maximize',
        'name': 'Total_steps',
        'goal': 'maximize'
    },
    'parameters': parameters_dict
}

# Initialize the sweep
sweep_id = wandb.sweep(sweep_config, project="q_learning_minigrid_hyperparameter_finetuning_final_minimizing_regret")
```

3.2.2 Plots:

1) Since plot for 5 runs(or seeds) as obtained in *wandb.ai* is not legible thus the obtained plot for 5 runs(or seeds) consisting of mean and variance[Figure 16] is included-

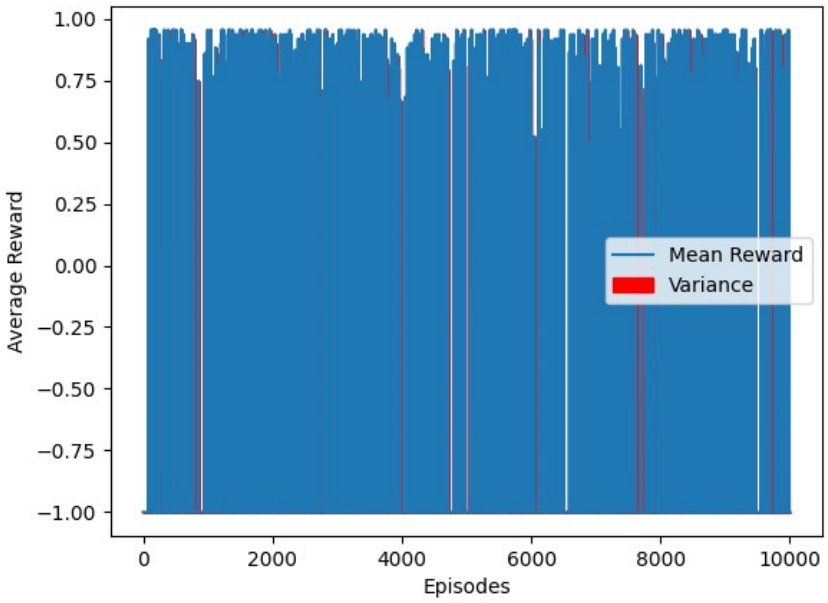

Figure 16: Q-Learning plot for MiniGrid with mean and variance for 5 runs

And, the plot of total steps for each episode with episodes consisting of mean and variance for 5 runs is[Figure 17]-

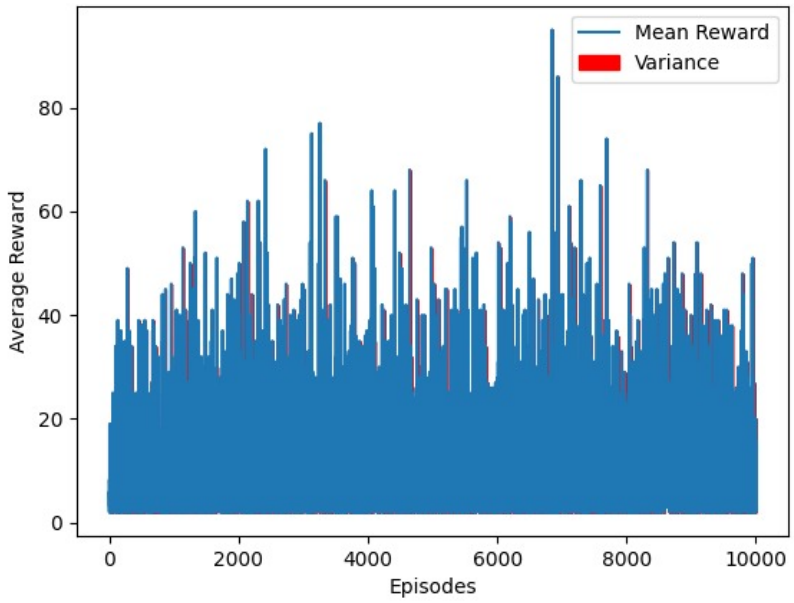

Figure 17: Q-Learning plot of total steps for each episode for MiniGrid with mean and variance for 5 runs

Please note that in above plot, it is Total number of steps instead of Average Reward in y-axis, apologize for typing error.

2) Plot obtained by sweeping for hyperparameters in *wandb.ai*[Figure 18]-

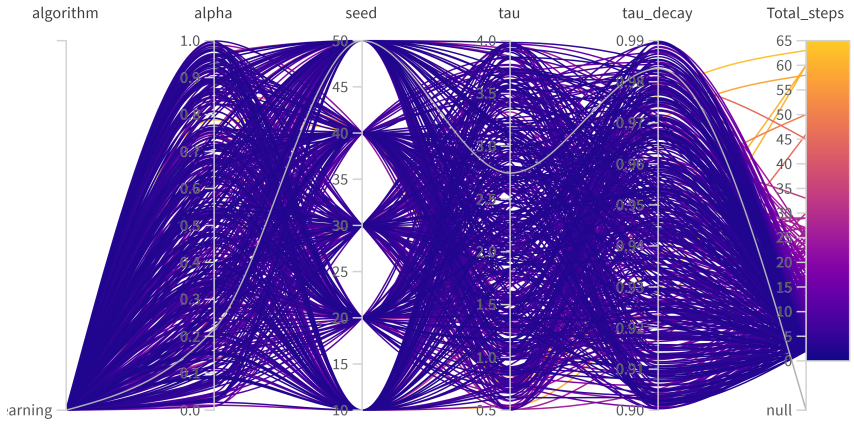

Figure 18: Q-Learning plot for Hypertuning

3.2.3 Top 3 best Hyperparameters:

The best hyperparameters are:-

Alpha	Tau	Tau Decay	Seed	Total Steps
0.751852	0.536355	0.929116	10	60
0.413186	0.823793	0.937002	40	60
0.250087	0.663323	0.970408	30	58

3.3 Inferences and Conjectures

- Similar observations as the last two environments can be drawn in this environment also.

4 *Wandb.ai* links

4.1 Cartpole plots

- `https://wandb.ai/sujal/cartpole_rl_experiment_q_learning_minimizing_regret?nw=nwusersujal`
- `https://wandb.ai/sujal/cartpole_rl_experiment_sarsa_minimizing_regret?nw=nwusersujal`
- `https://wandb.ai/sujal/sarsa_cartpole_hyperparameter_finetuning_minimizing_regret?nw=nwusersujal`
- `https://wandb.ai/sujal/q_learning_cartpole_hyperparameter_finetuning_minimizing_regret?nw=nwusersujal`

4.2 MountainCar plots

- https://wandb.ai/sujal/q_learning_mountaincar_hyperparameter_finetuning_final_minimizing_regret?nw=nwusersujal
- https://wandb.ai/sujal/mountaincar_rl_experiment_sarsa_minimizing_regret?nw=nwusersujal
- https://wandb.ai/sujal/mountaincar_rl_experiment_q_learning_minimizing_regret?nw=nwusersujal
- https://wandb.ai/sujal/sarsa_mountaincar_hyperparameter_finetuning_final_minimizing_regret?nw=nwusersujal

4.3 MiniGrid plots

- https://wandb.ai/sujal/q_learning_minigrid_hyperparameter_finetuning_final_minimizing_regret?nw=nwusersujal
- https://wandb.ai/sujal/sarsa_minigrid_hyperparameter_finetuning_final_minimizing_regret?nw=nwusersujal
- https://wandb.ai/sujal/minigrid_rl_experiment_sarsa_minimizing_regret?nw=nwusersujal
- https://wandb.ai/sujal/minigrid_rl_experiment_q_learning_minimizing_regret?nw=nwusersujal