Next Best Action Model with Reinforcement Learning

Group 6

002789131	Praneeth Korukonda
002624652	Balaji Rao Bondalapati
002768046	Sandeep kotha

Introduction

What is a Next Best Action Model?
A Next Best Action Model is a crucial component of recommendation systems, decision support systems, and personalised marketing strategies. It focuses on determining the most favorable action to suggest to a user at a given point in their journey, considering various factors such as user behaviour, historical data, and contextual information. This action could be recommending a product, suggesting a course of action, or any decision that enhances the user experience or achieves specific objectives.

Reinforcement Learning and Fitted Q Iteration
In the context of next best action modelling, reinforcement learning is a powerful approach. It treats the recommendation problem as a Markov Decision Process (MDP) where the system learns to make decisions by interacting with an environment. Reinforcement learning algorithms aim to find the best policy, a strategy that maps states to actions, to maximise cumulative rewards.
Fitted Q Iteration is one such reinforcement learning algorithm. It iteratively approximates the Q-function, which estimates the expected cumulative rewards of taking an action in a given state. This approximation allows the system to make informed decisions based on its learned knowledge.

Data

In this tutorial, we'll work with generated data, making it independent of external dependencies. This approach simplifies the demonstration and ensures that you can follow along without needing specific datasets.

Environment Simulator

To illustrate the concepts, we'll use an environment simulator similar to the one in the Dynamic Content Personalization example. This simulator creates user profiles, offers, and events, helping us understand how different actions affect user behaviour.
The environment parameters include events, offers, and demographic features, which can be customised to suit the specific scenario you're modelling. Users are divided into groups with varying preferences and characteristics.

Building the Next Best Action Model

We'll focus on creating a reinforcement learning model that learns the best actions for each user, taking into account their state (context) and historical data. The model aims to maximise rewards over time and provide personalised recommendations for users.

- **State Features**: We'll define the state features that represent a user's context. These may include demographic information, historical actions, and any relevant data that can influence decision-making.
- **Frame Rewards**: We'll calculate rewards for each time frame, helping the model understand the impact of different actions on user behavior. In our case, the rewards are based on user actions, such as purchases or visits.
- **Trajectories and Transitions**: We'll segment user interactions into trajectories and transitions, providing the model with sequences of state-action pairs. This data structure is essential for reinforcement learning.
- **Fitted Q Iteration**: We'll implement the Fitted Q Iteration algorithm to approximate the Q-function. This function helps determine the best action for each state, taking into account expected rewards and learned knowledge.
- **Optimal Action Selection**: Using the Q-function, we'll find the optimal action to recommend to a user in a given context. This action is chosen to maximize expected rewards, enhancing the user experience and achieving specific goals.

Step 1: Create a Simple User Behavior Simulator

In this step, we create a user behaviour simulator to generate synthetic user interaction data for the Next Best Action Model with Reinforcement Learning.

Environment Parameters:

- Events: We define three types of events: no action (0), a visit (1), and a purchase (2).
- Offers: There are three types of offers: advertisement (1), a small discount (2), and a large discount (3).

- Demographics: We consider two levels of sensitivity: low sensitivity (0) and high sensitivity (1).
- Number of Users (n): We simulate the behaviour of 1000 users.
- Time Intervals (k): Our simulation spans 100 time intervals.
- Number of Offers (m): We have three different offers.

Auxiliary Functions:

These functions are defined to assist in various aspects of the code

```
# Auxiliary functions
#
def multinomial_int(p):
    return np.where(np.random.multinomial(1, p) == 1)[0][0]

def count(ndarray, val):
    return np.count_nonzero(ndarray == val)

def index(ndarray, val, default):
    try:
        return ndarray.tolist().index(val)
    except:
        return default

def find_offer_times(f):
    return np.nonzero(f)[0]

def offer_seq(f):
    return f[np.where(f > 0)]
```

Environment Simulator:

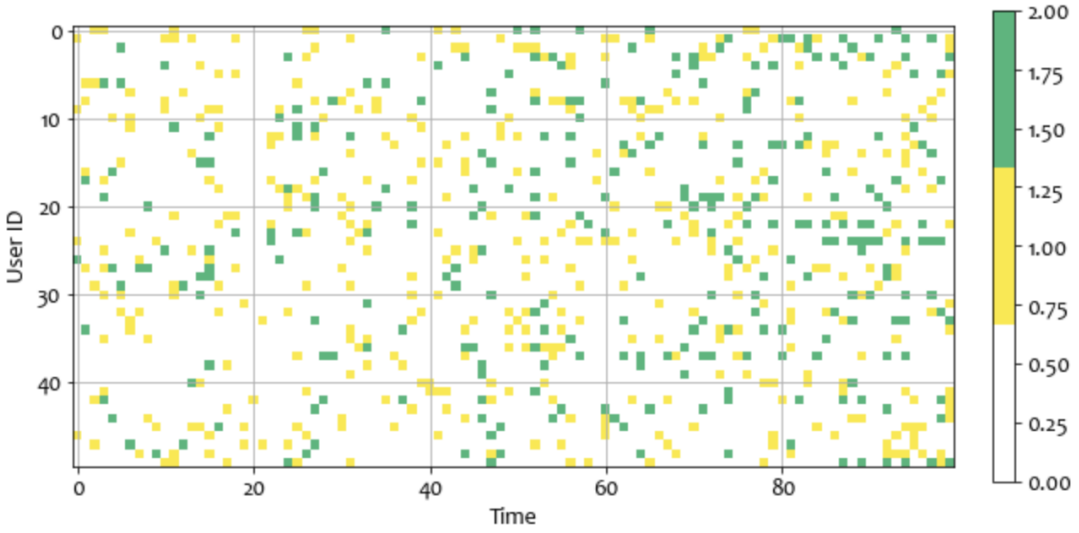
```
def get_event_pr(d, f):
    f_ids = offer_seq(f)
    f_ids = np.concatenate((f_ids, np.zeros(3 - len(f_ids))))

    if((f_ids[0] == 1 and f_ids[1] == 3) or
       (f_ids[1] == 1 and f_ids[2] == 3) or
       (f_ids[0] == 1 and f_ids[2] == 3)):
            p_events = [0.70, 0.08, 0.22]      # higher probability of a purchase
    else:
        p_events = [0.90, 0.08, 0.02]          # default probability distribution over events

    if(np.random.binomial(1, 0.1) > 0):        # add some noise
        p_events = [0.70, 0.08, 0.22]

    return p_events
```

```python
def generate_profiles(n, k, m):

    p_offers = [1 / m] * m                                  # offer probabilities (behavioral policy)
    t_offers = np.linspace(0, k, m + 2).tolist()[1 : -1]    # offer campaing times
    t_offer_jit = 5                                         # offer time jitter, standard deviation in time units

    P = np.zeros((n, k))                                    # matrix of profile events
    F = np.zeros((n, k))                                    # offer history
    D = np.zeros((n, 1))                                    # demographic features

    for u in range(0, n):
        D[u, 0] = np.random.binomial(1, 0.5)

        # determine m time points to issue offers for customer u
        offer_times_u = np.rint(t_offer_jit * np.random.randn(len(t_offers)) + t_offers)

        for t in range(0, k):                                   # simulate a trajectory for customer u
            if t in offer_times_u:
                F[u, t] = multinomial_int(p_offers) + 1         # issue an offer at time t according
                                                                # to the behavioral policy

            event = multinomial_int(get_event_pr(D[u], F[u]))   # simulate an event at time t
            P[u, t] = event

    return P, F, D
```

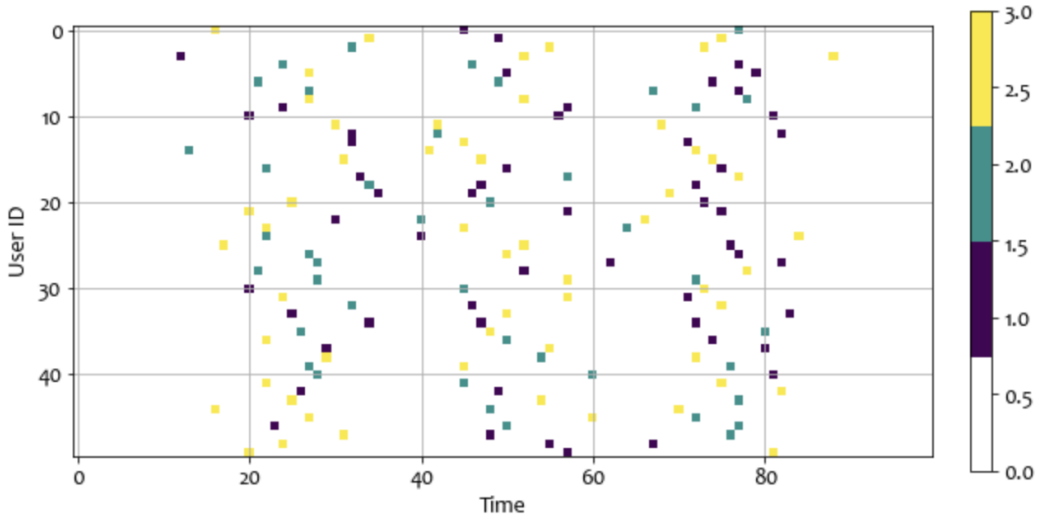
Visualising User Profiles:

The function visualises user profiles. This function creates a scatter to display how events are distributed over time for a subset of user profiles. Different colours represent different events, and the heatmap provides insights into user behaviour.

Training data:



Testing data:



That concludes Step 1, where we've created a user behaviour simulator to generate synthetic user interaction data. In the following steps, we will use this data to build and train the Next Best Action Model with Reinforcement Learning.

Step 2: Cut Trajectories into Transitions

In this step, we define functions to cut user trajectories into transitions. These transitions will be used to train and evaluate the Next Best Action Model with Reinforcement Learning.

```
#
# Cut trajectories into transitions
#
def state_features(p, f, d, t_start, t_end):
    p_frame = p[0:t_end]
    f_frame = f[0:t_end]
    f1 = index(f_frame, 1, k)
    f2 = index(f_frame, 2, k)
    f3 = index(f_frame, 3, k)
    return np.array([
        d[0],                   # demographic features
        count(p_frame, 1),      # number of visits
        f1,                     # first time offer 1 was issued
        f2,                     # first time offer 2 was issued
        f3                      # first time offer 3 was issued
    ])

def frame_reward(p, t_start, t_end):
    return count(p[t_start:t_end], 2)   # number of purchases in the time frame

def offer_time_ranges(times):
    rng = [-1] + times + [k]
    return list(zip(map(lambda x: x + 1, rng), rng[1:]))
```

- state_features(p, f, d, t_start, t_end): This function extracts state features at a given time frame. It collects demographic features, counts the number of visits, and tracks when each offer was issued.

- frame_reward(p, t_start, t_end): This function calculates the reward for a specific time frame. It counts the number of purchases made during that time frame.
- offer_time_ranges(times): This function determines the time ranges during which offers were presented in a user's trajectory.

```
def prepare_trajectories(P, F, D):
    T = []
    for u in range(0, n):
        offer_times = find_offer_times(F[u]).tolist()
        ranges = offer_time_ranges(offer_times)

        T_u = []
        for r in range(0, len(ranges)):
            (t_start, t_end) = ranges[r]
            state = state_features(P[u], F[u], D[u], 0, t_start)
            reward = frame_reward(P[u], t_start, t_end)
            state_new = state_features(P[u], F[u], D[u], t_start, min(t_end + 1, k))
            is_end_of_trajectory = t_end >= k

            if(t_end in offer_times):
                action = F[u, t_end]
            else:
                action = 1 # default action

            T_u.append([state, action, reward, state_new, is_end_of_trajectory])

        T.append(T_u)

    return np.array(T)

T = prepare_trajectories(P, F, D)
Tt = prepare_trajectories(Pt, Ft, Dt)
```

- prepare_trajectories(P, F, D): This function takes user profiles (P), offer histories (F), and demographic features (D) to create transition sequences for users. Each transition contains state information, actions, rewards, and information about the end of the trajectory.

In this step, we've successfully prepared the transition sequences that will be used for training and evaluating the Next Best Action Model with Reinforcement Learning. These transitions provide the necessary data for modelling user interactions and learning optimal actions.

Step 3: Policy Learning

In this step, we focus on policy learning by training a model to find the optimal action under a greedy policy and corresponding state value.

best_action(Q, state, actions)

- best_action(Q, state, actions) finds the action that maximizes the Q-value for a given state. It iterates through all possible actions, calculates their Q-values, and selects the action with the highest Q-value.

```
#
# Find the optimal action under a greedy policy and corresponding state value
#
def best_action(Q, state, actions):
    v_best = 0
    a_best = 0
    for a in actions:
        v = Q([np.append(state, a)])[0]
        if(v > v_best):
            v_best = v
            a_best = a

    return a_best, v_best

def Q_0(sa):
    return [1]

Q = Q_0
for i in range(1, 5): # FQI iterations
    X = []
    Y = []
    for sample in T.reshape((n * (m + 1), -1)):
        state, action, reward, state_new, is_end_of_trajectory = sample
        x = np.append(state, action) # feature vector consists of state-action pairs

        a_best, v_best = best_action(Q, state_new, offers)

        y = reward + v_best    # we ignore the initial conditions (ends of trajectories) for the sake of illustr

        X.append(x)
        Y.append(y)

    regr = RandomForestRegressor(max_depth=4, random_state=0, n_estimators=10)
    regr.fit(X, np.ravel(Y))
    Q = regr.predict
```

Fitted Q-Iteration (FQI) Loop:

- The main part of the code implements the FQI algorithm, which iteratively updates the Q-function to improve its accuracy. The loop runs for a specified number of iterations (in this case, for 4 additional iterations).

We are updating the Q-function using the Fitted Q-Iteration algorithm. The Q-function estimates the expected cumulative rewards for taking actions in specific states and is iteratively improved to find an optimal policy. This Q-function will later be used for policy evaluation and selecting the best actions in different states.

```
#
# Use the test set to evaluate the policy
#
states = Tt[:, :, 0].flatten().tolist()

values = []
best_actions = []
for s in states:
    a_best, v_best = best_action(Q, s, offers)
    values.append(v_best)
    best_actions.append(a_best)

s_tsne = TSNE(learning_rate = 100).fit_transform(states)
```

Now the trained Q-function (previously obtained through Fitted Q-Iteration) is used to evaluate a policy on a test set of states

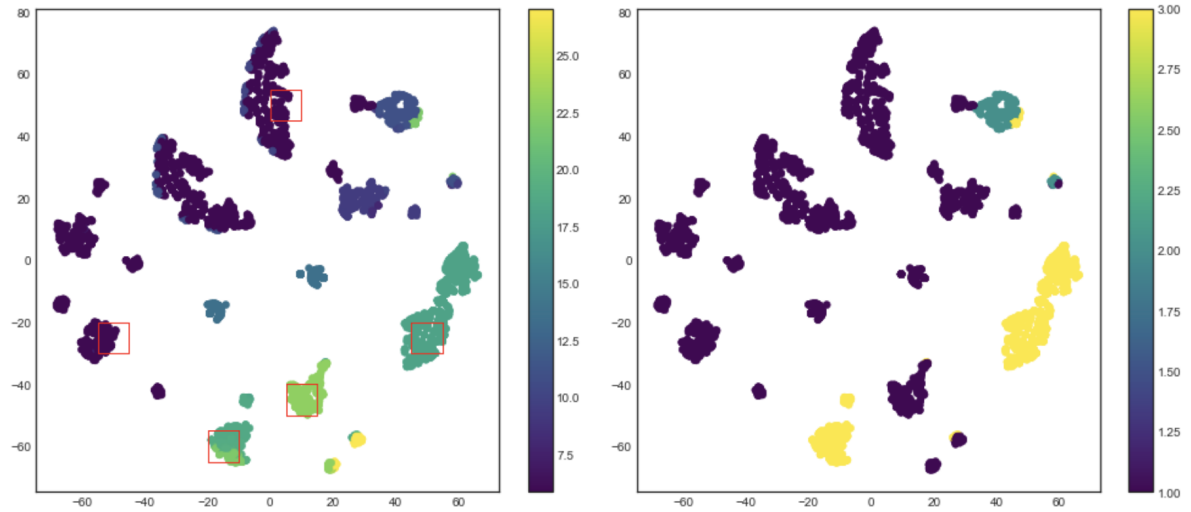
Flattened and extractacted the states from the test set Tt. These states represent the current conditions of the environment for each user in the test set.

For each state extracted from the test set, the code calculates two important pieces of information:

- **a_best**: The best action determined by the Q-function for the current state.
- **v_best**: The value associated with the best action in the given state.

Performed t-distributed Stochastic Neighbor Embedding (t-SNE) on the test set states to visualize them in a lower-dimensional space

Evaluated the performance of the policy derived from the learned Q-function on the test set. By calculating the best actions and values for the states in the test set, the code assesses how well the policy is performing in different scenarios. The t-SNE visualization helped provide insights into the distribution of states and the effectiveness of the policy in various regions of the state space.



In the first subplot (on the left), the code creates a scatter plot of the t-SNE projections of states. The color of each point in the scatter plot represents the calculated Q-value for that state. A color bar is added to the plot to indicate the range of Q-values, with a color map

In the second subplot (on the right), another scatter plot is created using the same t-SNE projections. This time, the color of each point represents the best action (action with the highest Q-value) for that state.

Step 4: Policy Evaluation

In this step, we evaluate the performance of the policy derived from the learned Q-function using Fitted Q-Iteration (FQI). We assess how different epsilon-greedy policies impact the overall return based on historical trajectories generated under a behavioral policy.

```python
def make_epsilon_greedy_policy(Q, eps):
    def egreedy_policy(state, action):
        a_best, v_best = best_action(Q, state, offers)

        if(a_best == action):
            return 1 - eps
        else:
            return eps / (m - 1)

    return egreedy_policy


# Probability of action given state
def behavioral_policy(state, action):
    return 1 / m

# Estimates the target policy return based on
# the profiles (trajectories) P generated under the behavioral policy
def evaluate_policy_return(T, behavioral_policy, target_policy):
    returns = []
    for trajectory in T:
        importance_weight = 1
        trajectory_reward = 0
        for transition in trajectory:
            state, action, reward  = transition[0:3]
            action_prob_b = behavioral_policy(state, action)
            action_prob_t = target_policy(state, action)

            importance_weight *= (action_prob_t / action_prob_b)
            trajectory_reward += reward

        returns.append(trajectory_reward * importance_weight)

    return np.mean(returns)
```

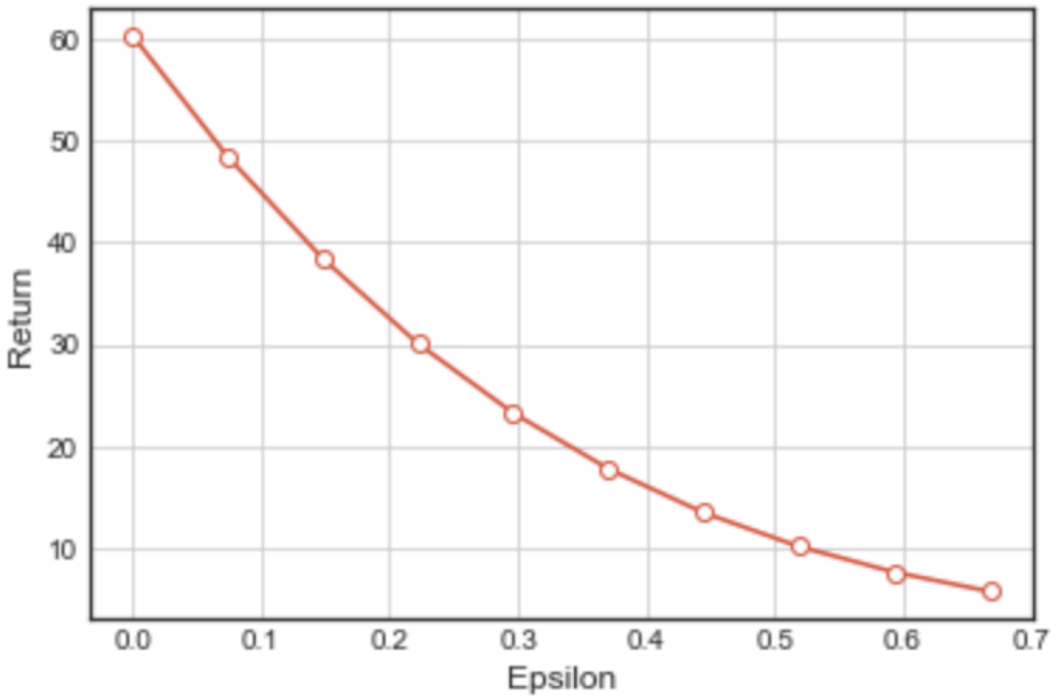
make_epsilon_greedy_policy(Q, eps):

This function creates an epsilon-greedy policy based on the learned Q-values. Epsilon-greedy is a common exploration strategy in reinforcement learning. It means that, with probability eps, the policy selects a random action (exploration), and with probability 1 - eps, it selects the action with the highest estimated Q-value (exploitation).

behavioral_policy(state, action):
This function defines the behavioural policy, which is typically the policy under which your historical trajectories (data) were collected. In this code, it's a uniform random policy, where each action has an equal probability of being selected (1 / m).

Evaluated how well the epsilon-greedy policy (target policy) performs compared to the random behavioural policy (used to generate the historical trajectories). By computing the expected return, you can assess whether the learned policy (epsilon-greedy) outperforms or matches the historical data generated under the behavioural policy.

We created a plot with epsilon values on the x-axis and policy returns on the y-axis. This plot helps us understand the trade-off between exploration and exploitation and choose an appropriate epsilon value for our policy.



Analysed how different levels of exploration (controlled by epsilon) impact the performance of the epsilon-greedy policy. The plot provides insight into the trade-off between exploration and exploitation in the context of reinforcement learning and helps in selecting an appropriate epsilon value for the policy.

Conclusion

- **Marketing and Advertising**: Implementing a Next Best Action model with reinforcement learning can significantly improve marketing and advertising strategies. It helps in deciding which ads or promotions to show to users, resulting in higher click-through rates and conversions.
- **Customer Relationship Management (CRM)**: Businesses can use this model to optimize their interactions with customers. It can suggest the most appropriate actions to take when dealing with a customer, improving overall customer satisfaction and loyalty.
- **E-commerce**: In e-commerce, the model can be employed to recommend the most relevant products or offers to customers, increasing the chances of purchase and boosting revenue.
- **Healthcare**: In healthcare, the Next Best Action model can assist in identifying suitable treatments and interventions for patients. For example, it can help in deciding which medical tests to recommend or which treatment plans to follow for specific patients.
- **Financial Services**: In the financial sector, this model can be utilized for making investment recommendations, suggesting financial products, or detecting fraudulent activities.
- **Content Recommendation**: Streaming platforms can utilize the model to recommend the next best content to users based on their viewing habits and preferences.
- **Manufacturing**: Manufacturers can optimize their production processes by determining the next best action at each step, thereby enhancing efficiency and reducing costs.

The Next Best Action Model with Reinforcement Learning offers a data-driven approach to dynamic decision-making, providing personalised recommendations and actions across diverse domains such as marketing, CRM, e-commerce, and healthcare. By continuously learning from historical data and iteratively refining its policy, it optimises user experiences and business outcomes. Its adaptability and effectiveness make it a valuable tool for organisations striving to enhance customer satisfaction and achieve their goals.