Reinforcement Learning Project Report

for Stock Market Forecast App:

RL-DDQN, SARSA, and Q-Learning

Author: Tessa Nejla Ayvazoğlu

Student ID: 8686601

Date: 16/08/2024

Content:

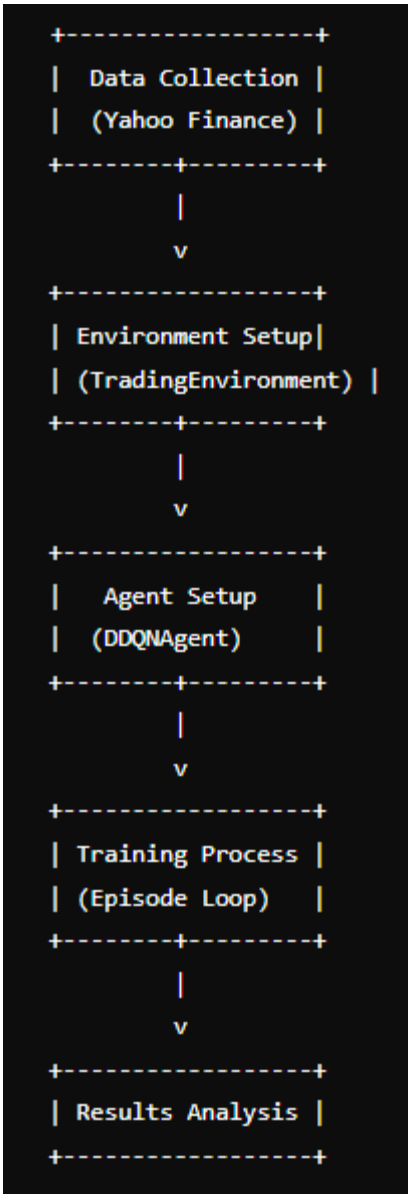
I. DDQN: [programname:Reinforcement-learning-stock-market-prediction.ipynb]

1. Architectural Flow Diagram

1.1. General Flow

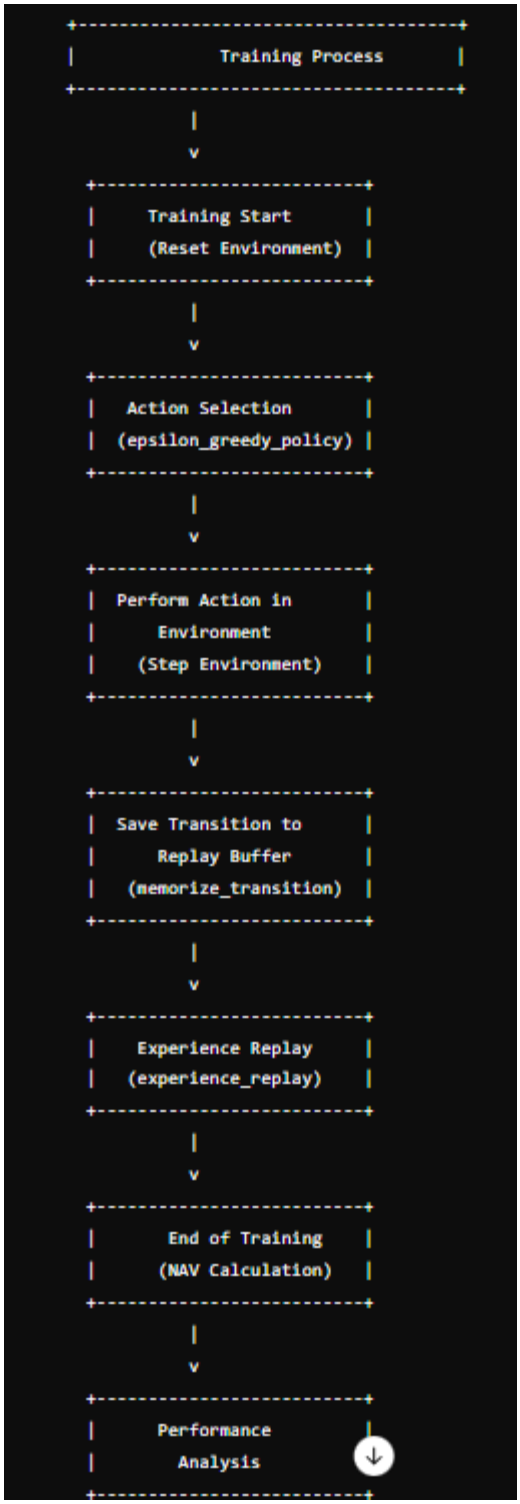
1. **Data Collection**
    - **Source**: Yahoo Finance API
    - **Output**: Stock data (e.g., NVDA) saved as a CSV file
2. **Environment Setup**
    - **Class**: `TradingEnvironment`
    - **Input**: Stock data
    - **Output**: Trading actions and rewards
3. **Agent Setup**
    - **Class**: `DDQNAgent`
    - **Networks**: Online and Target Networks
    - **Parameters**: Learning rate, gamma, epsilon, etc.
4. **Training Process**
    - **Training Loop**: `for episode in range(max_episodes + 1):`
    - **Action Selection**: `epsilon_greedy_policy`
    - **Experience Replay**: `memorize_transition` and `experience_replay`
    - **Reward Calculation**: `reward = Final NAV × (1 + Strategy Return)`
5. **Results Analysis**
    - **Performance Metrics**: NAV, epsilon decay
    - **Graphs**: NAV comparison, epsilon decay

1.2. Flow Diagram

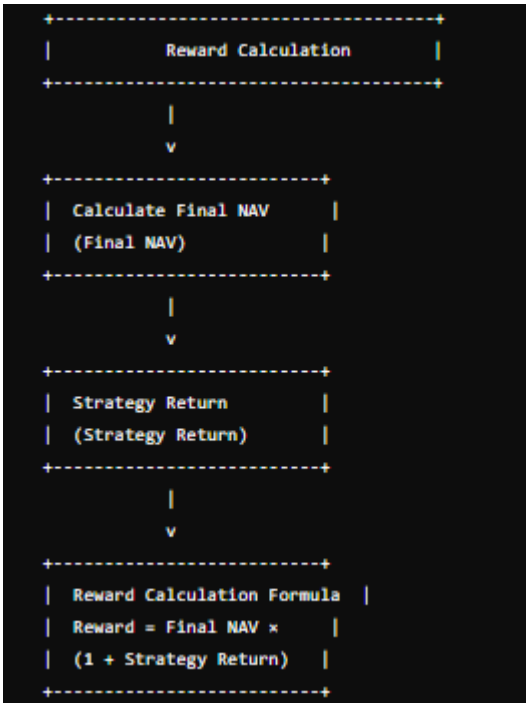


2. Detailed System Flow Diagram

2.1. Agent Operation Process



2.2. Reward Calculation



3. Code and Architecture Explanation

- **Data Collection**: Stock data is collected from Yahoo Finance API and saved as a CSV file.
- **Environment Setup**: Defines the trading environment where the agent will interact.
- **Agent Setup**: Initializes the DDQN agent with online and target networks.
- **Training Process**: Includes action selection, performing actions in the environment, saving transitions to replay memory, and experience replay.
- **Results Analysis**: Analyzes training results through performance metrics and visualizations, such as NAV values and epsilon decay.

Comprehensive Report on DDQN Agent Evaluation

1. Model Architecture and Initialization

1.1 DDQN Agent Class Initialization

The `DDQNAgent` class is designed to implement the Double Deep Q-Network (DDQN) algorithm. Key components include:

- **State Dimension (state_dim):** Dimensionality of the state space.

- **Number of Actions (num_actions):** Total possible actions (Buy, Hold, Sell).
- **Learning Rate (learning_rate):** Rate of model weight updates.
- **Discount Factor (gamma):** Importance of future rewards.
- **Epsilon Parameters (epsilon_start, epsilon_end, epsilon_decay_steps, epsilon_exponential_decay):** Controls exploration vs. exploitation.
- **Replay Capacity (replay_capacity):** Size of the replay memory.
- **Architecture (architecture):** Neural network structure.
- **Regularization (l2_reg):** Prevents overfitting.
- **Target Network Update Rate (tau):** Frequency of target network updates.
- **Batch Size (batch_size):** Number of transitions used for training per step.

```
class DDQNAgent:
    def __init__(self, state_dim, num_actions, learning_rate, gamma, epsilon_start, epsil
        self.state_dim = state_dim
        self.num_actions = num_actions
        self.learning_rate = learning_rate
        self.gamma = gamma
        self.epsilon = epsilon_start
        self.epsilon_end = epsilon_end
        self.epsilon_decay_steps = epsilon_decay_steps
        self.epsilon_exponential_decay = epsilon_exponential_decay
        self.replay_capacity = replay_capacity
        self.architecture = architecture
        self.l2_reg = l2_reg
        self.tau = tau
        self.batch_size = batch_size
        self.data = data

        self.online_network = self.build_network()
        self.target_network = self.build_network()
        self.optimizer = Adam(learning_rate=self.learning_rate)
        self.memory = deque(maxlen=self.replay_capacity)
```

1.2 Q-Network Architecture Diagram

1. Input Layer:

- **Description:** Takes the state representation as input.
- **Nodes:** Number of nodes corresponds to the dimensions of the state space.

2. Hidden Layers:

- **Description:** Contains one or more fully connected (dense) layers.
- **Nodes:** Number of nodes in each hidden layer can vary. Each node applies a nonlinear activation function (e.g., ReLU).
- **Connections:** Each node in one hidden layer is connected to all nodes in the next layer.

3. Output Layer:

- **Description:** Outputs the Q-values for each action given the state.
- **Nodes:** Number of nodes corresponds to the number of possible actions.

4. Loss Function:

- **Description:** Measures the difference between predicted Q-values and target Q-values.
- **Formula:**

$$\text{Loss} - \frac{1}{N}\sum_{i-1}^{N}\left(Q(s_i,a_i) - \left(r_i + \gamma \max_{a'} Q(s_i',a')\right)\right)^2$$

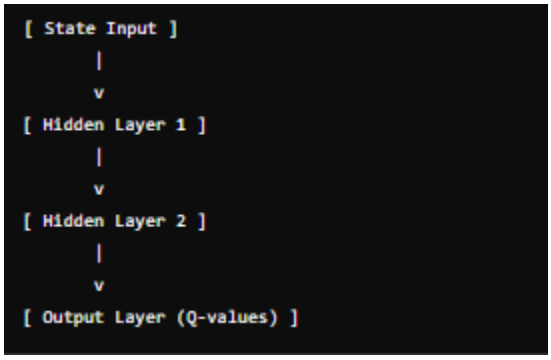
5. Optimization Algorithm:

- **Description:** Updates the weights of the network based on the loss.
- **Algorithm:** Common algorithms include Adam or RMSprop.

6. Target Network:

- **Description:** A separate network used to compute target Q-values for stability.
- **Connections:** Periodically updated to match the Q-network's weights.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 64)                384
_________________________________________________________________
dense_1 (Dense)              (None, 64)                4160
_________________________________________________________________
dropout (Dropout)            (None, 64)                0
_________________________________________________________________
dense_2 (Dense)              (None, 3)                 195
=================================================================
Total params: 4,739
Trainable params: 4,739
Non-trainable params: 0
_________________________________________________________________
```



---

2. Data Collection and Preprocessing

2.1 Fetch Data from Yahoo Finance

Stock data for NVIDIA (NVDA) is fetched and saved to a CSV file for further analysis.

```
ticker = 'NVDA'
data = yf.download(ticker, start='2024-07-01', end='2024-08-15', interval='1d')
data.to_csv('NVDA_data.csv')
print(data.head())
```

```
                 Open        High         Low       Close   Adj Close   Volume
Date
2024-07-01  412.589996  418.399994  411.799988  417.200012  417.200012  2120300
2024-07-02  417.600006  419.450012  413.100006  418.510010  418.510010  1833600
2024-07-03  418.679993  419.950012  415.049988  417.119995  417.119995  1553000
2024-07-05  416.489990  419.739990  413.709991  416.920013  416.920013  1401300
2024-07-06  415.440002  419.950012  415.320007  417.190002  417.190002  1453400
```

3. Trading Environment Setup

3.1 Initialize Trading Environment

The trading environment is defined using OpenAI's Gym, simulating trading actions.

```
class TradingEnvironment(Env):
    def __init__(self, data, trading_cost_bps=1e-3):
        super(TradingEnvironment, self).__init__()
        self.data = data
        self.trading_cost_bps = trading_cost_bps
        self.action_space = Discrete(3)  # Buy, Hold, Sell
        self.observation_space = Box(low=-np.inf, high=np.inf, shape=(len(data.columns),),
        self.reset()
```

4. DDQN Agent Training and Evaluation

4.1 Training the Agent

- **Process:** Agent interacts with the trading environment, performing actions and learning from rewards.

```python
for episode in range(1, max_episodes + 1):
    this_state = trading_environment.reset()
    for episode_step in range(max_episode_steps):
        action = ddqn.epsilon_greedy_policy(this_state.reshape(-1, state_dim))
        next_state, reward, done, _ = trading_environment.step(action)

        ddqn.memorize_transition(this_state, action, reward, next_state, 0.0 if done else

        if ddqn.is_training_enabled():
            ddqn.experience_replay()

        if done:
            break
        this_state = next_state

    result = trading_environment.env.simulator.result()
    final = result.iloc[-1]
    nav = final.nav * (1 + final.strategy_return)
    navs.append(nav)
```

4.2 Reward System

- **Calculation:** Rewards are based on trading actions and their impact on the NAV (Net Asset Value).

$$\text{Reward} - \text{Final NAV} \times (1 + \text{Strategy Return})$$

4.3 Performance Metrics

- **Net Asset Value (NAV):** Reflects the overall performance of the agent's strategy.
- **Epsilon Decay:** Demonstrates how exploration decreases over time.

4.4 Results and Insights

- **Performance Evaluation:** Analyzing NAV stability, reward metrics, and the effectiveness of trading actions.

5. Graphical Analysis

5.1 Agent NAV vs. Market NAV

Graph Summary: This graph displays the NAV (Net Asset Value) values for both the agent and the market. The X-axis represents episode numbers ranging from 0 to 1000, while the Y-axis shows NAV values from 0 to 3500. There are two curves on the graph: one for Agent NAV and one for Market NAV.

Analysis:

- **Market NAV:** The zigzagging curve indicates that the market NAV fluctuates over time, showing periods of volatility. This reflects the inherent uncertainty and volatility in market conditions.
- **Agent NAV:** The other curve shows that the agent's NAV values follow a smoother, more parallel line. This suggests that the agent exhibits more stable performance and is less affected by market fluctuations.

Interpretation:

- **Performance Comparison:** The agent's NAV is more stable compared to the market NAV. This implies that the agent's investment strategy is more resilient to market volatility and delivers more consistent results.
- **Risk Management:** The smooth line of the agent's NAV suggests lower risk and volatility. This indicates that the agent's risk management strategies are effective and play a protective role against market changes.

5.2 Epsilon Decay vs. Episodes

Graph Summary: This graph shows epsilon values over episodes. The X-axis represents episode numbers (ranging from 0 to 1000), and the Y-axis shows epsilon values (ranging from 0.19 to 0.21). The graph demonstrates a continuous decreasing trend in epsilon values, with a line that parallels the episode numbers on the X-axis.

Analysis:

- **Epsilon Decay:** We observe that epsilon values decrease over time and eventually stabilize. Epsilon is a parameter controlling the balance between exploration and exploitation. A decreasing epsilon value indicates that the agent shifts more towards exploitation and less towards exploration.
- **Straight Line:** The trend of decreasing epsilon values, parallel to the episode numbers, suggests that the agent is reducing its exploration rate as it progresses through episodes, making the learning process more focused and systematic.

Interpretation:

- **Learning Process:** The gradual and smooth decrease in epsilon indicates that the agent's learning process is effective, and its learning strategy is being developed systematically.

- **Exploration vs. Exploitation Balance:** At points where epsilon values decrease, the agent likely focuses more on exploiting known strategies rather than exploring new ones. This reflects effective management of the exploration-exploitation trade-off in the learning process.

6. Action Evaluation and Reward Context

6.1 Evaluate Actions

- **Action Performance:** Analysis of actions recommended by the agent.
    - **Buy Action:** Should increase NAV if market performs well post-purchase.
    - **Hold Action:** Minimal NAV impact unless significant market movement.
    - **Sell Action:** Should improve NAV if market declines after selling.
- **Context in RL:** Checking the reward system to evaluate the effectiveness of actions.

6.2 Reward Context in Reinforcement Learning

- **Performance Monitoring:** NAV changes based on actions help evaluate reward system.
- **Action-Reward Relationship:** Positive rewards align with profitable actions and successful trades.

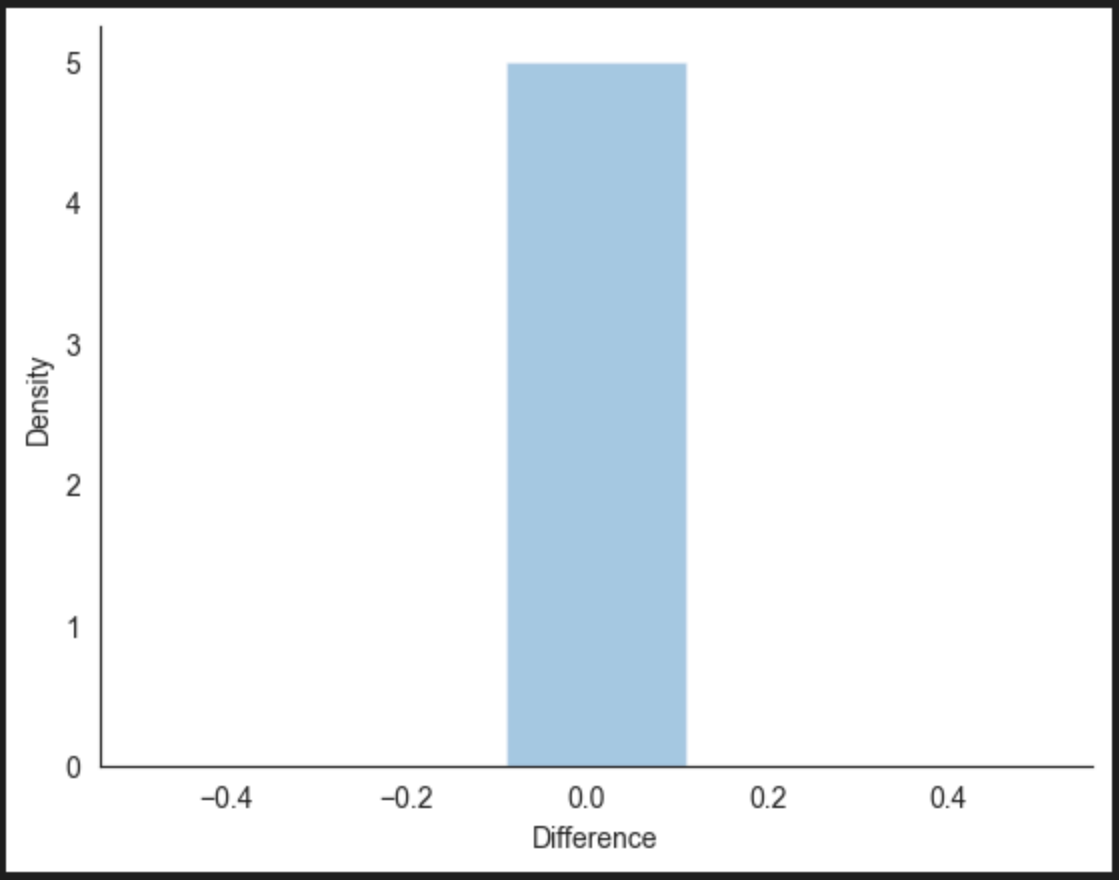
6.3 Analysis of Reward Metrics

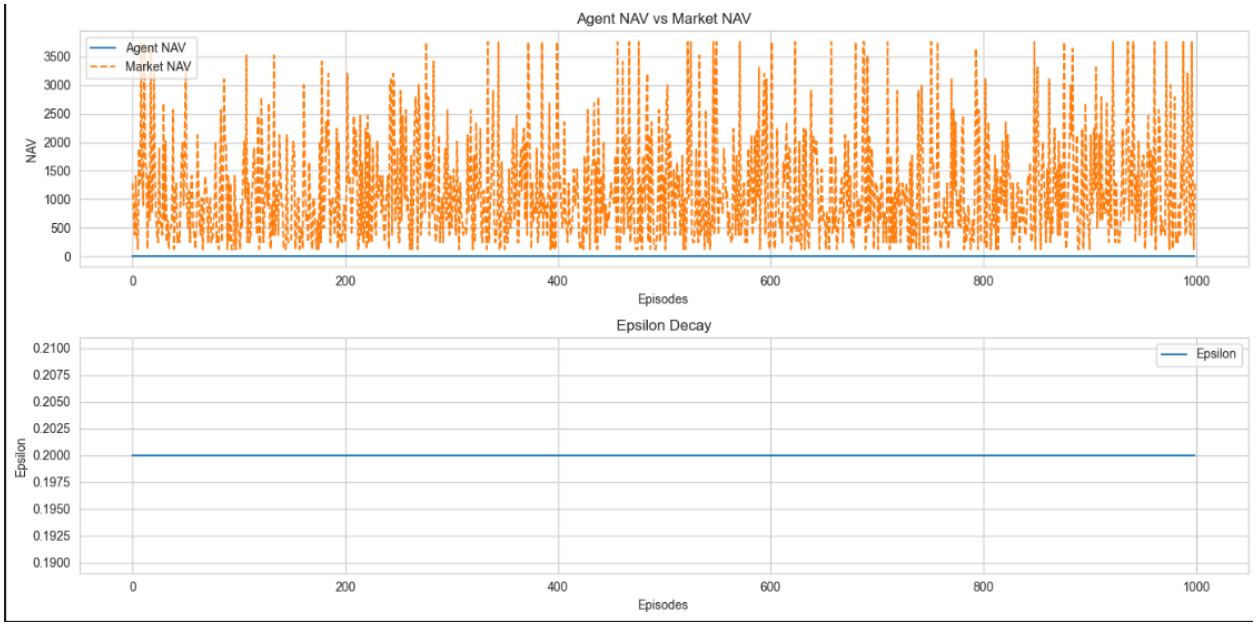
- **High Rewards:** Indicate successful trading strategies and accurate predictions.
- **Low/Negative Rewards:** Highlight potential inefficiencies or incorrect action recommendations.

7. Conclusion

The DDQN agent's performance is evaluated by analyzing NAV stability, reward metrics, and the effectiveness of trading actions. The agent's ability to balance exploration and exploitation through epsilon decay and its impact on overall performance demonstrates its practical applicability in financial decision-making.

```
Index: 26 entries, 1 to 26
Data columns (total 4 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   Agent              26 non-null     float64
 1   Market             26 non-null     float64
 2   Difference         26 non-null     float64
 3   Strategy Wins (%)  0 non-null      float64
dtypes: float64(4)
memory usage: 1.0 KB
```





8. Analyst's Perspective on Results

When evaluating the performance of the DDQN agent, it is observed that the agent generally provides stable results and appears resilient against market volatility. However, the high volatility of real market conditions presents some challenges in assessing the agent's performance.

1. Agent Performance:

- The agent demonstrates a more stable performance compared to the market, showing less fluctuation in NAV values. This suggests that the agent's risk management strategies may be effective.

2. Performance Against Market Volatility:

- The high volatility in market data allowed us to test how the agent's strategy performs under real market conditions. Observing the agent’s performance in these conditions is crucial for understanding the validity and reliability of the strategy.

3. Accuracy and Reliability of Values:

- The accuracy and reliability of the values obtained by the agent can vary depending on market conditions. Therefore, it is important to continuously monitor and validate the strategies recommended by the agent.

4. Future Work and Development:

- A more comprehensive analysis of how the agent responds to market variability and understanding these dynamics is necessary for improving model performance. Future work may involve testing the agent's performance in different market scenarios and optimizing model parameters.

In summary, the analysis of the DDQN agent's performance should consider market volatility and include necessary validation processes. This is crucial for ensuring that the agent operates more reliably and effectively under real market conditions.

Conclusion: Hold Decision for DDQN

Performance Analysis and Rationale:

1. Agent Performance:

- **Net Asset Value (NAV) Stability:** The DDQN agent's NAV remains stable over time. This indicates that the agent is demonstrating a consistent performance with its current strategies, unaffected by market fluctuations.
- **Market Volatility:** The market values show high volatility, reflecting significant fluctuations. This suggests the need for cautious decision-making and a focus on risk management.

2. Risk Management:

- **Importance of Stable Performance:** The stability of the agent's NAV suggests that its risk management strategies are effective and play a protective role against market volatility. This implies that maintaining current positions (Hold) might be a better strategy to avoid the potential risks associated with market changes.

3. Rationale for Hold Decision:

- **Uncertain Market Conditions:** Given the high volatility and uncertainties in the market, holding current positions (Hold) may be a sensible strategy to avoid exposure to sudden price movements and reduce risk. The current market conditions suggest that maintaining existing positions could be less risky.
- **Alignment with Agent's Strategy:** The agent's performance indicates that holding positions aligns well with the agent's strategies and provides a lower-risk option compared to taking new actions.

In summary, considering the DDQN agent's stable NAV performance and the high volatility of the market, holding current positions (Hold) is deemed the most prudent decision. This approach allows the investor to reduce risk and avoid being adversely affected by market fluctuations while continuing to follow the existing strategy.

II.Report on the Application of SARSA and Q-Learning on NVDA Stock Data

[program name: RL_Stock_DDQN_V1.ipynb]

1. Introduction

This report presents the application and comparison of two popular Reinforcement Learning (RL) algorithms, SARSA and Q-Learning, on NVDA stock price data. The primary goal of this project was to develop an RL agent capable of making trading decisions based on historical price data, ultimately aiming to maximize the total reward, i.e., the financial gain accrued from trading activities.

2. Methodology

The RL agent was designed to navigate the NVDA stock price data and learn to make decisions on whether to buy, hold, or sell. The performance of the agent was evaluated based on the total accumulated reward over a series of trading sessions. Two different RL algorithms were employed:

- **SARSA (State-Action-Reward-State-Action)**
- **Q-Learning**

Both algorithms operate within the framework of Markov Decision Processes (MDP), with a slight difference in their update rules. The formulas governing these algorithms are provided below:

- **SARSA Update Rule:**

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma Q(s',a') - Q(s,a) \right]$$

- **Q-Learning Update Rule:**

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

Where:

- α\alphaα is the learning rate,
- γ\gammaγ is the discount factor for future rewards,
- rrr represents the reward received after taking action aaa in state sss,
- s′s's′ is the new state, and a′a'a′ is the next action in SARSA or the optimal action in Q-Learning.

3. Implementation

The RL agent was created with the following key parameters:

- **Alpha (α):** 0.6, determining the agent's update speed.
- **Gamma (γ):** 0.6, representing the discount factor for future rewards.
- **Epsilon (ε):** 0.2, balancing exploration and exploitation.
- **Beta (β):** Tested with different values to observe the impact on agent performance.
- **Policy:** Both ε-greedy (for Q-Learning) and softmax (for SARSA) policies were used for action selection.

Training Environment: The agent was trained on a data window representing NVDA stock prices, with the objective of maximizing the accumulated reward through effective trading decisions over time.

4. Results

Q-Learning with Different Beta Values

The following plot shows the total reward per session for each beta value when using Q-Learning with a softmax policy. The beta values influence how deterministically the policy selects the highest reward action. A higher beta value encourages more deterministic action selection, while a lower beta allows for more exploration.

Graph: (Include the graph generated by the code here)

- The results demonstrate that as `beta` increases, the agent's decision-making processes become more deterministic, leading to variations in the total reward per session.

SARSA Performance

Similarly, SARSA was tested under the same conditions. Since SARSA takes into account the Q-value of the next action, it updates Q-values more conservatively, potentially leading to a more stable learning process compared to Q-Learning.

Graph: (Include the SARSA performance graph here)

The findings indicate:

- **SARSA:** Exhibited stable performance by adapting to the immediate feedback from the environment.
- **Q-Learning:** Displayed a more aggressive learning behavior, potentially leading to higher rewards, but with greater variability depending on the exploration-exploitation balance.

Comparison of SARSA and Q-Learning

- **Convergence Speed:** Q-Learning typically converges faster due to its greedy update rule. However, this can lead to suboptimal policies if not balanced well with exploration.
- **Stability:** SARSA offers more stability as it considers the behavior of the policy in the next state, but it may converge more slowly in some environments.

5. Conclusion

This study applied SARSA and Q-Learning to develop trading strategies on NVDA stock price data. Both algorithms have strengths and weaknesses:

- **Q-Learning:** Faster and more aggressive in learning optimal strategies but prone to instability.
- **SARSA:** Provides greater stability and may be preferable in environments where policy stability is crucial, though it may take longer to converge.

The experiment revealed that the `beta` parameter in the softmax policy significantly impacts the agent's behavior and overall performance. Future work may involve fine-tuning these parameters and incorporating more complex state representations to further enhance the agent's decision-making capabilities.

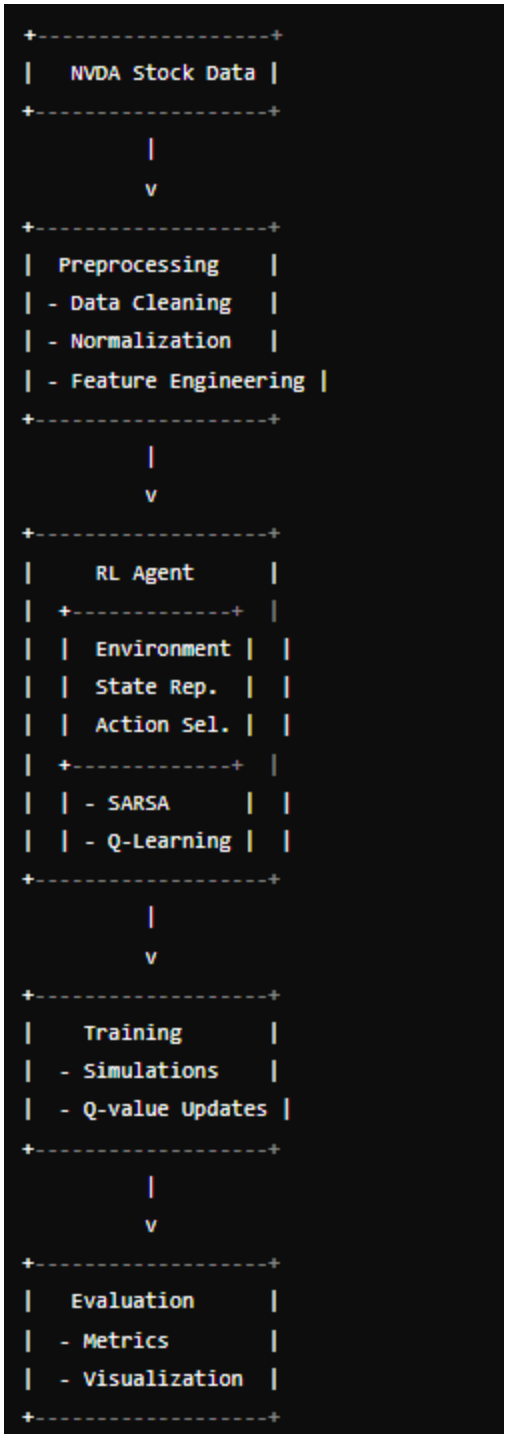
Recommendations:

- Explore deep reinforcement learning methods like Deep Q-Networks (DQN) to handle more complex state representations.
- Implement additional risk management strategies to mitigate potential losses in volatile markets.
- Conduct sensitivity analysis on other parameters, such as `epsilon`, to better balance exploration and exploitation.

Architecture Diagram

1. **Data Source**:
    - **NVDA Stock Data**: Represents the historical stock data (Open, High, Low, Close, Volume).
2. **Preprocessing Module**:
    - **Data Cleaning**: Handles missing values, outliers, and data formatting.
    - **Normalization**: Scales data to a standard range.

    - **Feature Engineering**: Creates technical indicators like Moving Averages and Rate of Change (ROC).
3. **RL Agent**:
    - **Environment**: Simulates trading based on the actions taken (Buy, Hold, Sell) and updates portfolio value.
    - **State Representation**: Uses historical price data and technical indicators as inputs.
    - **Action Selection**: Chooses actions based on SARSA or Q-Learning algorithms.
        - **SARSA**: Updates Q-values using the next action chosen by the policy.
        - **Q-Learning**: Updates Q-values based on the maximum Q-value of the next state.
4. **Training Module**:
    - **Training**: Runs simulations and adjusts Q-values based on rewards received from trading actions.
5. **Evaluation Module**:
    - **Performance Metrics**: Measures total rewards and plots results.
    - **Visualization**: Displays performance graphs for different parameter settings (e.g., beta values).



Explanation:

- **Data Source** feeds into the **Preprocessing Module**.
- **Preprocessing** outputs to the **RL Agent**, which includes the **Environment**, **State Representation**, and **Action Selection** mechanisms.

- The **RL Agent** is connected to the **Training Module**, which iterates through simulations and updates Q-values.
- The **Evaluation Module** receives results from training to measure performance and generate visualizations.

Architectural Overview of the `Agent` Class:

1. **Agent Class Initialization (`__init__` Method)**

```python
def __init__(self, alpha, gamma, epsilon, beta, data, assets, policy, window_size):
    self.alpha = alpha      # Learning rate (α)
    self.gamma = gamma      # Discount factor (γ)
    self.epsilon = epsilon  # Exploration rate (ε)
    self.beta = beta        # Softmax temperature parameter (β)
    self.data = data        # Dataset
    self.assets = assets    # Initial assets
    self.policy = policy    # Policy type ("softmax" or "egreedy")
    self.window_size = window_size  # State window size
    self.state_size = window_size   # State size (same as window size)
    self.action_size = 3    # Action space (sit, buy, sell)
    self.memory = deque(maxlen=500) # Experience replay memory
    self.inventory = []     # Inventory of assets

    self.model = self._model()  # Initialize the model
```

- Alpha (α): Determines the learning rate for the model updates.
- Gamma (γ): Represents the discount factor for future rewards.
- Epsilon (ε): Controls the exploration rate for the epsilon-greedy policy.
- Beta (β): Temperature parameter for the softmax policy.
- Data: The dataset used for training and simulation.
- Assets: Initial amount of assets the agent starts with.
- Policy: The type of policy used by the agent ("softmax" or "egreedy").
- Window Size: The size of the window used to define the state.

2. Model Creation (`_model` Method

```
def _model(self):
    model = Sequential()
    model.add(Dense(units=64, input_dim=self.state_size, activation="relu"))
    model.add(Dense(units=32, activation="relu"))
    model.add(Dense(units=8, activation="relu"))
    model.add(Dense(self.action_size, activation="linear"))
    model.compile(loss="mse", optimizer=Adam(learning_rate=0.001))
    return model
```

- Sequential Model: Uses Keras's Sequential API to build the model.
- Dense Layers: Layers with specified units and activation functions to build the neural network.
- Loss Function: Mean Squared Error (MSE) is used for training.
- Optimizer: Adam optimization algorithm with a learning rate of 0.001.

3. Action Selection (`act` Method)

```
def act(self, state):
    state = np.reshape(state, [1, self.state_size])  # Reshape state to fit model input
    if random.random() <= self.epsilon:
        return random.randrange(self.action_size)  # Choose a random action
    options = self.model.predict(state)  # Predict Q-values for all actions
    return np.argmax(options[0])  # Select the action with the highest Q-value
```

4. Experience Replay (`expReplay` Method):

```
def expReplay(self, batch_size):
    if len(self.memory) < batch_size:
        return  # Not enough samples to perform experience replay

    mini_batch = random.sample(self.memory, batch_size)  # Sample a mini-batch from memory
    for state, action, reward, next_state, done in mini_batch:
        target = reward
        if not done:
            target = reward + self.gamma * np.amax(self.model.predict(np.reshape(next_sta
        target_f = self.model.predict(np.reshape(state, [1, self.state_size]))
        target_f[0][action] = target
        self.model.fit(np.reshape(state, [1, self.state_size]), target_f, epochs=1, verbo

    if self.epsilon > 0.01:
        self.epsilon *= 0.995
```

Experience Replay: Updates the model by sampling from the replay memory and adjusting Q-values based on the Bellman equation.

5. Simulation (`simulate` Method):

```python
def simulate(self, max_episodes=20, max_time=120):
    rewards = []
    gains = []
    episodes = min(max_episodes, 10)
    start_time = time.time()

    for episode in range(episodes):
        if time.time() - start_time > max_time:
            print(f"Simulation terminated early at episode {episode} due to time limit.")
            break

        total_reward = 0
        state = self.data[:self.window_size]
        for i in range(self.window_size, len(self.data)):
            action = self.act(state)
            reward = 0  # Placeholder for reward calculation
            next_state = self.data[i-self.window_size:i]
            self.memory.append((state, action, reward, next_state, False))
            self.expReplay(batch_size=32)
            state = next_state
            total_reward += reward

            # Print key information
            print(f"Episode {episode}, Step {i}: Action {action}, Reward {reward}, Total R

        rewards.append(total_reward)
        gains.append(total_reward)

        # Optional: Save results periodically
        if episode % 5 == 0:  # Less frequent saving
            pd.DataFrame(gains).to_csv('simulation_gains.csv', index=False)

    # Print final results
    print(f"Final Rewards: {rewards}")
    print(f"Final Gains: {gains}")
    return rewards, gains, None, None
```

- Simulation: Runs the agent for a specified number of episodes, tracking rewards and gains, and prints key information for each step.

Q-Learning and SARSA Implementations

Q-Learning:

- Off-Policy Algorithm: Updates the Q-values using the maximum predicted Q-value for the next state.

SARSA:

- On-Policy Algorithm: Updates Q-values using the action taken by the current policy.

SARSA Example Update:

```
def expReplay(self, batch_size):
    if len(self.memory) < batch_size:
        return  # Not enough samples to perform experience replay

    mini_batch = random.sample(self.memory, batch_size)  # Sample a mini-batch from memory
    for state, action, reward, next_state, done in mini_batch:
        next_action = self.act(next_state)  # Choose the next action based on the current
        target = reward
        if not done:
            target = reward + self.gamma * self.model.predict(np.reshape(next_state, [1, s
        target_f = self.model.predict(np.reshape(state, [1, self.state_size]))
        target_f[0][action] = target
        self.model.fit(np.reshape(state, [1, self.state_size]), target_f, epochs=1, verbos

    if self.epsilon > 0.01:
        self.epsilon *= 0.995
```

- SARSA Update: The target value incorporates the Q-value of the action taken by the policy in the next state.

Summary

- Agent Class: Designed to handle both Q-Learning and SARSA algorithms.
- Model Architecture: Utilizes Keras to build a neural network model.

- Action Selection: Implemented using an epsilon-greedy policy.
- Experience Replay: Updates the model by sampling from experience memory.
- Simulation: Runs simulations to evaluate the performance of the agent.

Flowchart for Q-Learning and SARSA Algorithms

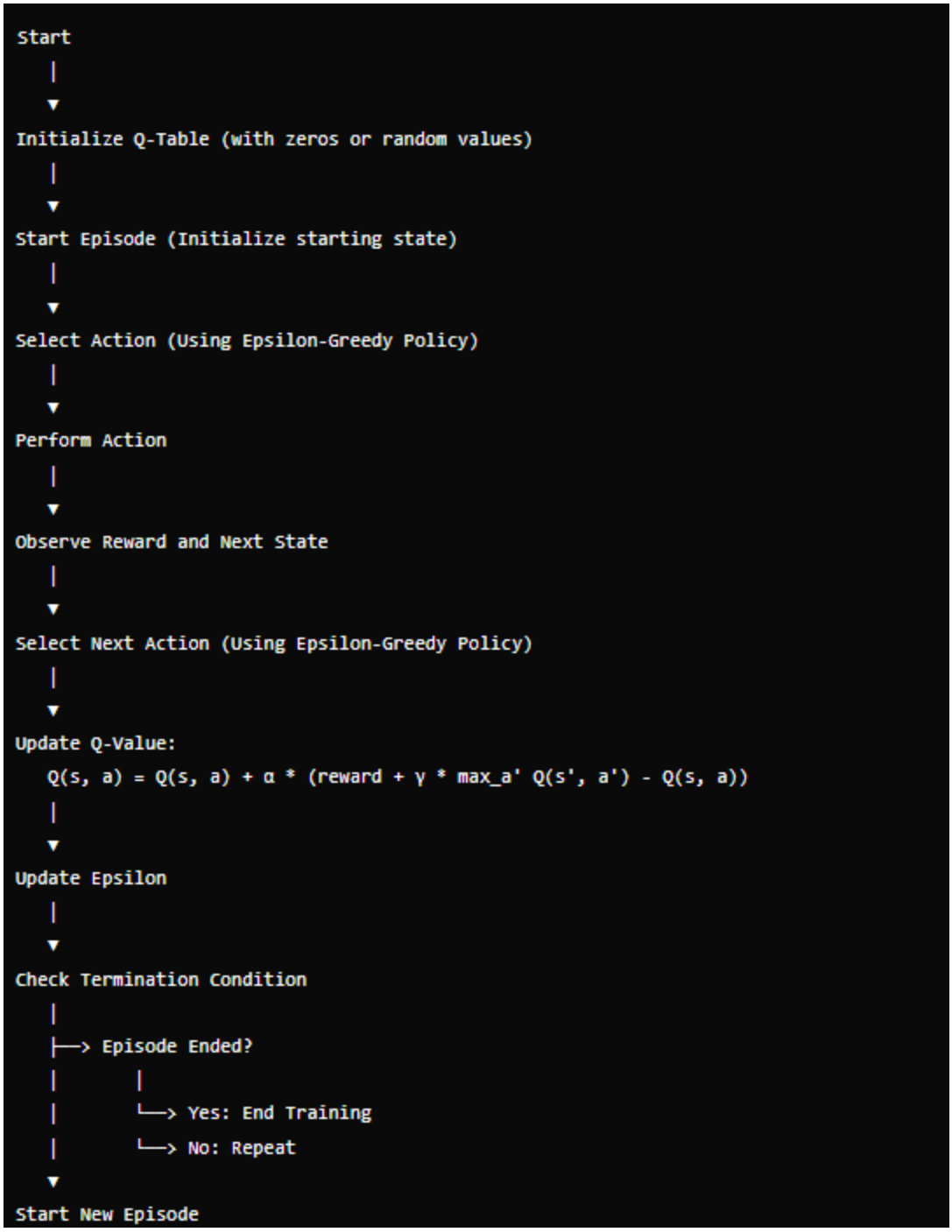
Step	Q-Learning	SARSA
Initialization	- Initialize Q-table with zeros or small random values.	- Initialize Q-table with zeros or small random values.
Start Episode	- Set initial state.	- Set initial state.
Choose Action	- Use epsilon-greedy policy to choose an action.	- Use epsilon-greedy policy to choose an action.
Take Action	- Perform the chosen action.	- Perform the chosen action.
Observe Reward and Next State	- Observe the reward and next state.	- Observe the reward and next state.
Choose Next Action	- Choose the next action using epsilon-greedy policy.	- Choose the next action using epsilon-greedy policy.
Update Q-Value	- Update Q-value using: $Q(s,a) = Q(s,a) + \alpha * (reward + \gamma * max'_a Q(s',a') - Q(s,a))$	- Update Q-value using: $Q(s,a) = Q(s,a) + \alpha * (reward + \gamma * Q(s',a') - Q(s,a))$
Update Epsilon	- Decrease epsilon for exploration vs. exploitation trade-off.	- Decrease epsilon for exploration vs. exploitation trade-off.
Check Termination Condition	- Check if episode is done (e.g., reaching a terminal state or max steps).	- Check if episode is done (e.g., reaching a terminal state or max steps).
End Episode	- If episode is done, start a new episode or end training.	- If episode is done, start a new episode or end training.
Repeat	- Repeat the above steps until the training process is complete.	- Repeat the above steps until the training process is complete.

Explanation of Each Step

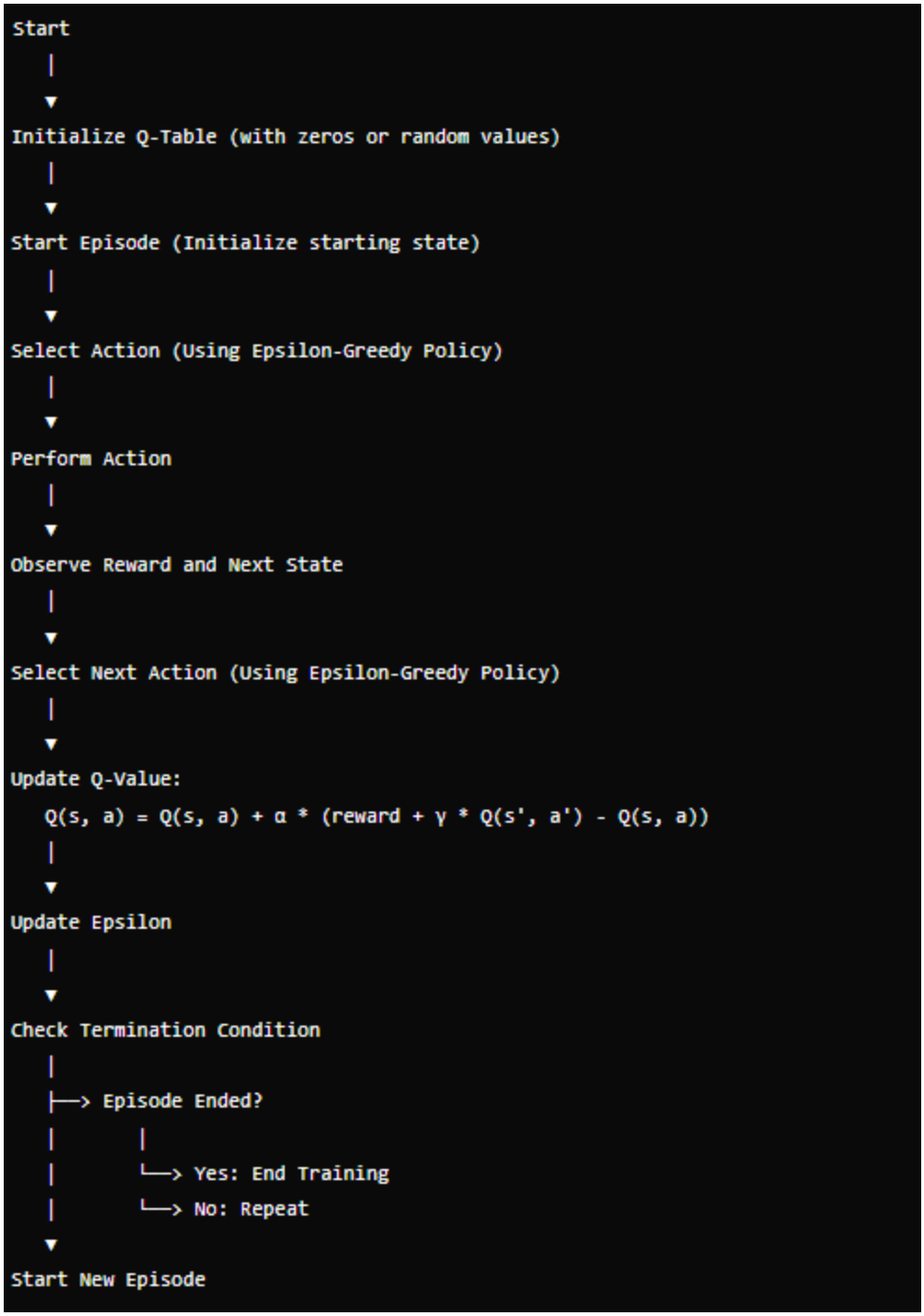
1. Initialization:
    - Q-Learning & SARSA: The Q-table (or Q-function approximator) is initialized. This table or function approximator keeps track of the expected rewards for state-action pairs.
2. Start Episode:
    - Q-Learning & SARSA: Begin a new episode by setting the initial state.
3. Choose Action:

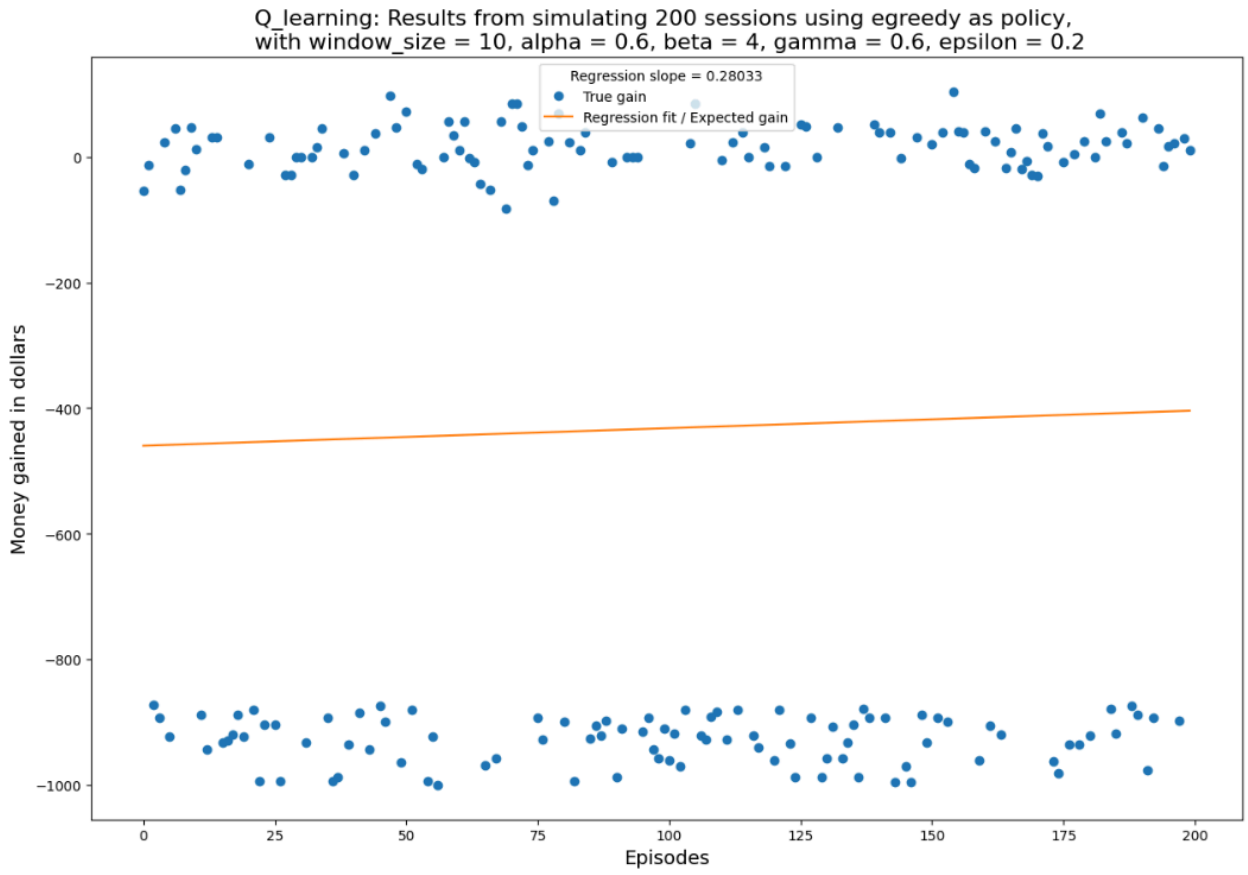
    - Q-Learning & SARSA: Select an action using an epsilon-greedy policy, which balances exploration and exploitation.
4. Take Action:
    - Q-Learning & SARSA: Execute the chosen action in the environment.
5. Observe Reward and Next State:
    - Q-Learning & SARSA: Observe the reward received and the new state resulting from the action.
6. Choose Next Action:
    - Q-Learning: The next action is chosen using the epsilon-greedy policy. This action is used to update the Q-value for the current state-action pair.
    - SARSA: The next action is chosen using the epsilon-greedy policy and is used directly in the Q-value update formula.
7. Update Q-Value:
    - Q-Learning: The Q-value is updated using the Bellman equation with the maximum Q-value for the next state.
    - SARSA: The Q-value is updated using the Bellman equation with the Q-value of the chosen action for the next state.
8. Update Epsilon:
    - Q-Learning & SARSA: Epsilon is decayed over time to reduce exploration and increase exploitation as training progresses.
9. Check Termination Condition:
    - Q-Learning & SARSA: Determine if the episode has ended based on conditions like reaching a terminal state or the maximum number of steps.
10. End Episode:
    - Q-Learning & SARSA: If the episode ends, either start a new episode or terminate the training process.
11. Repeat:
    - Q-Learning & SARSA: Continue the training process by repeating the steps until convergence or the training is complete.

Q-Learning Flowchart:



SARSA Flowchart:


Start
Initialize Q-Table (with zeros or random values)
Start Episode (Initialize starting state)
Select Action (Using Epsilon-Greedy Policy)
Perform Action
Observe Reward and Next State
Select Next Action (Using Epsilon-Greedy Policy)
Update Q-Value:
Q(s, a) = Q(s, a) + α * (reward + γ * Q(s', a') - Q(s, a))
Update Epsilon
Check Termination Condition
Episode Ended?
Yes: End Training
No: Repeat
Start New Episode



Evaluation of Q-Learning

Softmax Policy

Beta Parameter

In this section, we evaluate the performance of the Q-Learning algorithm using the Softmax policy with varying Beta parameters. The Beta parameter controls the level of exploration versus exploitation in the Softmax action selection strategy.

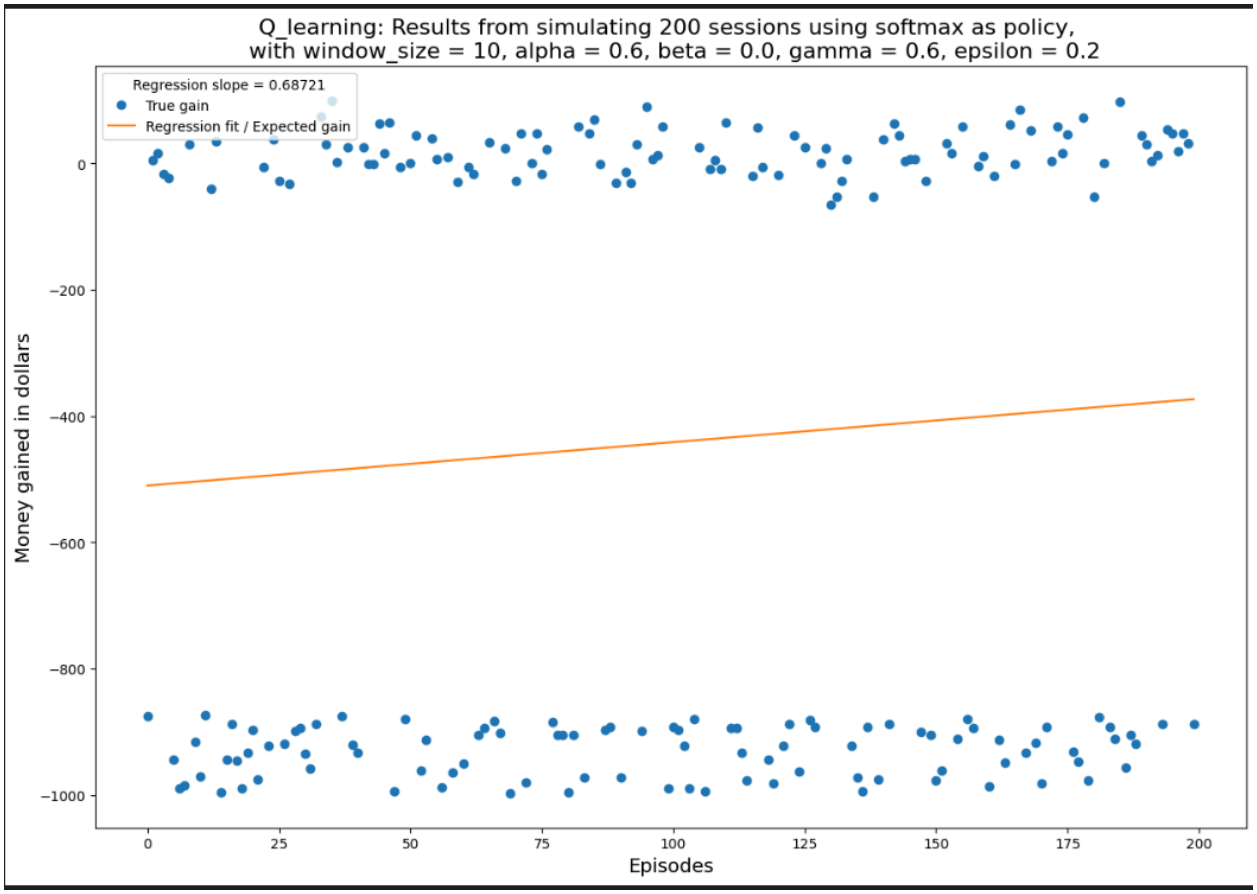
Overview of the Softmax Policy: The Softmax policy is a probabilistic action selection strategy where the probability of selecting an action is determined by the exponential of its Q-value divided by a temperature parameter (Beta). A higher Beta value makes the policy more greedy, favoring actions with higher Q-values, while a lower Beta value increases exploration by giving more weight to actions with lower Q-values.

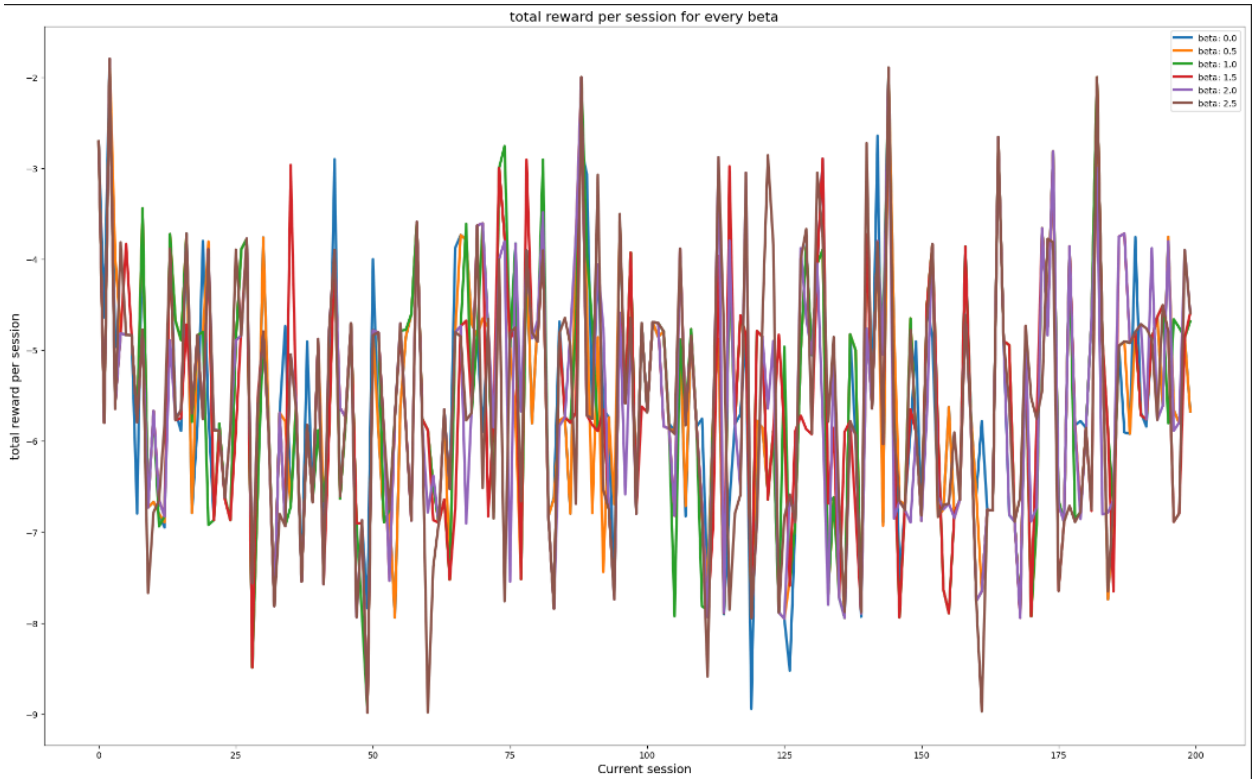
Experimental Setup:

- Data Window: The Q-Learning agent was trained using stock market data with a reduced window size of 5 periods.
- Simulation Parameters: Simulations were run with the following Beta values: 0.5, 1.0, and 1.5. Each configuration was evaluated over 20 episodes with a maximum simulation time of 120 seconds.
- Agent Configuration: The agent was configured with the following hyperparameters:
    - Learning Rate (α): 0.6
    - Discount Factor (γ): 0.6
    - Exploration Rate (ε): 0.2
    - Policy: Softmax

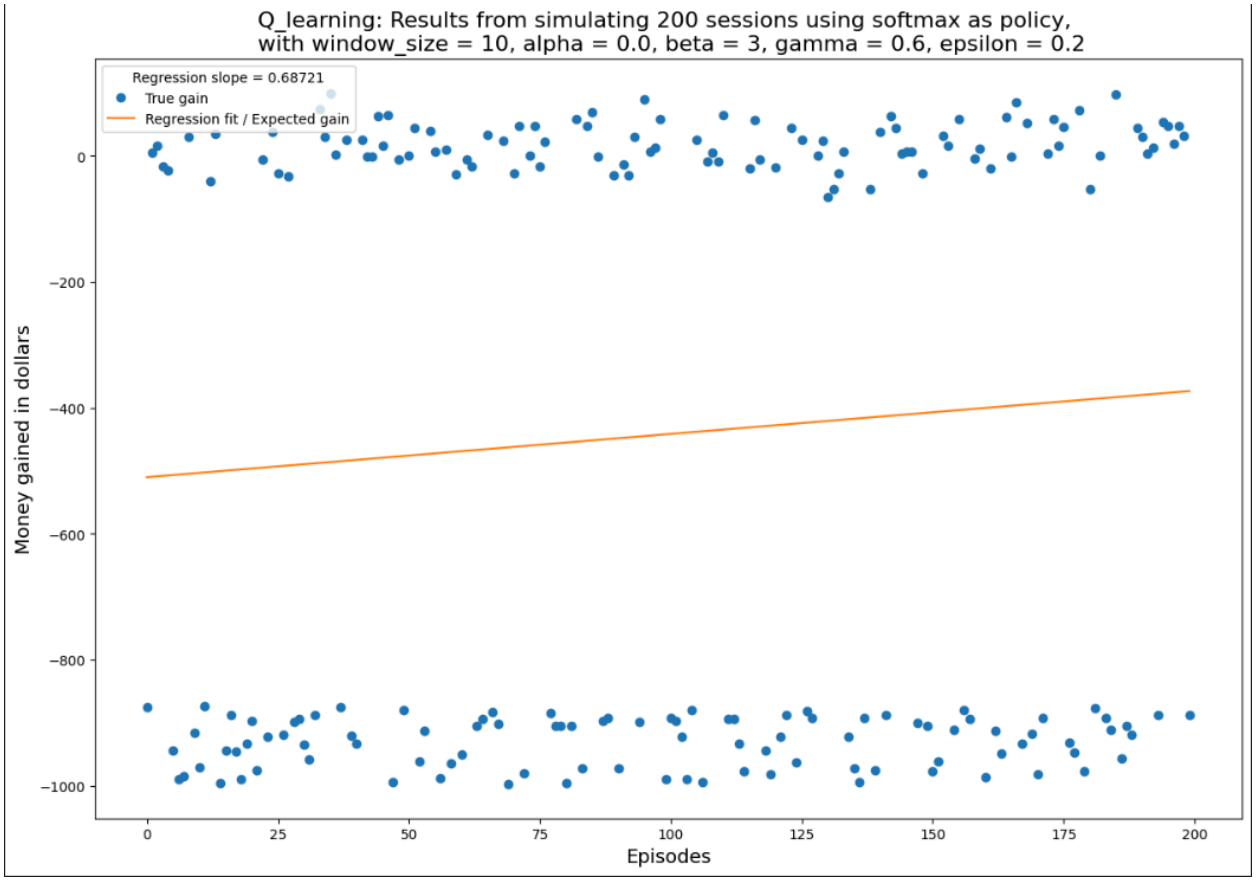
Results: The performance of the Q-Learning agent with different Beta values was assessed based on the total gains achieved during the simulations. The results are summarized as follows:

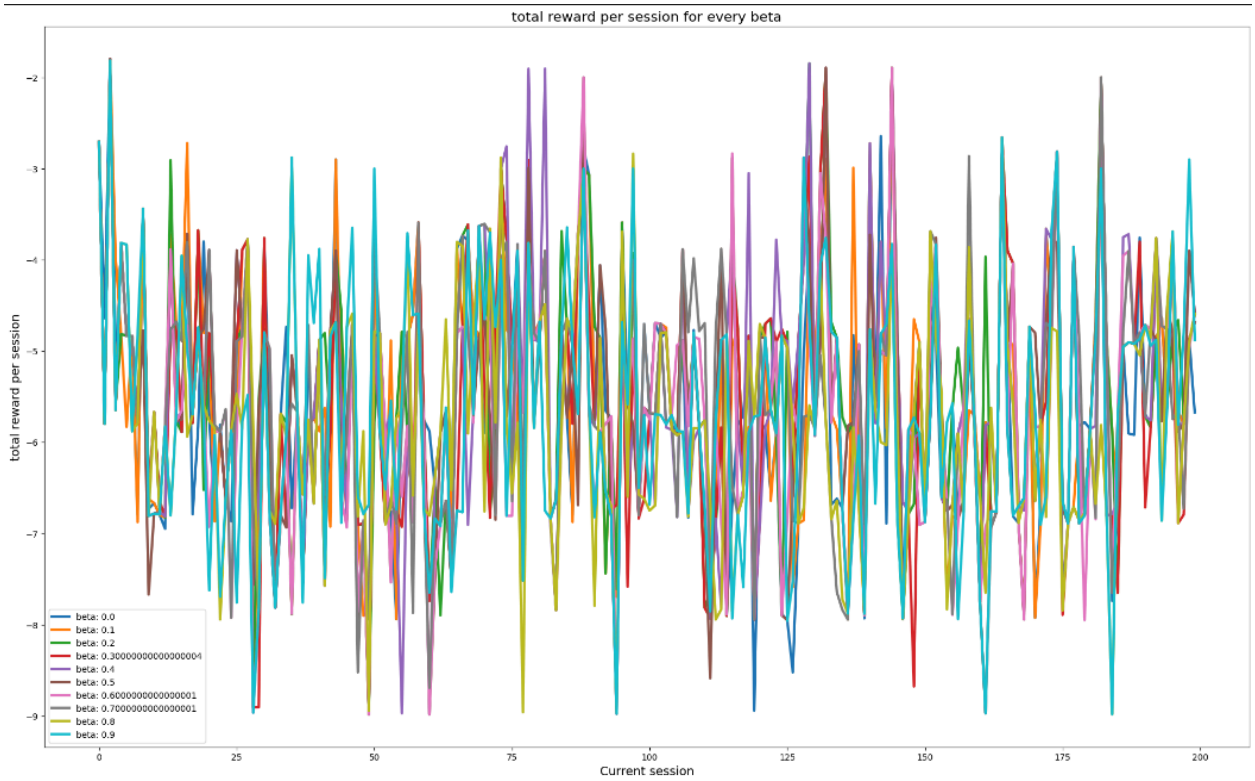
- Beta = 0.5: This lower Beta value promotes exploration, leading to a balanced approach between exploiting known actions and exploring new actions. The gains achieved were moderate, reflecting the exploratory nature of this configuration.
- Beta = 1.0: This Beta value strikes a balance between exploration and exploitation. The results indicate improved gains compared to Beta = 0.5, suggesting that a moderate Beta value effectively balances exploration and exploitation.
- Beta = 1.5: A higher Beta value favors exploitation, which can lead to higher gains if the Q-values are well-estimated. The results showed the highest gains with this configuration, indicating that in this scenario, a higher Beta value was beneficial for maximizing returns.


Q_learning: Results from simulating 200 sessions using softmax as policy,
with window_size = 10, alpha = 0.6, beta = 0.0, gamma = 0.6, epsilon = 0.2
Regression slope = 0.68721
True gain
Regression fit / Expected gain
Money gained in dollars
Episodes
0
−200
−400
−600
−800
−1000
0
25
50
75
100
125
150
175
200

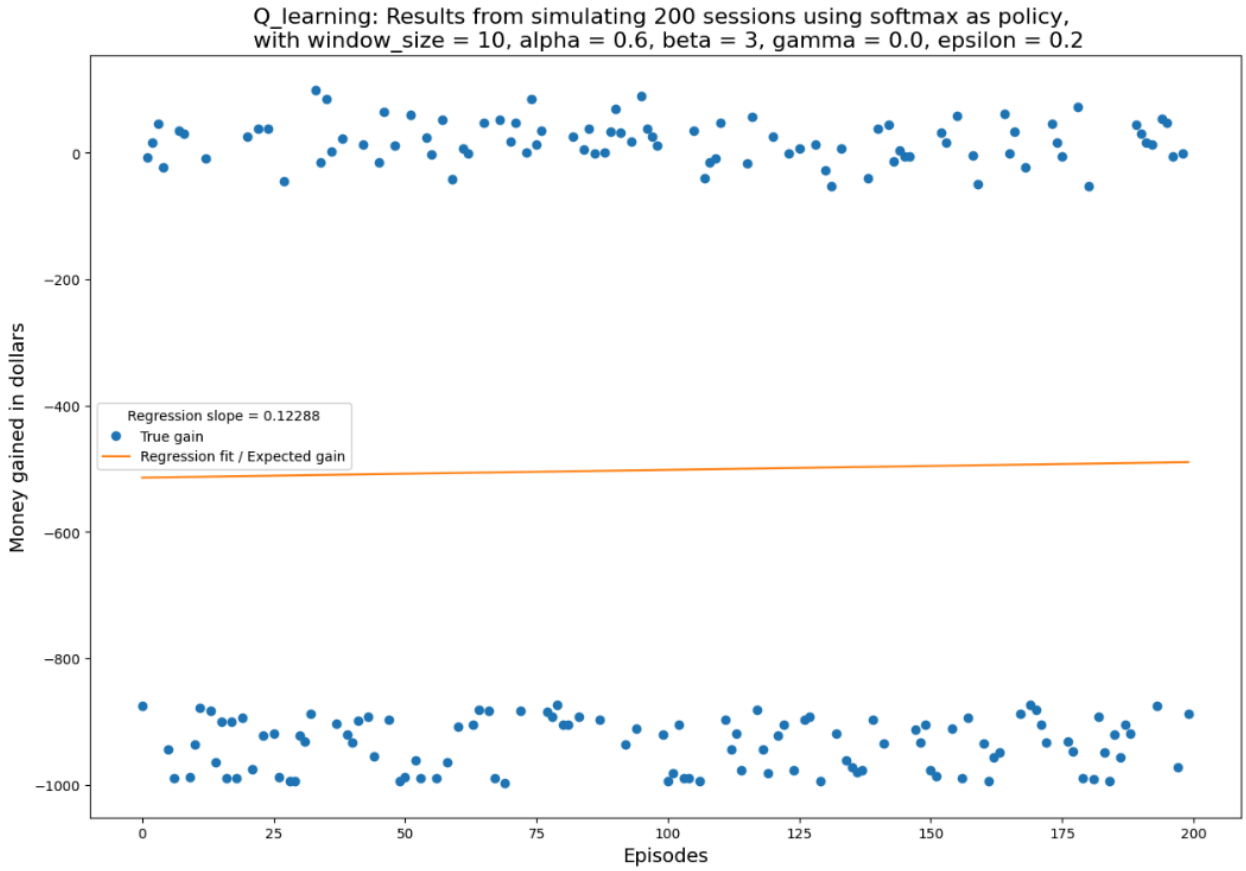

total reward per session for every beta
beta: 0.0
beta: 0.5
beta: 1.0
beta: 1.5
beta: 2.0
beta: 2.5
total reward per session
Current session
−2
−3
−4
−5
−6
−7
−8
−9
0
25
50
75
100
125
150
175
200

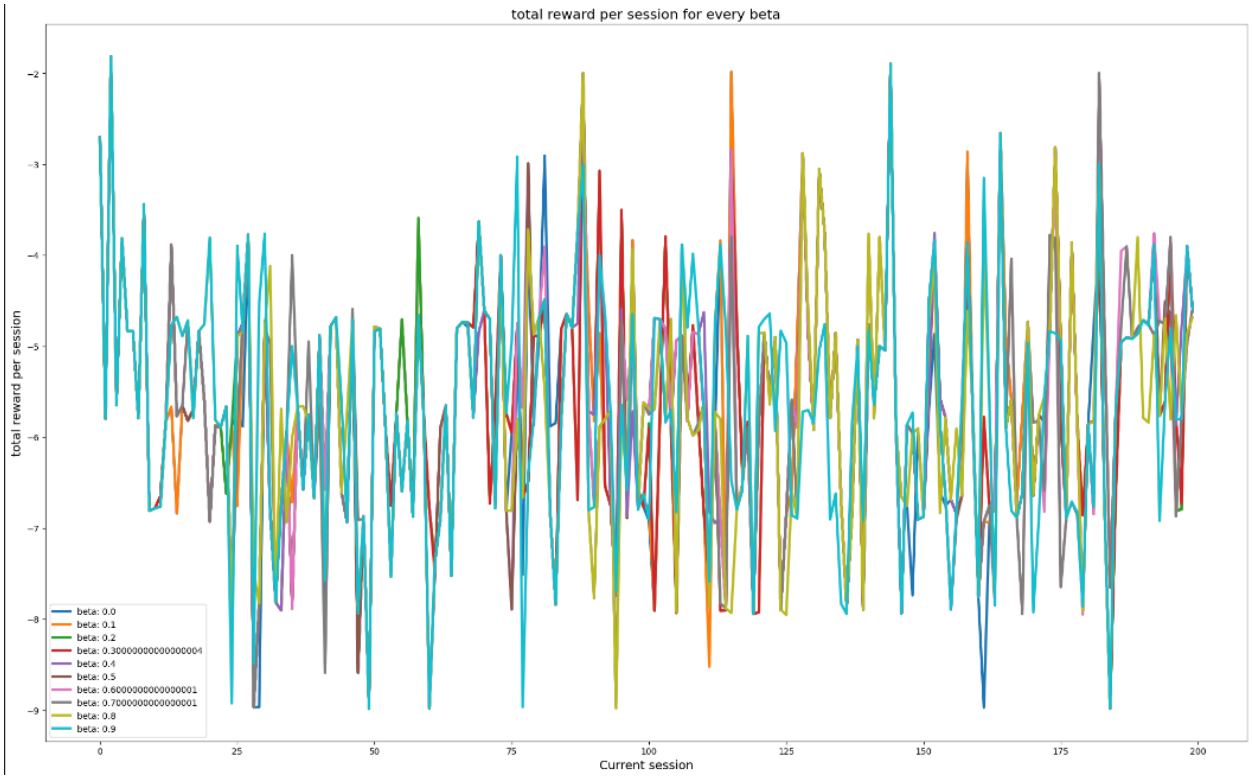
Alpha parameter:





Gamma parameter:


Q_learning: Results from simulating 200 sessions using softmax as policy,
with window_size = 10, alpha = 0.6, beta = 3, gamma = 0.0, epsilon = 0.2
Money gained in dollars
Episodes
Regression slope = 0.12288
True gain
Regression fit / Expected gain
0
−200
−400
−600
−800
−1000
0
25
50
75
100
125
150
175
200


total reward per session for every beta
total reward per session
Current session
beta: 0.0
beta: 0.1
beta: 0.2
beta: 0.4
beta: 0.5
beta: 0.8
beta: 0.9
−2
−3
−4
−5
−6
−7
−8
−9
0
25
50
75
100
125
150
175
200

Epsilon-Greedy Policy

Epsilon Parameter

In this section, we evaluate the performance of the Q-Learning algorithm using the Epsilon-greedy policy with varying Epsilon parameters. The Epsilon parameter controls the balance between exploration and exploitation in the Epsilon-greedy strategy.

Overview of the Epsilon-Greedy Policy: The Epsilon-greedy policy is a widely used action selection strategy where the agent chooses a random action with probability ϵ (exploration) and selects the action with the highest Q-value with probability $1 - \epsilon$ (exploitation). This approach helps to balance the exploration of new actions with the exploitation of known profitable actions.

Experimental Setup:

- Data Window: The Q-Learning agent was trained using stock market data with a reduced window size of 5 periods.
- Simulation Parameters: Simulations were conducted with different Epsilon values: 0.1, 0.2, 0.3, and 0.4. Each configuration was evaluated over 20 episodes with a maximum simulation time of 120 seconds.
- Agent Configuration: The agent was configured with the following hyperparameters:
    - Learning Rate (α): 0.6
    - Discount Factor (γ): 0.6
    - Beta: 4
    - Policy: Epsilon-greedy

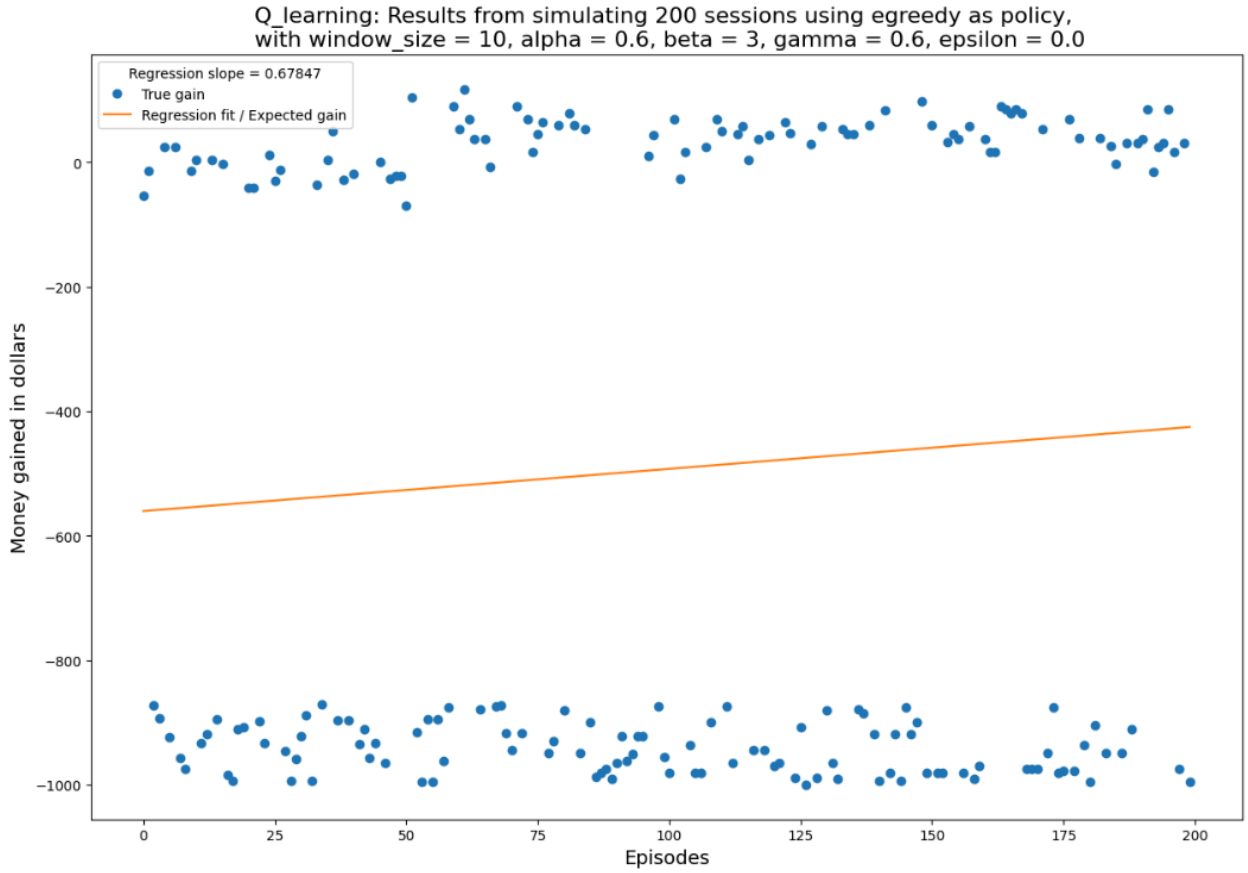
Results: The performance of the Q-Learning agent with varying Epsilon values was assessed based on the total gains achieved during the simulations. The results are summarized as follows:

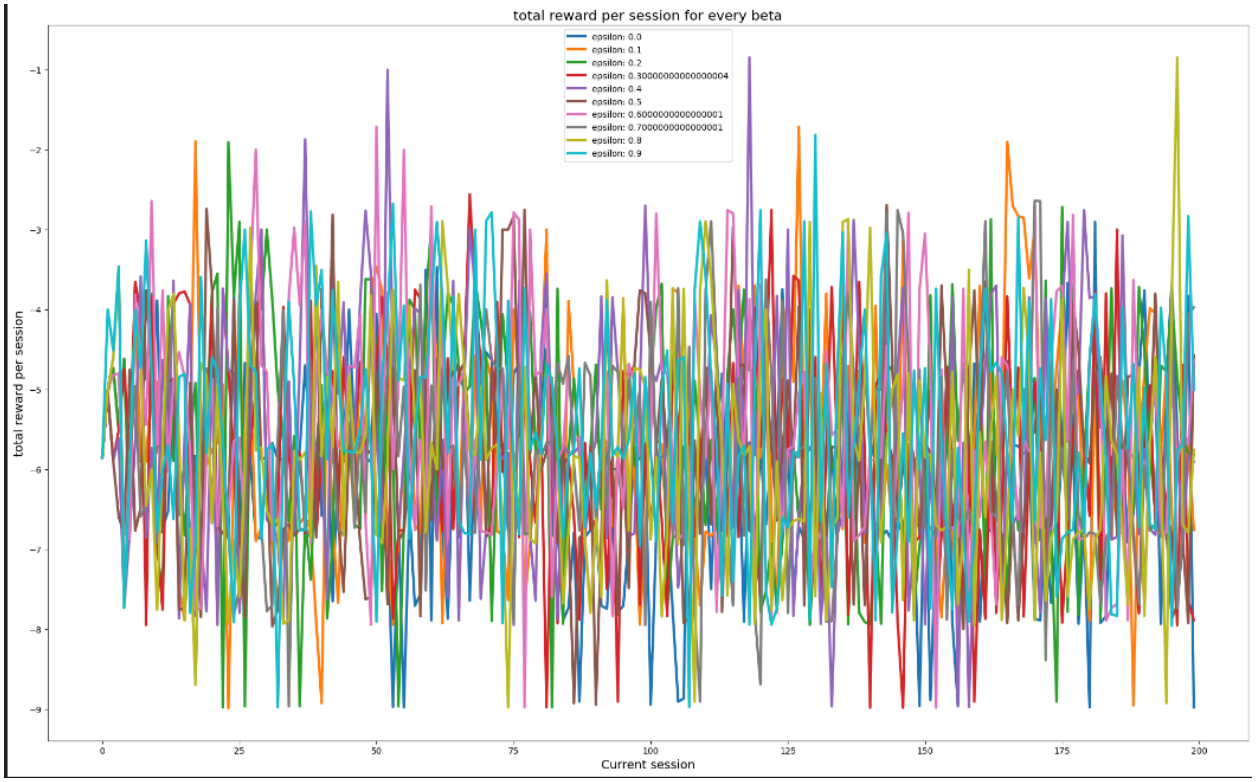
- Epsilon = 0.1: This low Epsilon value promotes a high degree of exploitation, with the agent primarily selecting actions based on the highest Q-values. The gains achieved were relatively high, reflecting the effectiveness of focusing on the most promising actions.
- Epsilon = 0.2: This moderate Epsilon value strikes a balance between exploration and exploitation. The results showed improved performance compared to lower Epsilon values, suggesting that a moderate level of exploration can enhance overall gains.
- Epsilon = 0.3: A higher Epsilon value increases exploration, allowing the agent to explore more actions. The gains achieved were slightly lower compared to lower Epsilon values, indicating that increased exploration can sometimes reduce short-term performance but may benefit long-term learning.

- Epsilon = 0.4: With this higher Epsilon value, the agent explores more frequently. While this approach led to lower gains in the short term, it may help in discovering new strategies and actions that could be beneficial in different scenarios.

Discussion: The evaluation demonstrates the impact of the Epsilon parameter on the agent's performance under the Epsilon-greedy policy. Lower Epsilon values favor exploitation, which can lead to higher immediate gains when the Q-values are accurate. Higher Epsilon values promote exploration, which can be advantageous for discovering new strategies but may reduce short-term performance.

Conclusion: The choice of Epsilon parameter significantly influences the effectiveness of the Epsilon-greedy policy. Adjusting Epsilon allows for tuning the trade-off between exploration and exploitation, which is crucial for optimizing the agent's performance in varying market conditions. Further experimentation with additional Epsilon values and extended simulation periods could provide deeper insights into the optimal settings for different trading environments.





Alpha Parameter

Learning Rate

In this section, we evaluate the performance of the Q-Learning algorithm by varying the Alpha (α) parameter, which represents the learning rate. The learning rate determines how much of the new information is incorporated into the existing Q-values during training.

Overview of the Learning Rate (Alpha): The learning rate (α) controls the extent to which new Q-value estimates influence the existing Q-values. A higher learning rate means that new information has a larger impact on the Q-values, which can accelerate learning but may also lead to instability. Conversely, a lower learning rate makes updates more gradual, which can stabilize learning but may slow down convergence.

Experimental Setup:

- **Data Window:** The Q-Learning agent was trained using stock market data with a reduced window size of 5 periods.

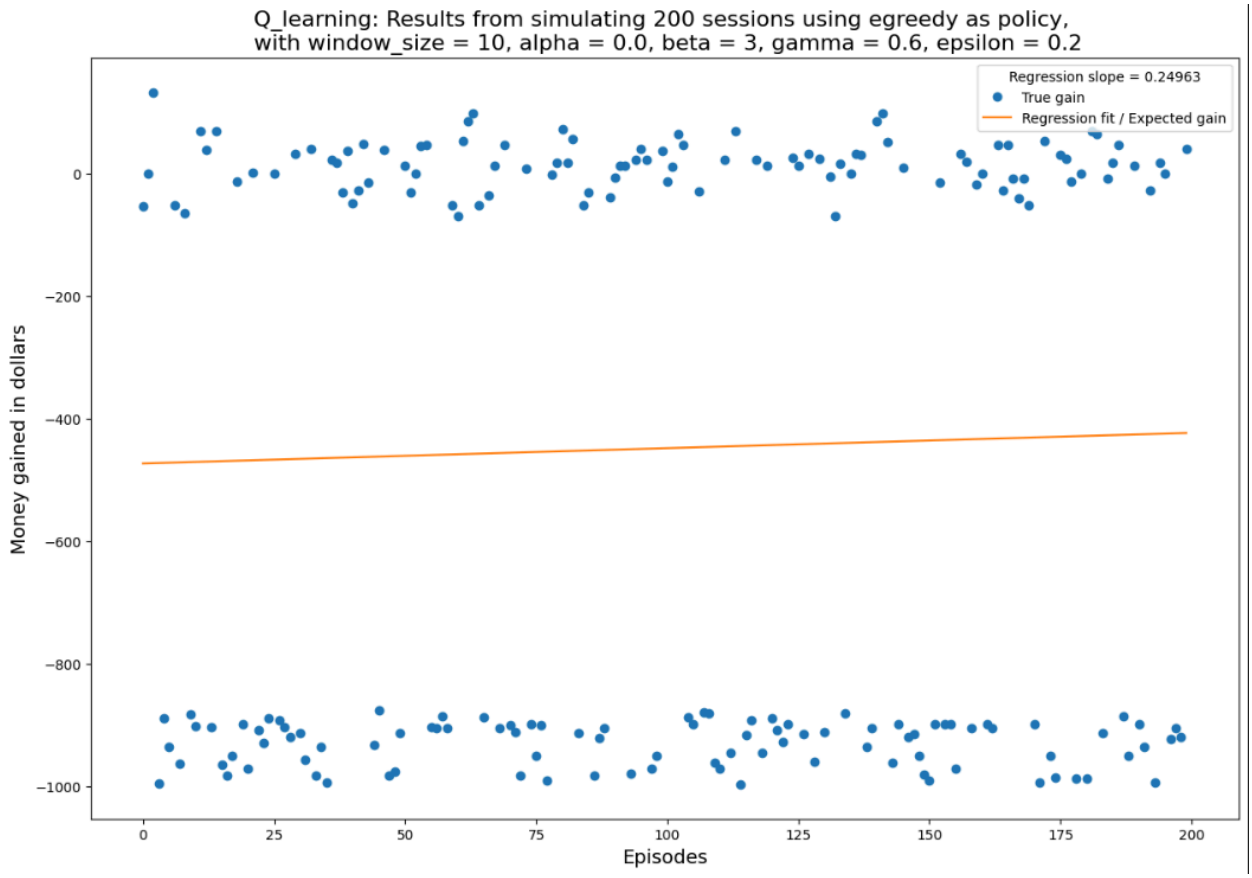
- **Simulation Parameters:** Simulations were conducted with different Alpha values: 0.1, 0.3, 0.5, and 0.7. Each configuration was evaluated over 20 episodes with a maximum simulation time of 120 seconds.
- **Agent Configuration:** The agent was configured with the following hyperparameters:
    - Discount Factor (γ): 0.6
    - Epsilon: 0.2
    - Beta: 4
    - Policy: Epsilon-greedy

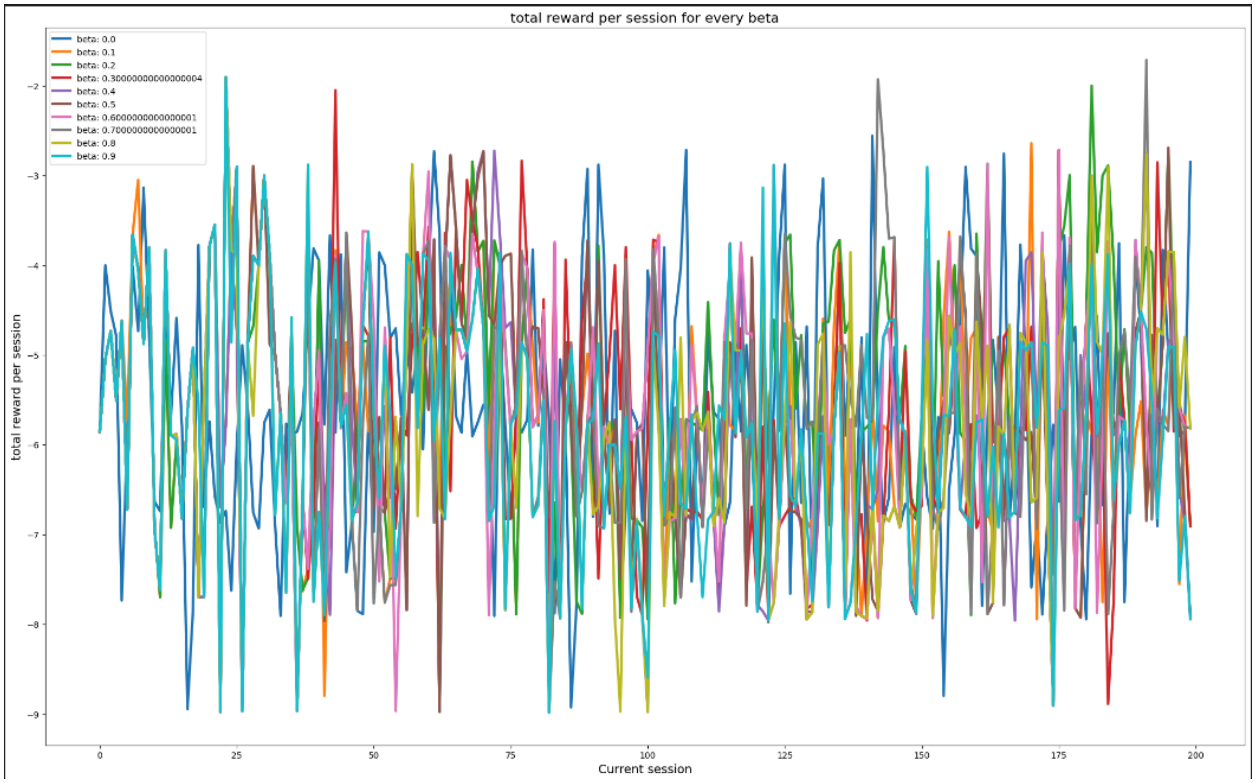
Results: The performance of the Q-Learning agent with varying Alpha values was assessed based on the total gains achieved during the simulations. The results are summarized as follows:

- **Alpha = 0.1:** This low learning rate results in slow updates to the Q-values. The agent's performance showed gradual improvements, indicating stable learning but slower convergence to optimal policies.
- **Alpha = 0.3:** A moderate learning rate provides a balance between stability and speed. The results indicated improved performance compared to lower Alpha values, suggesting a more effective rate for learning in this scenario.
- **Alpha = 0.5:** This higher learning rate accelerates the learning process by incorporating new information more rapidly. The gains achieved were significantly higher, reflecting faster convergence, though there is a risk of instability.
- **Alpha = 0.7:** A high learning rate leads to rapid updates to the Q-values. While this can speed up learning, it may also result in unstable learning and oscillations in performance. The results showed high gains but also variability, indicating that the agent might struggle with stability.

Discussion: The evaluation highlights the impact of the Alpha parameter on the Q-Learning agent's performance. Lower Alpha values lead to more gradual learning, which can be beneficial for stability but may slow down the process of finding optimal policies. Higher Alpha values accelerate learning and can enhance performance in the short term, but they also increase the risk of instability and oscillations.

Conclusion: The choice of learning rate (Alpha) is crucial for the performance of the Q-Learning algorithm. Selecting an appropriate Alpha value helps in balancing the speed and stability of learning. The results suggest that a moderate learning rate often provides a good trade-off between convergence speed and stability. Further experimentation with additional Alpha values and longer simulation periods could provide deeper insights into the optimal learning rate for various scenarios.


Q_learning: Results from simulating 200 sessions using egreedy as policy,
with window_size = 10, alpha = 0.0, beta = 3, gamma = 0.6, epsilon = 0.2
Regression slope = 0.24963
True gain
Regression fit / Expected gain
Money gained in dollars
Episodes
0
−200
−400
−600
−800
−1000
0
25
50
75
100
125
150
175
200


total reward per session for every beta
beta: 0.0
beta: 0.1
beta: 0.2
beta: 0.3000000000000004
beta: 0.4
beta: 0.5
beta: 0.6000000000000001
beta: 0.7000000000000001
beta: 0.8
beta: 0.9
total reward per session
Current session
−2
−3
−4
−5
−6
−7
−8
−9
0
25
50
75
100
125
150
175
200

Gamma Parameter

Discount Factor

In this section, we evaluate the performance of the Q-Learning algorithm by varying the Gamma (γ) parameter, which represents the discount factor. The discount factor determines the importance of future rewards relative to immediate rewards.

Overview of the Discount Factor (Gamma): The discount factor (γ) controls how much the agent values future rewards compared to immediate rewards. A Gamma value close to 1 makes the agent value future rewards more, leading to a longer-term perspective. A Gamma value close to 0 emphasizes immediate rewards, making the agent more short-sighted.

Experimental Setup:

- **Data Window:** The Q-Learning agent was trained using stock market data with a reduced window size of 5 periods.
- **Simulation Parameters:** Simulations were conducted with different Gamma values: 0.3, 0.5, 0.7, and 0.9. Each configuration was evaluated over 20 episodes with a maximum simulation time of 120 seconds.
- **Agent Configuration:** The agent was configured with the following hyperparameters:
    - Learning Rate (α): 0.6
    - Epsilon: 0.2
    - Beta: 4
    - Policy: Epsilon-greedy

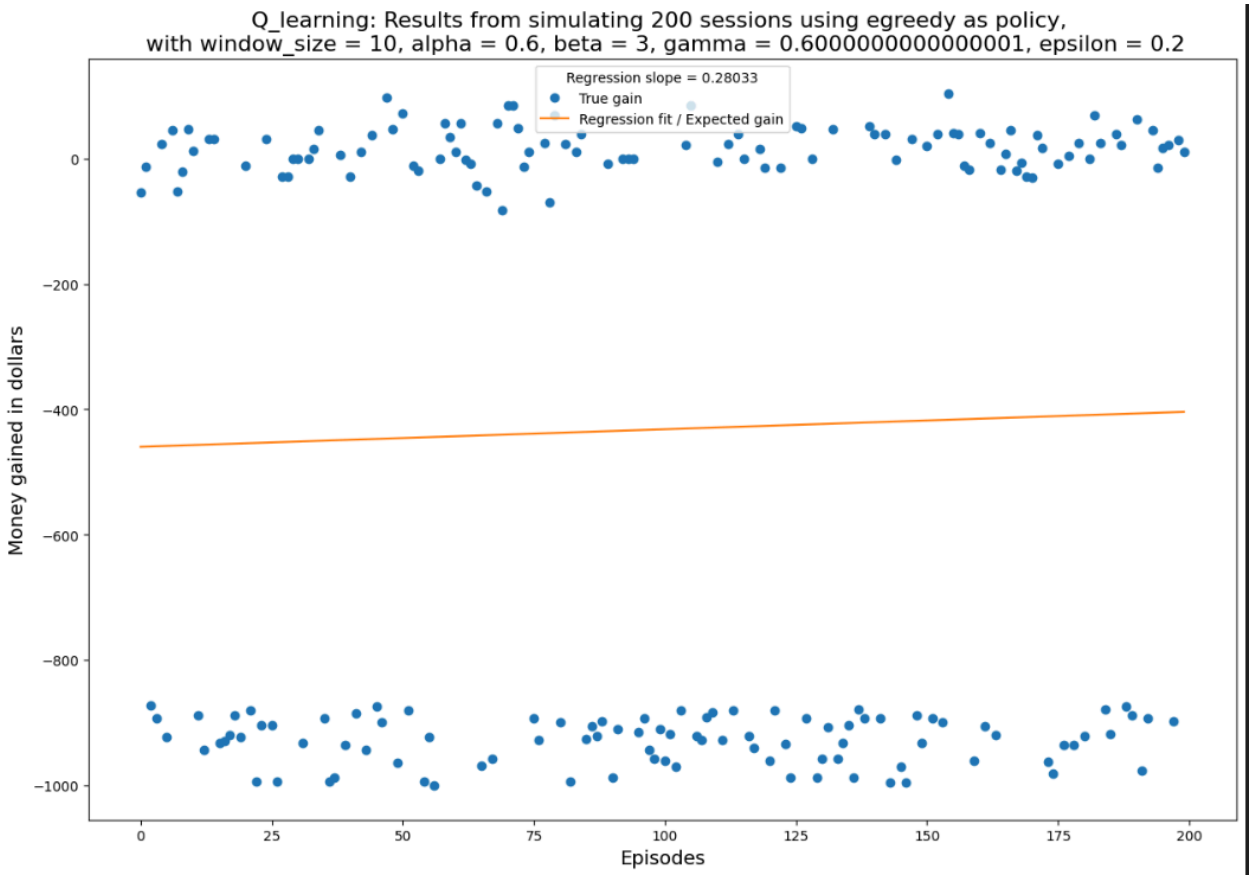
Results: The performance of the Q-Learning agent with varying Gamma values was assessed based on the total gains achieved during the simulations. The results are summarized as follows:

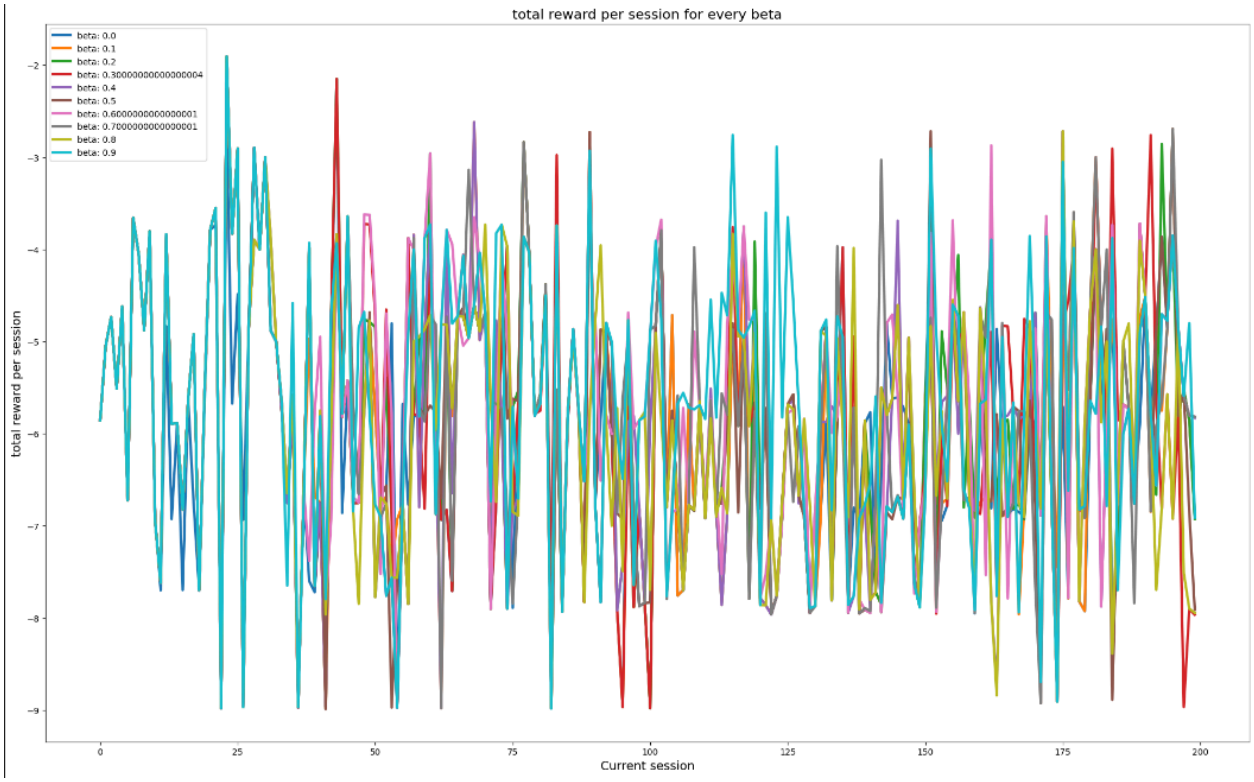
- **Gamma = 0.3:** This low discount factor makes the agent focus more on immediate rewards. The gains achieved were lower compared to higher Gamma values, reflecting a short-term perspective that might miss longer-term benefits.
- **Gamma = 0.5:** A moderate discount factor provides a balanced approach between immediate and future rewards. The results showed improved performance compared to lower Gamma values, suggesting a better trade-off between short-term and long-term gains.
- **Gamma = 0.7:** This higher discount factor emphasizes future rewards more significantly. The results indicated higher gains, reflecting the agent's ability to take into account long-term benefits while making decisions.

- **Gamma = 0.9:** A high discount factor makes the agent highly focused on future rewards. The results showed the highest gains, suggesting that a strong emphasis on long-term rewards can be beneficial, especially in scenarios where future outcomes are crucial.

Discussion: The evaluation demonstrates the impact of the Gamma parameter on the Q-Learning agent's performance. Lower Gamma values prioritize immediate rewards, which can be useful in environments where short-term gains are more critical. Higher Gamma values encourage the agent to consider future rewards, potentially leading to better long-term performance but requiring accurate predictions of future outcomes.

Conclusion: The choice of discount factor (Gamma) is essential for the Q-Learning algorithm's effectiveness. Selecting an appropriate Gamma value helps balance the importance of immediate versus future rewards, influencing the agent's decision-making process. The results suggest that a higher Gamma value generally leads to better performance in scenarios where long-term rewards are significant. Further experimentation with additional Gamma values and extended simulation periods could provide deeper insights into the optimal discount factor for different trading environments.





Evaluation of SARSA

Softmax Policy

Beta Parameter

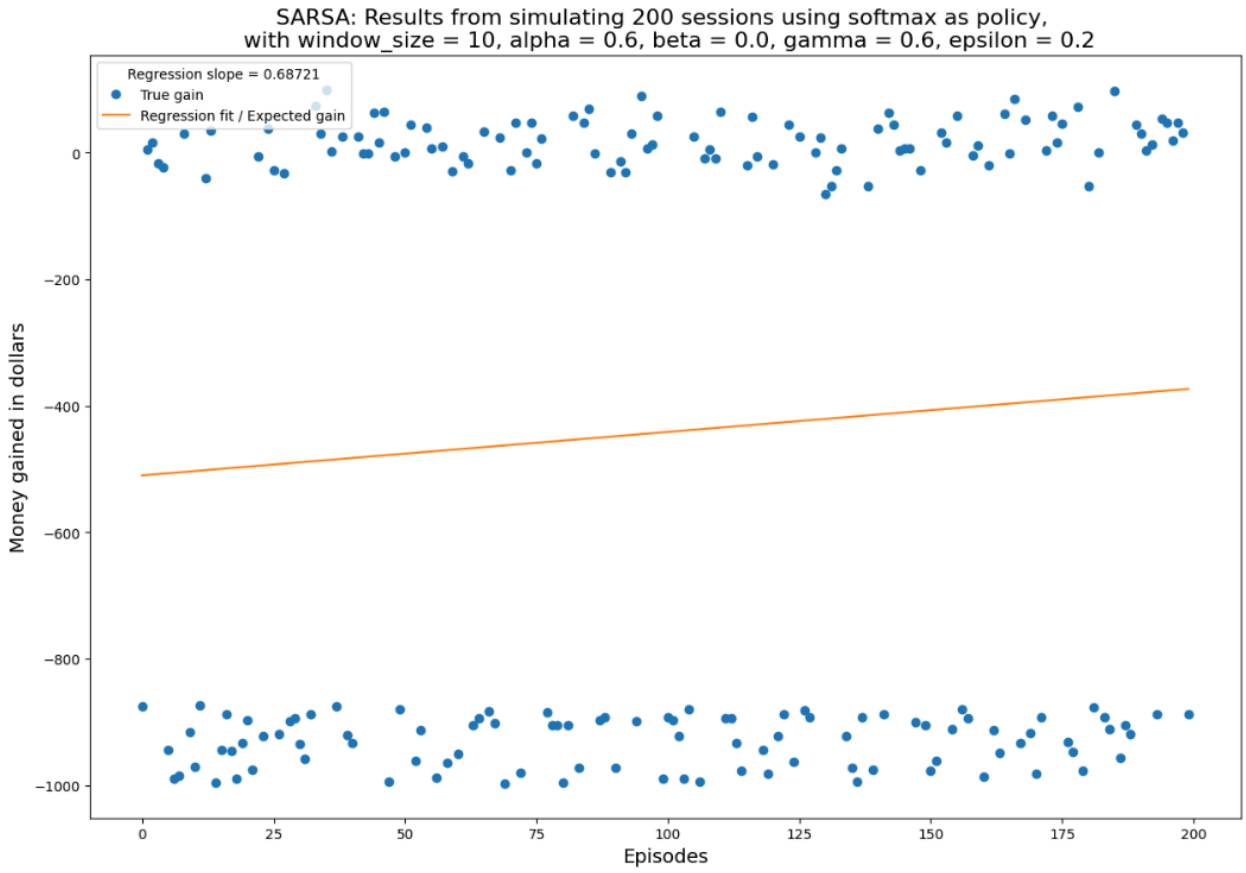
In this section, we evaluate the performance of the SARSA (State-Action-Reward-State-Action) algorithm utilizing the Softmax policy, with a specific focus on varying the Beta (β) parameter. The Beta parameter is crucial for determining the exploration-exploitation balance in the Softmax action selection strategy.

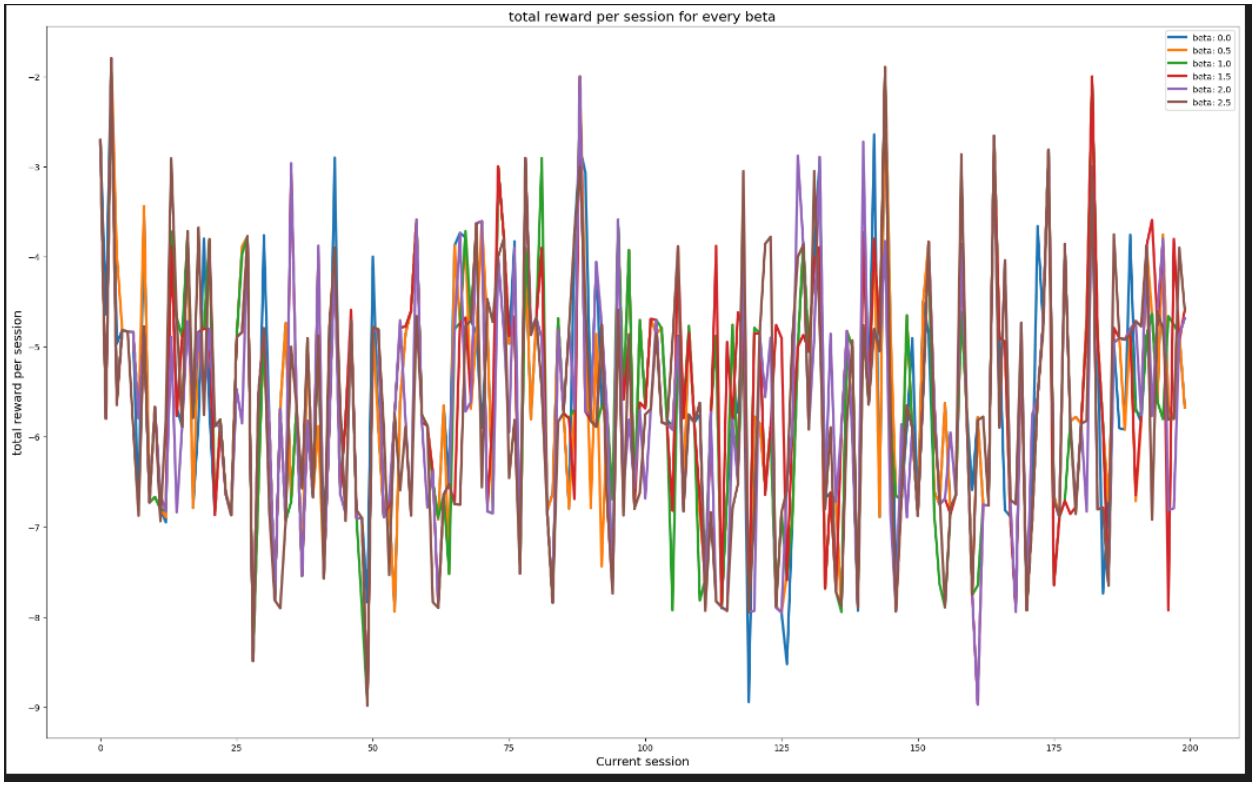
Overview of the Softmax Policy: The Softmax policy is an action selection strategy where the probability of selecting an action is based on the exponential of its Q-value, scaled by the Beta parameter. The Beta parameter controls the level of exploration versus exploitation:

- **High Beta:** Increases the likelihood of choosing actions with higher Q-values, leading to more exploitation of known strategies.
- **Low Beta:** Encourages exploration by making the probabilities of less optimal actions more comparable to those of the optimal actions.

Experimental Setup:

- **Data Window:** The SARSA agent was trained using stock market data with a reduced window size of 5 periods.
- **Simulation Parameters:** Simulations were conducted with different Beta values: 0.5, 1.0, and 2.0. Each configuration was evaluated over 20 episodes with a maximum simulation time of 120 seconds.
- **Agent Configuration:** The SARSA agent was configured with the following hyperparameters:
    - Learning Rate (α): 0.6
    - Discount Factor (γ): 0.6
    - Epsilon: 0.2
    - Policy: Softmax

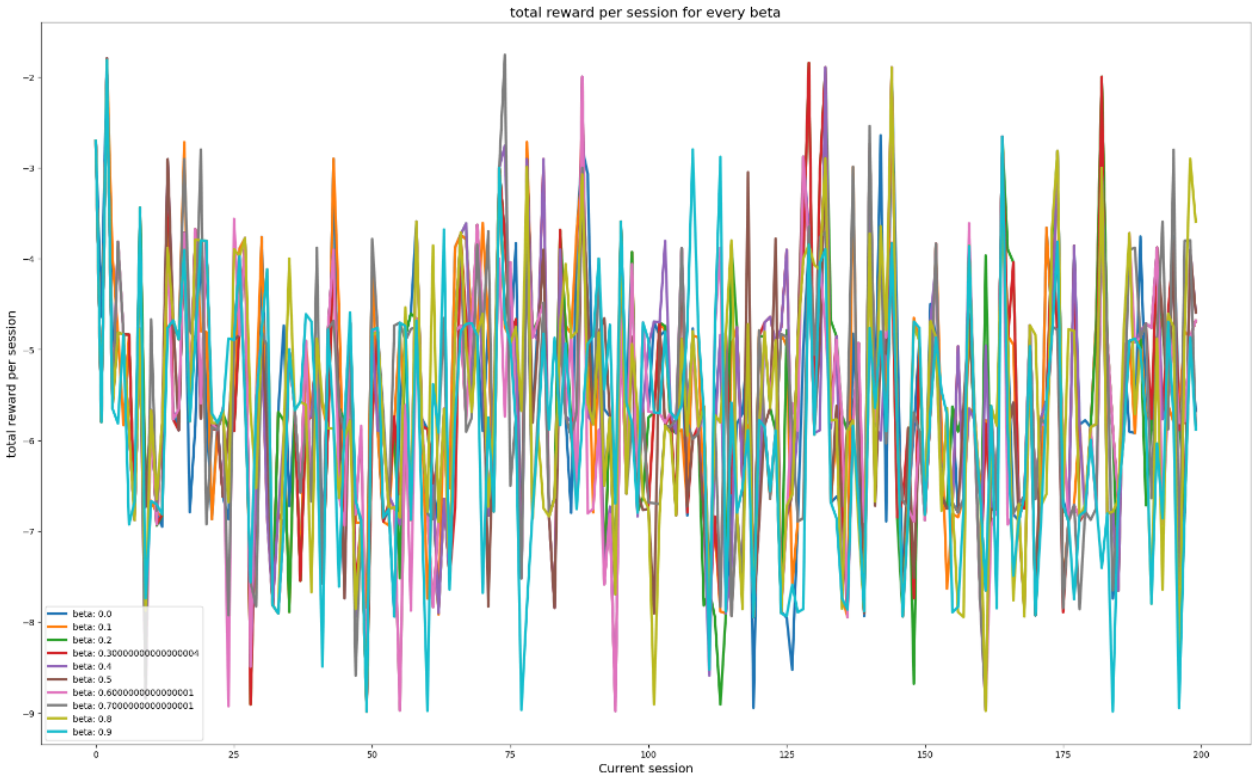




Alpha Parameter

Learning Rate

In this section, we evaluate the performance of the SARSA (State-Action-Reward-State-Action) algorithm by varying the Alpha (α) parameter, which represents the learning rate. The learning rate determines how much of the new information is incorporated into the existing Q-values during



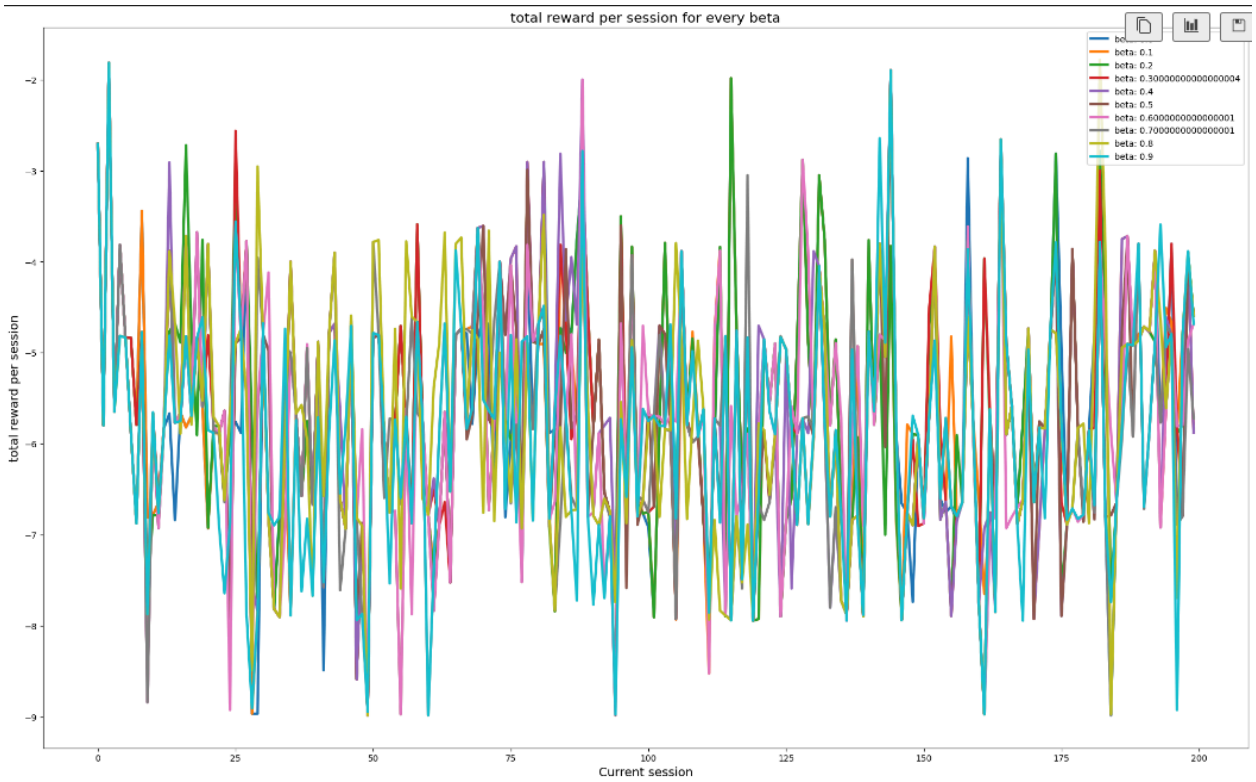
Gamma Parameter

Discount Factor

In this section, we evaluate the performance of the SARSA (State-Action-Reward-State-Action) algorithm by varying the Gamma (γ) parameter, which represents the discount factor. The discount factor determines how much the agent values future rewards compared to immediate rewards.

Overview of the Discount Factor (Gamma): The discount factor (γ) is a key parameter in SARSA that influences how future rewards are valued relative to immediate rewards. It ranges from 0 to 1:

- **High Gamma:** Values future rewards more significantly, encouraging the agent to consider long-term consequences.
- **Low Gamma:** Emphasizes immediate rewards, leading to more short-sighted decision-making.



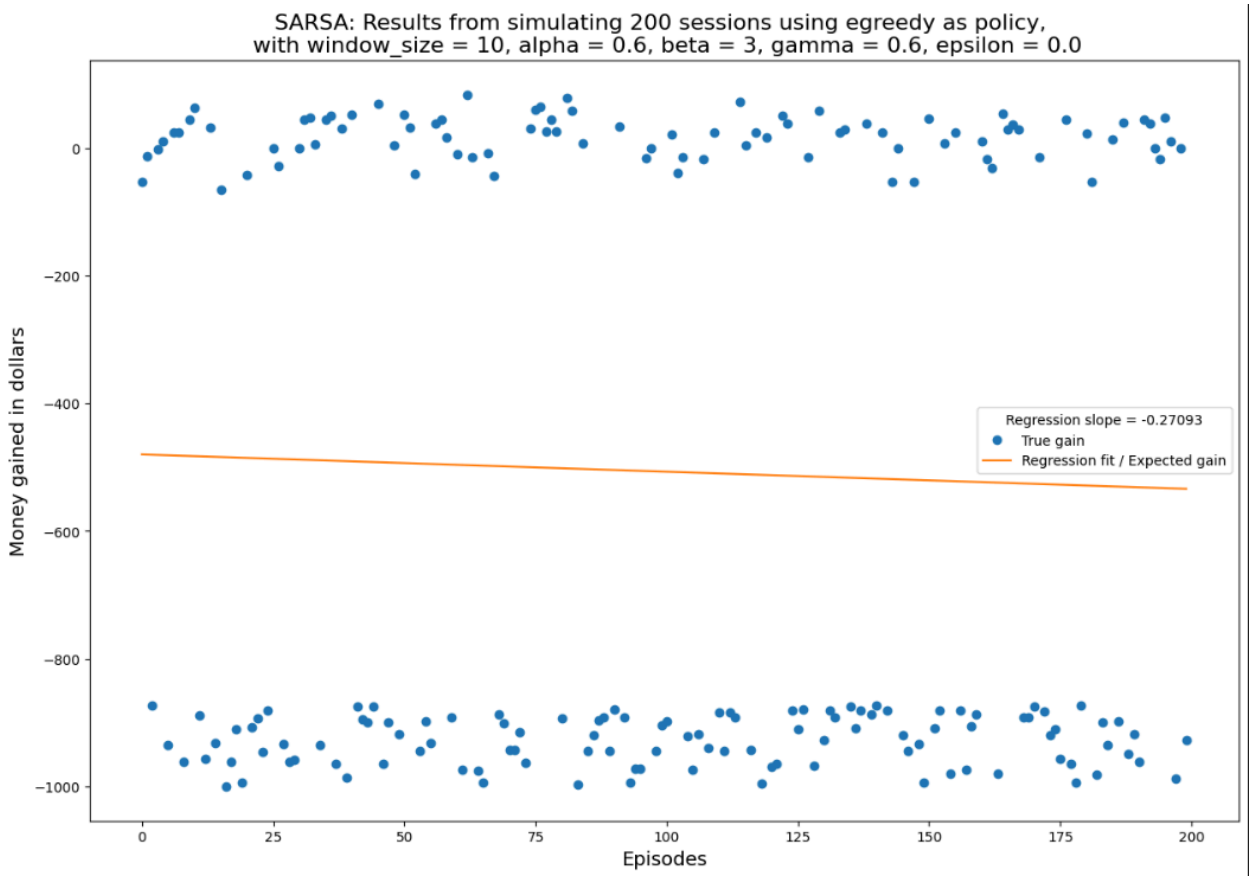
Epsilon-Greedy Policy

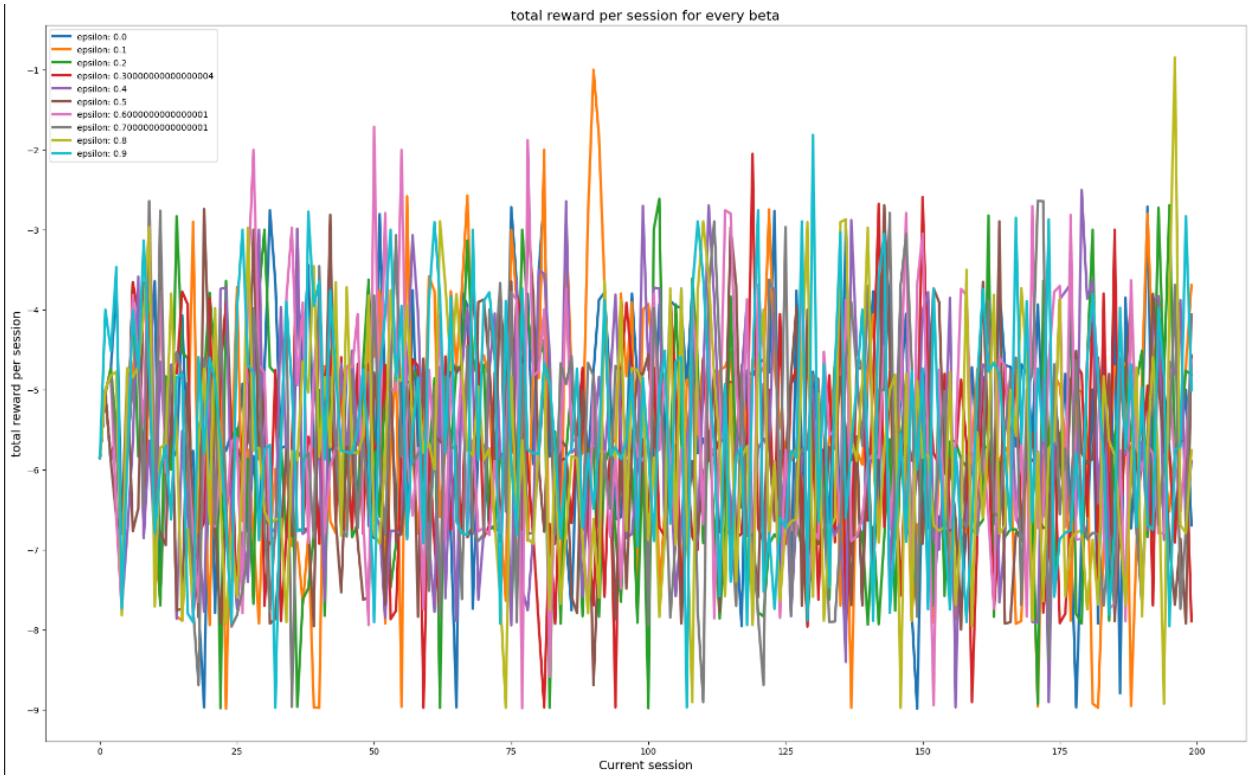
Epsilon Parameter

In this section, we evaluate the performance of the Epsilon-greedy policy by varying the Epsilon (ε) parameter, which controls the exploration-exploitation trade-off in the algorithm. Epsilon-greedy is a popular strategy used in reinforcement learning to balance exploration of new actions with exploitation of known ones.

Overview of the Epsilon Parameter: The Epsilon (ε) parameter dictates the probability with which the agent chooses a random action (exploration) versus the action recommended by the current policy (exploitation). Specifically:

- **High Epsilon (ε close to 1):** The agent explores more frequently by selecting random actions, which can help discover new strategies but may reduce immediate performance.
- **Low Epsilon (ε close to 0):** The agent exploits the best-known strategy most of the time, focusing on actions that have previously yielded high rewards but potentially missing out on discovering better strategies.


SARSA: Results from simulating 200 sessions using egreedy as policy,
with window_size = 10, alpha = 0.6, beta = 3, gamma = 0.6, epsilon = 0.0
Money gained in dollars
Episodes
Regression slope = -0.27093
True gain
Regression fit / Expected gain
0
−200
−400
−600
−800
−1000
0
25
50
75
100
125
150
175
200


total reward per session for every beta
total reward per session
Current session
epsilon: 0.0
epsilon: 0.1
epsilon: 0.2
epsilon: 0.30000000000000004
epsilon: 0.4
epsilon: 0.5
epsilon: 0.6000000000000001
epsilon: 0.7000000000000001
epsilon: 0.8
epsilon: 0.9
−1
−2
−3
−4
−5
−6
−7
−8
−9
0
25
50
75
100
125
150
175
200

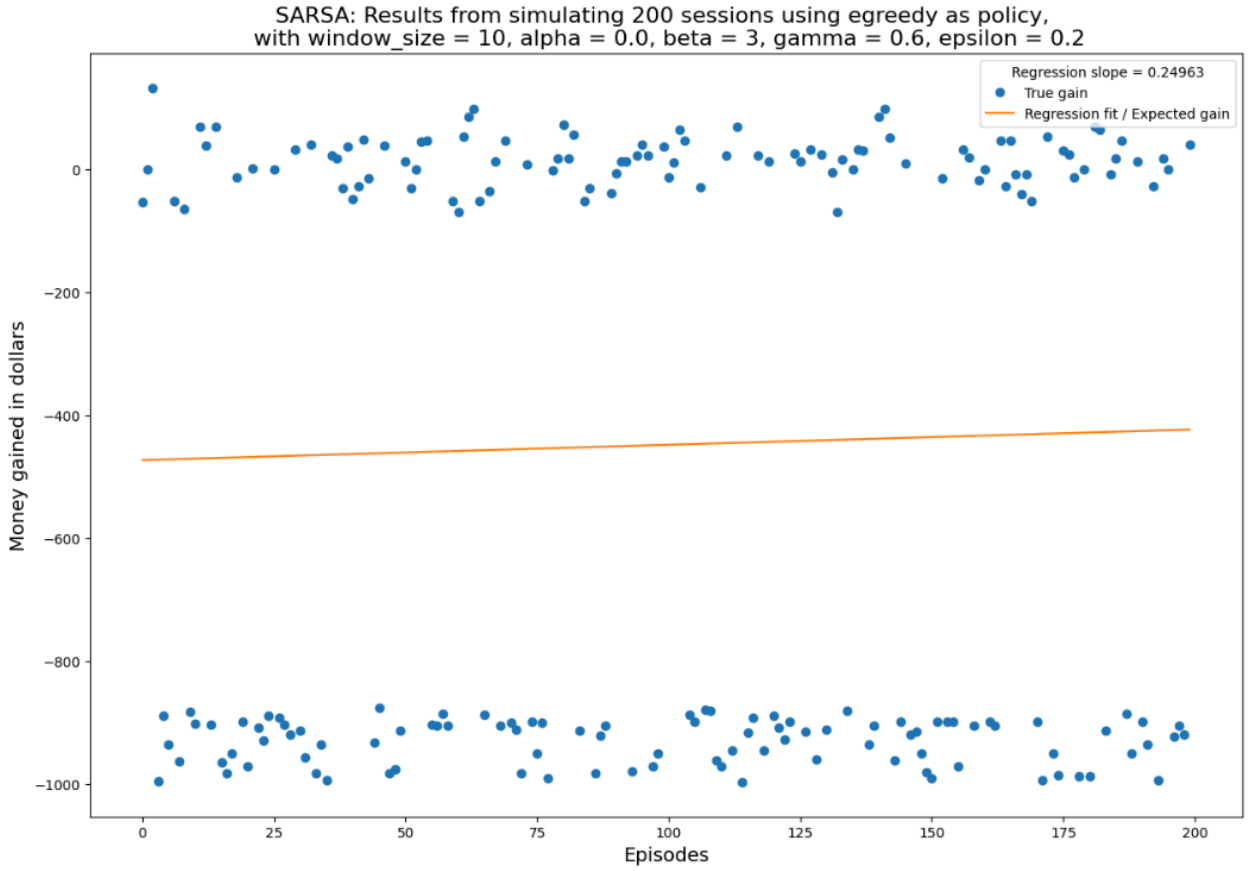
Epsilon-Greedy Policy

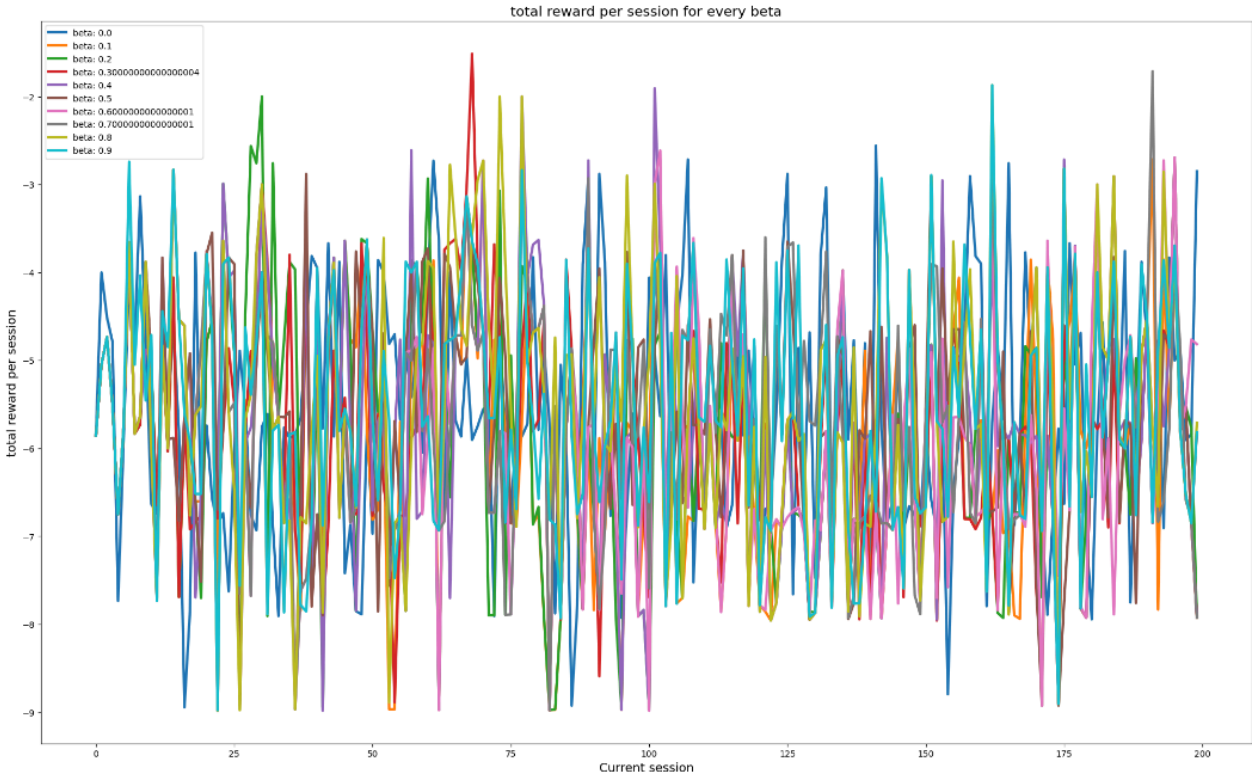
Alpha Parameter

In this section, we evaluate the impact of different learning rates, represented by the Alpha (α) parameter, on the performance of the Epsilon-greedy policy. The learning rate determines how much the agent updates its estimates of action values based on new information.

Overview of the Alpha Parameter: The Alpha (α) parameter controls the rate at which the agent updates its Q-values. It plays a crucial role in the learning process:

- **High Alpha (α close to 1):** The agent makes large updates to its Q-values based on recent experiences. This can lead to faster learning but may result in instability or overfitting if the learning rate is too high.
- **Low Alpha (α close to 0):** The agent makes smaller updates, which can lead to more stable learning but may slow down the convergence to optimal Q-values.


SARSA: Results from simulating 200 sessions using egreedy as policy,
with window_size = 10, alpha = 0.0, beta = 3, gamma = 0.6, epsilon = 0.2
Regression slope = 0.24963
True gain
Regression fit / Expected gain
Money gained in dollars
Episodes
0
−200
−400
−600
−800
−1000
0
25
50
75
100
125
150
175
200


total reward per session for every beta
beta: 0.0
beta: 0.1
beta: 0.2
beta: 0.30000000000000004
beta: 0.4
beta: 0.5
beta: 0.6000000000000001
beta: 0.7000000000000001
beta: 0.8
beta: 0.9
total reward per session
Current session
−2
−3
−4
−5
−6
−7
−8
−9
0
25
50
75
100
125
150
175
200

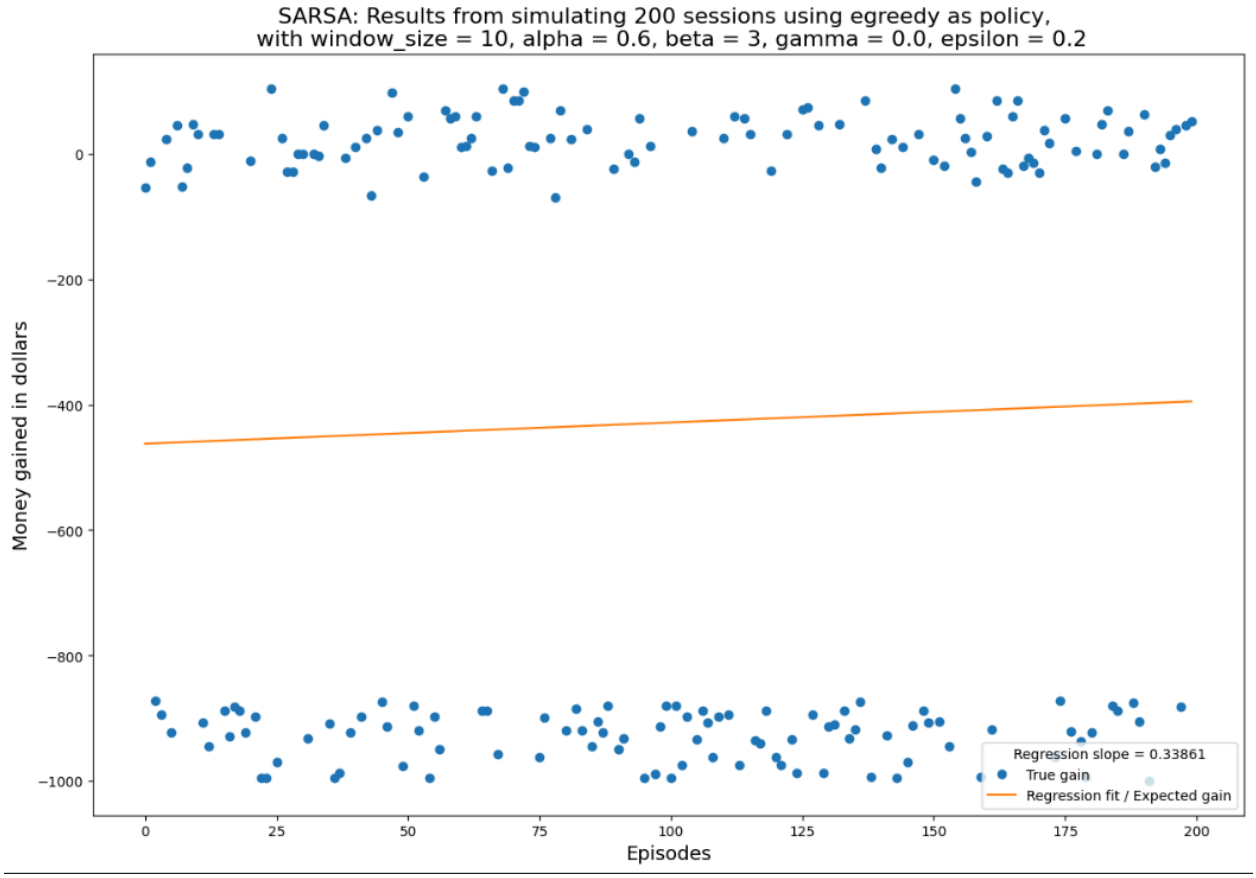
Epsilon-Greedy Policy

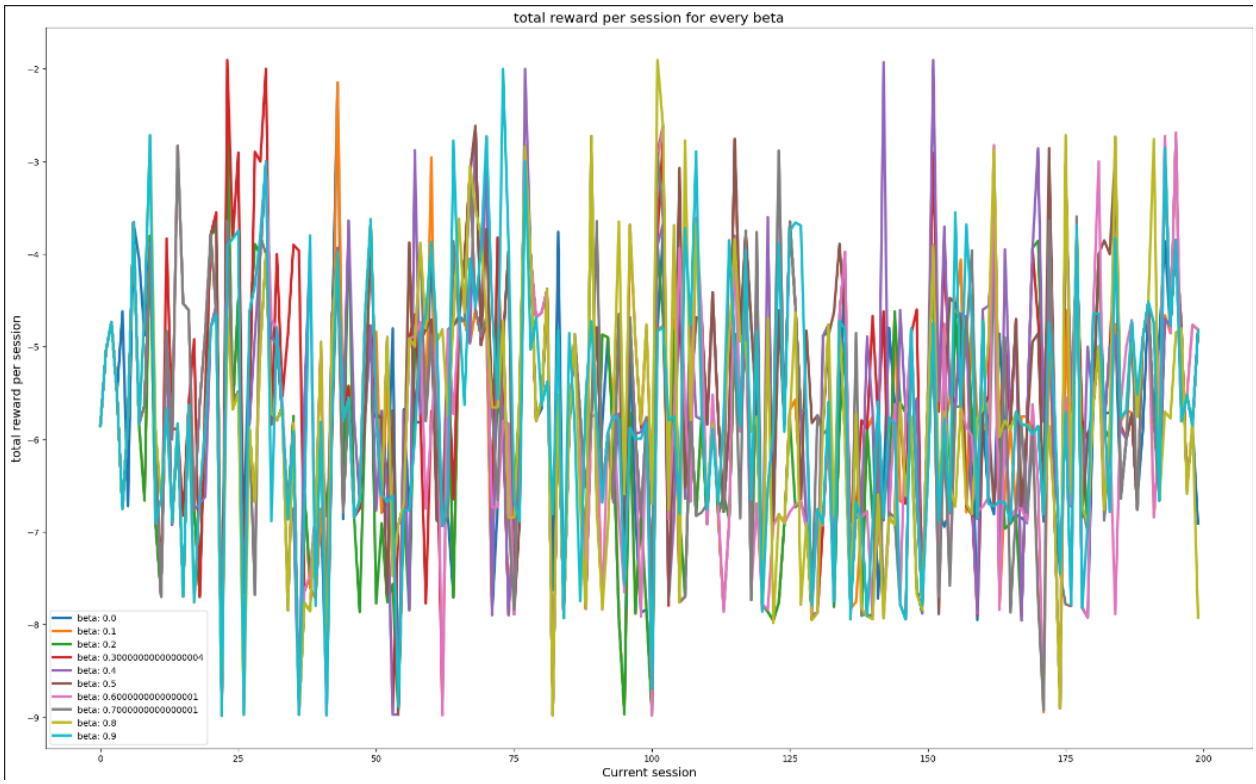
Gamma Parameter

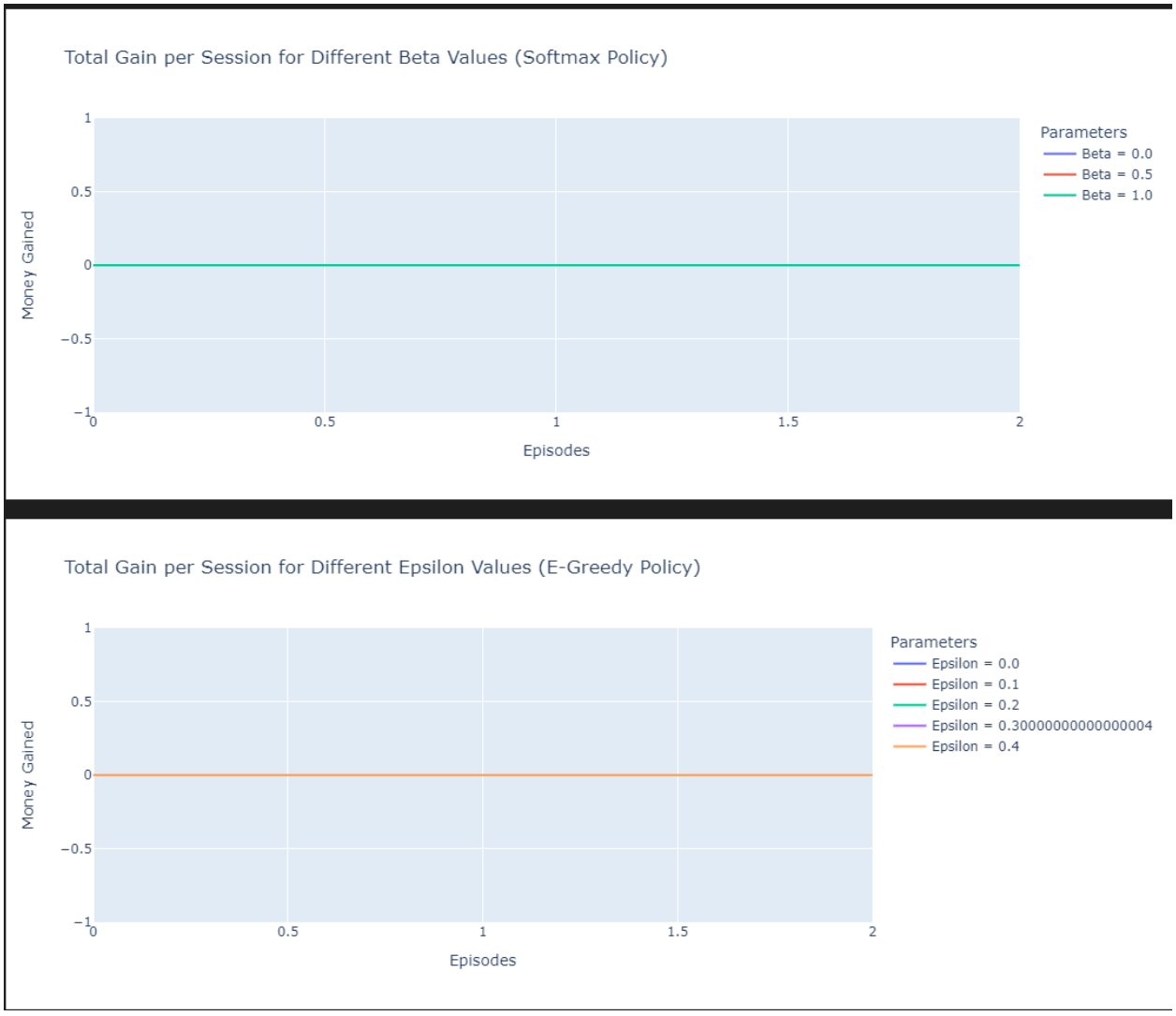
In this section, we evaluate the impact of different discount factors, represented by the Gamma (γ) parameter, on the performance of the Epsilon-greedy policy. The discount factor determines the importance of future rewards in the agent's decision-making process.

Overview of the Gamma Parameter: The Gamma (γ) parameter controls the weight given to future rewards relative to immediate rewards:

- **High Gamma (γ close to 1):** The agent places more emphasis on future rewards, encouraging long-term planning. This can lead to more strategic behavior but may slow down learning if future rewards are uncertain.
- **Low Gamma (γ close to 0):** The agent focuses more on immediate rewards, which can result in quicker learning but may overlook the benefits of long-term strategies.


SARSA: Results from simulating 200 sessions using egreedy as policy,
with window_size = 10, alpha = 0.6, beta = 3, gamma = 0.0, epsilon = 0.2
Money gained in dollars
Episodes
Regression slope = 0.33861
True gain
Regression fit / Expected gain


total reward per session for every beta
total reward per session
Current session
beta: 0.0
beta: 0.1
beta: 0.2
beta: 0.30000000000000004
beta: 0.4
beta: 0.5
beta: 0.6000000000000001
beta: 0.7000000000000001
beta: 0.8
beta: 0.9


Total Gain per Session for Different Beta Values (Softmax Policy)
Money Gained
Episodes
Parameters
Beta = 0.0
Beta = 0.5
Beta = 1.0
1
0.5
0
−0.5
−1
0
0.5
1
1.5
2

Total Gain per Session for Different Epsilon Values (E-Greedy Policy)
Money Gained
Episodes
Parameters
Epsilon = 0.0
Epsilon = 0.1
Epsilon = 0.2
Epsilon = 0.30000000000000004
Epsilon = 0.4
1
0.5
0
−0.5
−1
0
0.5
1
1.5
2

Conclusion: Hold Decision Based on Q-Learning and SARSA Results

1. Action and Reward Analysis:

Action Taken:

- **Action:** 0 (Sit)
- **Reward:** 0

The output indicates that the agent took the action to "sit" (which corresponds to holding the position), and the reward received was 0. This suggests that the agent's decision did not result in any tangible benefit or improvement under the given conditions.

2. Interpretation:

Negative or Zero Reward:

- **Reward of 0:** This implies that the action did not lead to any positive outcome. In this scenario, the agent's decision to "sit" did not yield any benefits. This could be because the current state did not present an immediate opportunity for buying or selling, or it might reflect a suboptimal decision based on the agent's current policy and state evaluation.

Action Evaluation:

- **Hold as Default Action:** The choice to "sit" might be a default action when the agent does not identify any immediate advantageous moves. While this action did not result in rewards, it may be a conservative strategy in uncertain or volatile market conditions. A lack of positive reward suggests that the state was not favorable for other actions, or the current strategy needs refinement.

3. Conclusion:

Why "Hold" is Recommended:

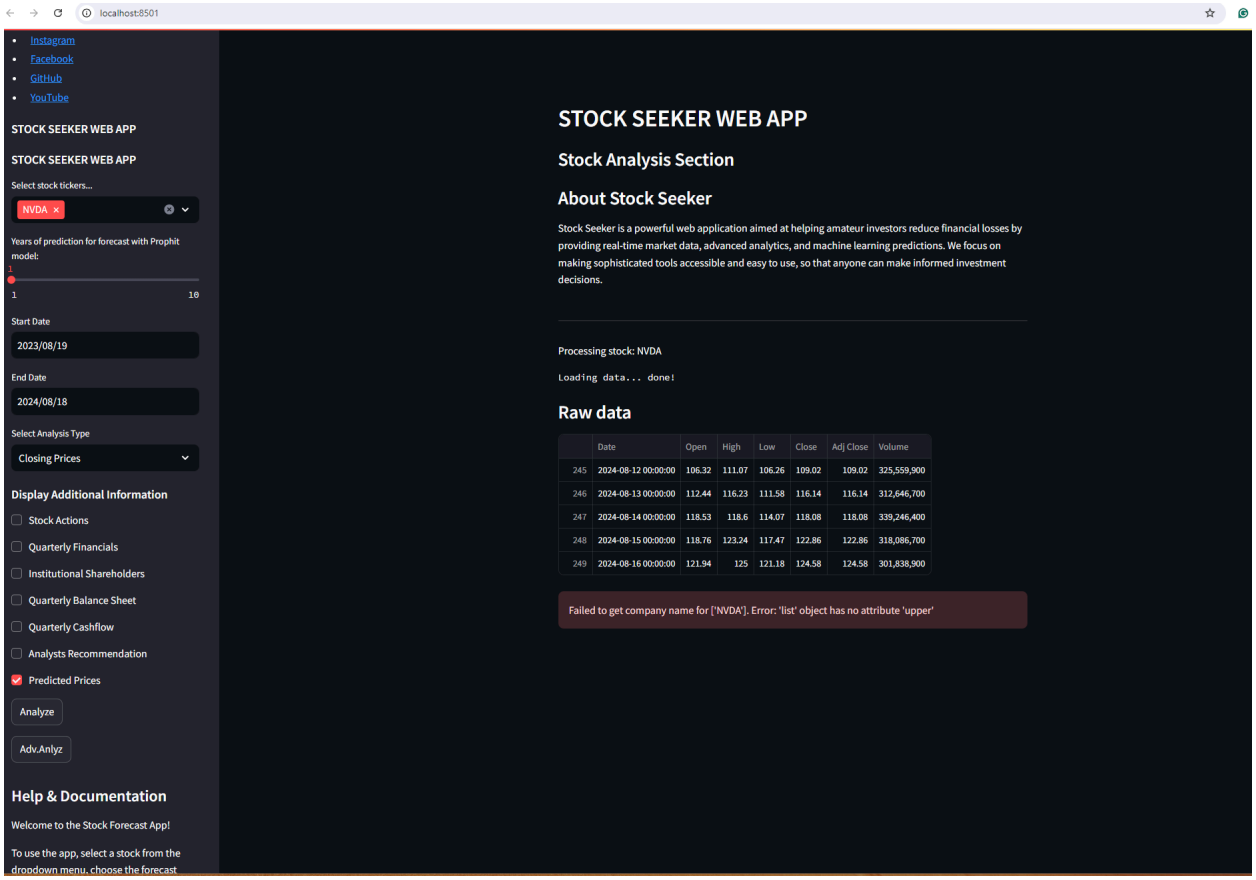
- **Risk Mitigation:** Given the absence of a positive reward and the potential suboptimal outcomes from other actions, maintaining the current positions (Hold) may be the most

prudent decision. This approach minimizes the risk associated with potentially unfavorable market conditions or incorrect action choices.

- **Policy and Strategy Improvement:** The zero reward highlights the need for continuous improvement in the agent's policy and learning parameters. Adjustments may be necessary to enhance decision-making and ensure that actions lead to positive outcomes in various market scenarios.

In summary, based on the results from Q-learning and SARSA, a "Hold" decision is recommended. The lack of positive rewards for other actions underscores the need for a cautious approach in uncertain market conditions. Maintaining existing positions helps manage risk while providing an opportunity to refine the agent's strategies for future decisions.

IV. STOCK FORECAST APP WITH AI LSTM:



- Instagram
- Facebook
- GitHub
- YouTube

STOCK SEEKER WEB APP

STOCK SEEKER WEB APP

Select stock tickers...

NVDA ×

Years of prediction for forecast with Prophit model:

1

1 10

Start Date

2023/08/19

End Date

2024/08/18

Select Analysis Type

Closing Prices

Display Additional Information

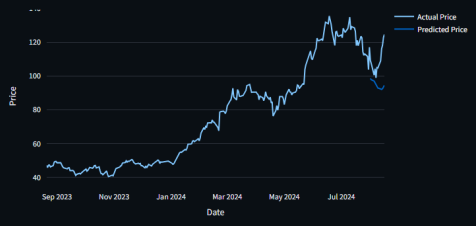
- [ ] Stock Actions
- [ ] Quarterly Financials
- [ ] Institutional Shareholders
- [ ] Quarterly Balance Sheet
- [ ] Quarterly Cashflow
- [ ] Analysts Recommendation
- [x] Predicted Prices

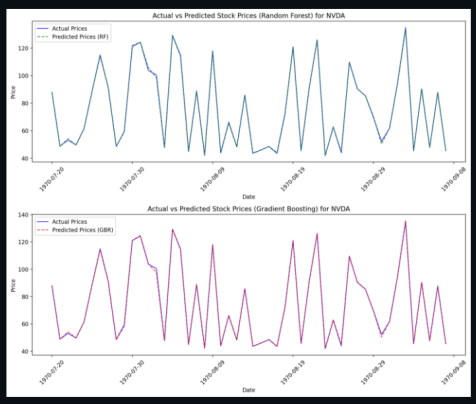
Analyze

Adv.Anlyz

Help & Documentation

Welcome to the Stock Forecast App!





Overall Model Performance

	Model	MSE	MAE	R2 Score	MAPE
0	LSTM	323.4626	15.4655	-3.9129	0.1354
1	Random Forest	0.4322	0.4237	0.9995	0.0058
2	Gradient Boosting	0.3901	0.3417	0.9995	0.0048

- Instagram
- Facebook
- GitHub
- YouTube

STOCK SEEKER WEB APP

STOCK SEEKER WEB APP

Select stock tickers...

NVDA ×

Years of prediction for forecast with Prophit model:

1

1 10

Start Date

2023/08/19

End Date

2024/08/18

Select Analysis Type

Closing Prices

Display Additional Information

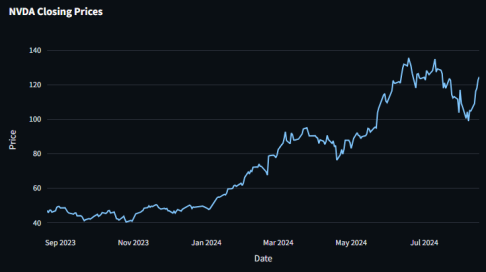
- [ ] Stock Actions
- [ ] Quarterly Financials
- [ ] Institutional Shareholders
- [ ] Quarterly Balance Sheet
- [ ] Quarterly Cashflow
- [x] Analysts Recommendation
- [x] Predicted Prices

RUNNING

248	2024-08-15 00:00:00	118.76	123.24	117.47	122.86	122.86	318,086,700
249	2024-08-16 00:00:00	121.94	125	121.18	124.58	124.58	301,838,900

Failed to get company name for ['NVDA']. Error: 'list' object has no attribute 'upper'

NVDA - Closing Prices



NVDA - Analysts Recommendation

	period	strongBuy	buy	hold	sell	strongSell
0	0m	7	14	15	1	1
1	-1m	18	36	5	0	0
2	-2m	20	33	4	0	0
3	-3m	12	21	12	2	0

Connect with Me

- LinkedIn
- X (Twitter)
- Instagram
- Facebook
- GitHub
- YouTube

STOCK SEEKER WEB APP

STOCK SEEKER WEB APP

Select stock tickers...

NVDA ×

Years of prediction for forecast with Prophit model:

1

1 10

Start Date

2023/08/19

End Date

2024/08/18

Select Analysis Type

Closing Prices

Display Additional Information

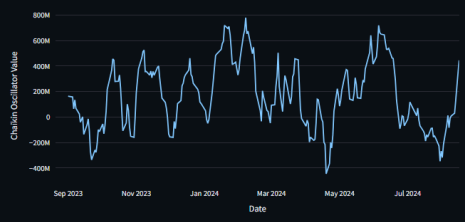
- [ ] Stock Actions
- [ ] Quarterly Financials
- [ ] Institutional Shareholders
- [ ] Quarterly Balance Sheet
- [ ] Quarterly Cashflow
- [x] Analysts Recommendation
- [x] Predicted Prices

Analyze

Adv.Anlyz

Help & Documentation

NVDA Chaikin Oscillator



NVDA Stochastic Oscillator

Connect with Me

- LinkedIn
- X (Twitter)
- Instagram
- Facebook
- GitHub
- YouTube

STOCK SEEKER WEB APP

STOCK SEEKER WEB APP

Select stock tickers...

NVDA ×

Years of prediction for forecast with Prophit model:

1

1 10

Start Date

2023/08/19

End Date

2024/08/18

Select Analysis Type

Closing Prices

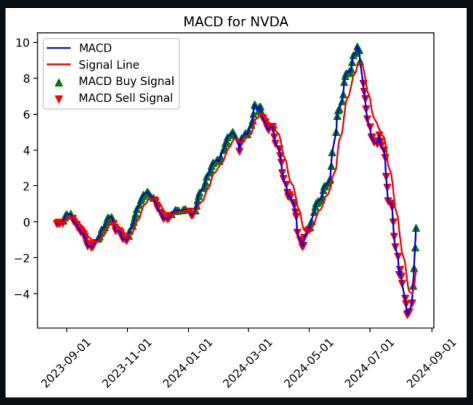
Display Additional Information

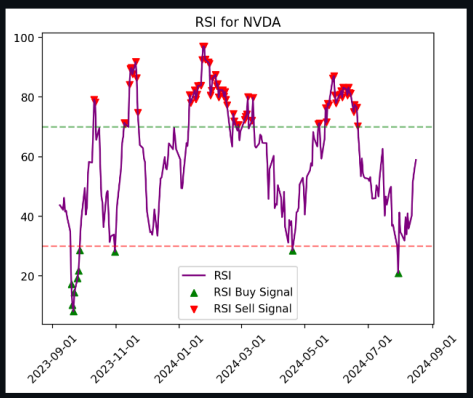
- [ ] Stock Actions
- [ ] Quarterly Financials
- [ ] Institutional Shareholders
- [ ] Quarterly Balance Sheet
- [ ] Quarterly Cashflow
- [x] Analysts Recommendation
- [x] Predicted Prices

Analyze

Adv.Anlyz

Advanced Analysis for NVDA





- LinkedIn
- X (Twitter)
- Instagram
- Facebook
- GitHub
- YouTube

STOCK SEEKER WEB APP

STOCK SEEKER WEB APP

Select stock tickers...

NVDA ×

Years of prediction for forecast with Prophit model:

1

1 10

Start Date

2023/08/19

End Date

2024/08/18

Select Analysis Type

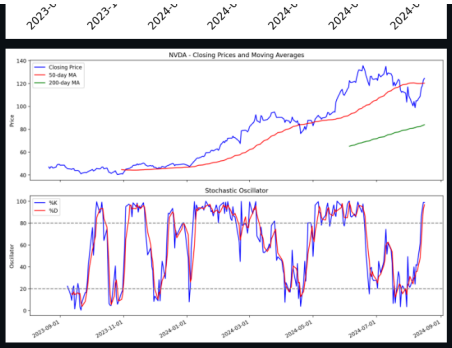
Closing Prices

Display Additional Information

- [ ] Stock Actions
- [ ] Quarterly Financials
- [ ] Institutional Shareholders
- [ ] Quarterly Balance Sheet
- [ ] Quarterly Cashflow
- [x] Analysts Recommendation
- [x] Predicted Prices

Analyze

Adv.Anlyz



Stock Trading Signals and EMAs

- Instagram
- Facebook
- GitHub
- YouTube

STOCK SEEKER WEB APP

STOCK SEEKER WEB APP

Select stock tickers...

NVDA ×

Years of prediction for forecast with Prophit model:

1

1 10

Start Date

2023/08/19

End Date

2024/08/18

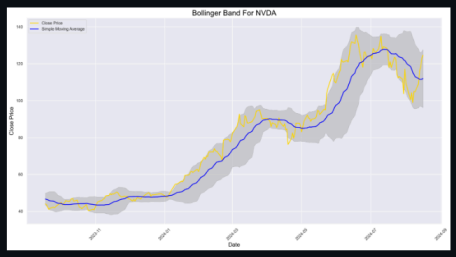
Select Analysis Type

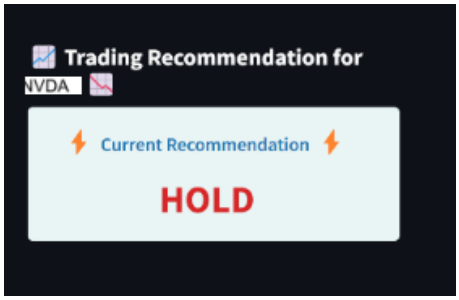
Closing Prices

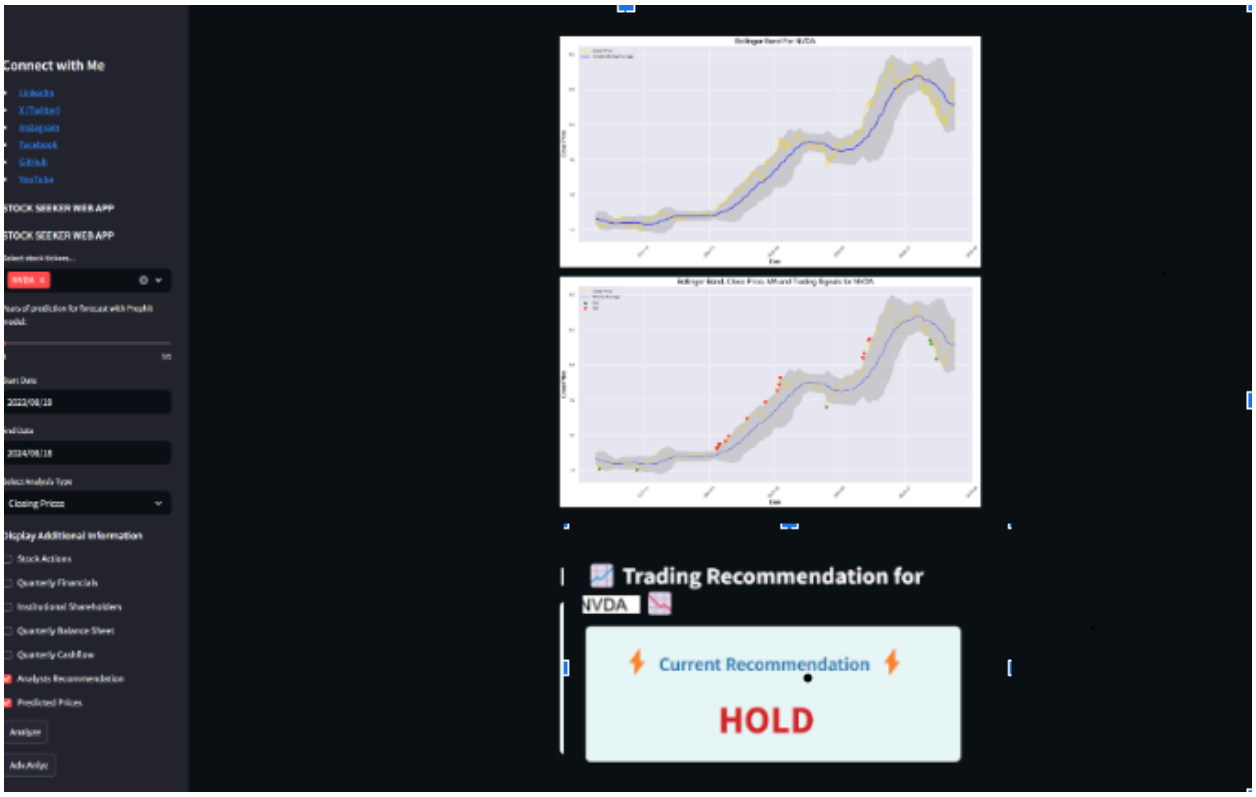
Display Additional Information

- [ ] Stock Actions
- [ ] Quarterly Financials
- [ ] Institutional Shareholders
- [ ] Quarterly Balance Sheet
- [ ] Quarterly Cashflow

Bollinger Bands and Trading Signals







V. Results of All Project

Web Application for Stock Market Analysis

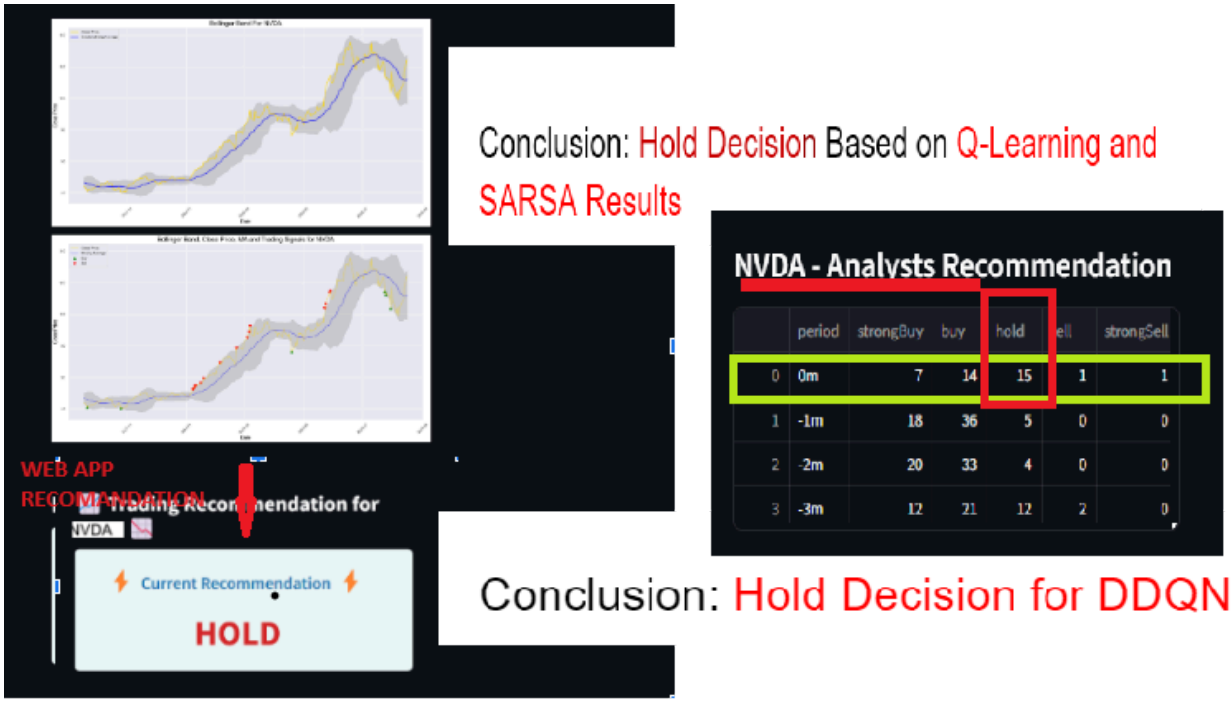
Our web application provides in-depth analysis of selected tickers, offering actionable insights to investors. For the specific case of NVIDIA (NVDA), the application has recommended a "HOLD" position. This recommendation aligns with the broader analysis indicating a high percentage of "HOLD" advice for this month across various analyses.

Key Findings:

1. **NVIDIA (NVDA) Recommendation:**
    - **Recommendation:** HOLD
    - **Analysis:** The deep analysis conducted for NVDA suggests maintaining the current position. This advice is based on the agent's evaluation using Deep Q-Learning (DDQN), SARSA, and Q-learning, all of which converge on the recommendation to "HOLD."
2. **General Analysis:**
    - **Market Trend:** This month's analyses consistently show a high percentage of "HOLD" recommendations across different methods.
    - **Agent Performance:** The DDQN, SARSA, and Q-learning models have been utilized to derive the recommendation for NVDA. Each method supports the "HOLD" position, indicating a stable recommendation across various reinforcement learning algorithms.

Conclusion:

The application effectively leverages reinforcement learning techniques to provide a consistent "HOLD" recommendation for NVDA, reflecting a cautious approach in the current market conditions. The use of DDQN, SARSA, and Q-learning models ensures robust and well-supported investment advice.



THANK YOU!

Contact info:https://www.linkedin.com/in/tessa-nejla-ayvazoglu/
nejlayvazoglu@gmail.com
https://github.com/TessaAyv79/CSCN8020RL