EURECOM
Sophia Antipolis

Semester Project Report
Fall 2020-21

Reinforcement learning for cache-friendly recommendations

Team

Cosimo Chetta
Valentin Nelu Ifrim
Aditya Mohan
Marco Vinai

Friday 12th February, 2021

Contents

List of Figures

List of Tables

1. Introduction

1.1 Motivation

The streaming experience of the users for content that is delivered over a Content Delivery Network (CDN) highly depends on the network cost that is associated with the content being delivered. Recent works in the Networking community have shown that it is possible to significantly decrease the network load at peak traffic times by nudging the users towards content that is less costly (in terms of delivery) for the operator [5]. Traditionally, this is done by understanding the content that is relevant to the user. However, this is a two-way streak since the content that is recommended to the user influences his behavior. Thus, influencing the users towards low-cost content has recently gained momentum as a strategy to boost the network performance. This can be formulated as a Reinforcement Learning problem viewing the recommender system as an agent that needs to understand the user behavior and recommend content to the user that is both, relevant to the user and also cached. This is called as Network Friendly Recommendations. To achieve this, we use Markov Decision Processes (MDPs) to model the user with a random session length. This approach is highly flexible and allow modeling users who are reactive to the quality of the received recommendations. However, extending this approach to large catalog sizes and multiple recommendations suffers from the curse of dimensionality. Thus, solving this requires developing approaches specifically focused on being able to achieve this, but in a tractable way.

1.2 Main Objectives

The main aim of this semester project was to design a recommender system which can keep the user satisfied with the suggested recommendations and reduce the network cost. The issue with the known vanilla tabular RL, that we tackled in our work, was that the catalogue size can reach thousands items, resulting in action spaces that are extremely large, and a tabular search through these action-spaces increases the convergence time by a lot.

Our work was split into two main parts:

- Find an approximation that would reach convergence faster than the vanilla Q-Learning methods;
- Extend the vanilla Q-learning method to multiple recommendations and test a new method that achieves similar results more effectively by splitting this update to multiple simultaneous updates.

2. Theoretical Background

2.1 Markov Decision Processes (MDP)

One way to look at the behavior of agents [9] is by looking at how they interact with their environment, and how this interaction allows them to behave differently over time through the process called learning. In this view, the behavior of the agent can be modeled in a closed-loop manner through a fundamental description of the action and sensation loop. The agent receives input - sensation - from the environment through sensors, acts on the environment through actuators, and observes the impact of its action on its understanding of the environment through a mechanism of quantification in the form of the rewards it receives for its action. Reinforcement Learning (RL) is a paradigm in Machine Learning that uses this model of the agent-environment interaction to capture the important aspects of the problem facing a learning agent interacting with its environment to achieve a goal. It allows the agent to learn from the consequences of its actions, rather than from being explicitly taught. The methods in Reinforcement Learning are closed-loop because the learning system's actions can influence its later inputs, by affecting not only the immediate reward but also the next situation and, consequently, subsequent rewards. Being closed-loop, having no explicit instructions on the actions to take, and the play of the action-reward loop are what distinguish RL from Supervised or Unsupervised Learning. [1]

The Reinforcement Learning Model is summarized in Figure 1. The Box pertaining to the Environment can also be interpreted either as the world (when the agent has the ability to observe the world fully), or as the agent's understanding of the world (In the case when the agent has a limited partial-observability of the world). The world has a lot of attributes that might influence, and be influenced, by the agent's action. Thus, a method to model these attributes is highly useful since it allows the agent's behaviour to be interpreted in the form of transitions. This can be further formalized by defining the observations as a certain *State* in which the agent might exist. The state might either be equal to the observation, or be a subset of the observation, depending on the problem.

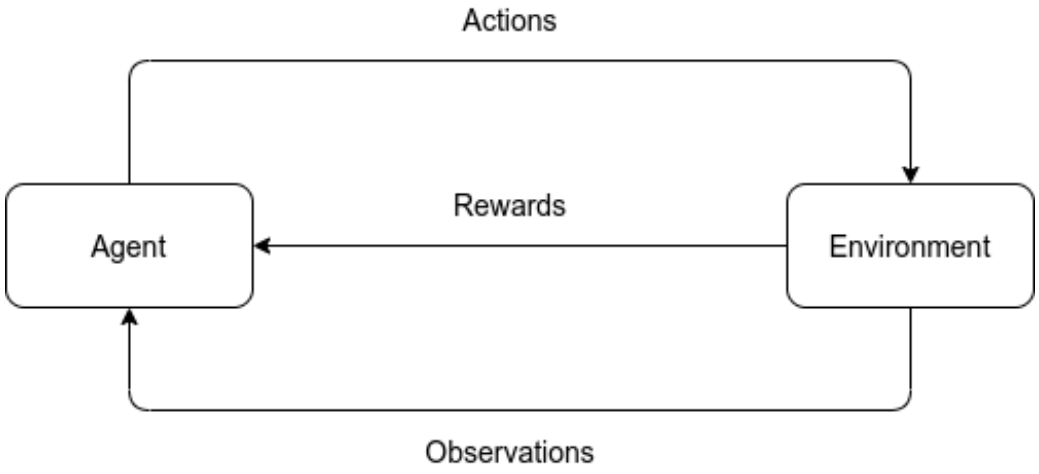


Figure 1: Agent-Environment Model

Markov processes are random processes indexed by time and are used to model systems that have limited memory of the past [7]. The fundamental intuition behind Markov processes is the property that the future is independent of the past, given the present. Thus, the present state of the agent can be conditioned on a limited number of previous states, and not the whole history of its states or actions. This *window* over which the state is conditioned determines the order of the

Markov Process. For first-order processes, a state only depends on the previous state, and so the conditional equation can be written as shown by Equation 1.

$$P(S_{t+1}|S_t) = P(S_{t+1}|S_1, ..., S_t) \tag{1}$$

Using this property - called the *Markov Property* - the Environment can be modeled as a Directed graph in which the nodes represent the states in which an agent can exist and the edges represent the transitions between these states. When for each state, there are multiple transitions possible, the agent has to decide which transitions to follow, and this is called a Markov Decision Process (MDP) [4]. An example of such a process is shown in Figure 2.

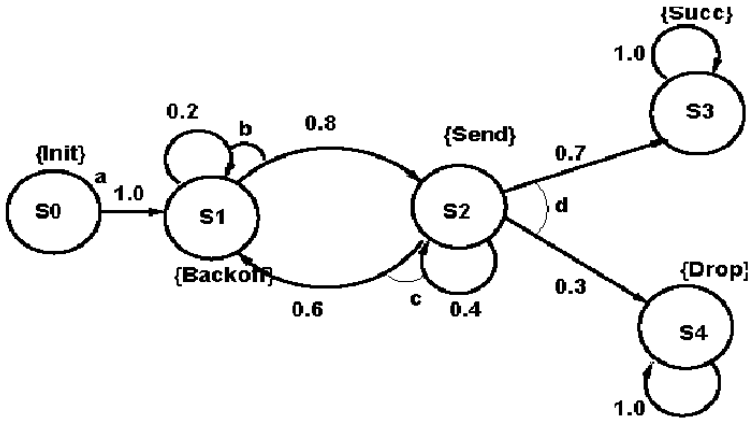


Figure 2: Example MDP. Source:[8]

Now, the RL problem can be reformulated as a solution to this MDP by formulating a mechanism to maximize the reward.

2.2 Value Iteration

The MDP can be formulated as $< S, A, T, \gamma, R >$ where,

- **S**: Set of all possible states.
- **A**: Set of all possible actions.
- **T**: Transition Dynamics of the system where $T(s'|s,a)$ is the probability density of the next state being s'.
- **R**: Reward for each transition.
- γ: The factor that discounts future rewards.

The schema that the agent follows to make its decisions in an MDP to accomplish its goal is called a policy. Thus, a policy is a mapping from states to actions and can be understood as the probability of taking a certain action in a given state, formulated as shown in Equation 2.

$$\pi(a|s) = P(A = a_t|S = s_t) \tag{2}$$

The discounted return of a roll-out can be defined as shown in Equation 3. This is a measure the rewards accumulated from the current state into the future. The value of γ controls the shadow of the future that the agent uses to measure the impact of future rewards.

$$G_t = \sum_{i=1}^{\infty} \gamma^{i-1} R_{t+i} \tag{3}$$

The value of a state and action under any policy π can be written as shown in Equations 4.

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t | s_t = s] \\ q_\pi(s,a) &= \mathbb{E}_\pi[G_t | s_t = s, a_t = a] \end{aligned} \tag{4}$$

The goal of the agent is to learn an optimal policy π^* that maximizes the expected discounted return G_t defined in Equation 5. This means that when the agent operates under the policy π^*, then it selects the next states in a way so as to maximize the expectation of the discounted rewards from that state, into the future.

$$\pi^* = \arg\max_\pi \mathbb{E}_\pi[G_t] \tag{5}$$

The agent does this either by iterating over the value of the states and actions - Value Iteration - or by directly iterating over the learned policies - Policy Iteration. In value iteration, the formulation of the optimal policy can be made recursively using the Bellman Update [1], as shown in Equation 6.

$$\begin{aligned} v^*(s) &= \max_a \mathbb{E}[r + \gamma v^*(s_{t+1}) | s_t = s] \\ q_\pi(s,a) &= \mathbb{E}[r + \gamma \max_{a_{t+1}} q^*(s_{t+1}, a_{t+1}) | s_t = s, a_t = a] \end{aligned} \tag{6}$$

2.3 Q-learning

An algorithm used in RL is Q-Learning. It is a *model-free* algorithm that aims at learning the quality of an action, more precisely the q function shown in Equation 6. It is defined as a model-free algorithm since it does not require a model of the environment. The optimal policy is found iteratively. First it takes action a on state s (such that $a = \arg\max_a q(s,a)$ in the given state), note the reward r and the next state s', then it chooses the max Q-Value in state s' and eventually use all these info to update $q(s,a)$, then move to s' and execute epsilon greedy action which does not necessarily result in taking action that has the max Q-Value in state s'. More in detail the algorithm is shown

in Algorithm 1.

Algorithm 1: Q - learning

Algorithm parameters: step size $\alpha \in (0,1]$, small $\epsilon > 0$;
initialize $Q(s,a)$, for all $s \in \mathbf{S}, a \in \mathbf{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$;
for *each episode* **do**
    Initialize s
    for *each step of episode* **do**
        Choose a from s using policy derived from Q (e.g., ϵ-greedy);
        Take action a, observe R, s';
        $Q(s,a) = Q(s,a) + \alpha[R + \gamma \max_a Q(\phi(s', a')) - Q(s,a)]$
    until s is terminal

3. Function Approximation

When the Reinforcement Problem is still small (a small number of states and actions) it is possible to save the the q-Value in tables, in those cases it is possible to talk about Tabular methods.

3.1 Theory

In a real scenario, the catalogue size can scale up to hundreds of contents. The Q-learning tabular approach struggle to find an optimal in acceptable times thus a different approach is needed to compute the $q(s,a)$ value. In this problem, as mentioned in Section 1., the states are the videos currently playing while the actions are the videos suggested. Therefore in case of a scenario with n videos as catalogue size, the possible actions are $n-1$. With Function Approximation it is possible to approximate the Q-table with a model that takes as input a feature representation $\phi(d)$, where d can be either a state or a state-action tuple and ϕ a function that return a particular encoding. In literature with functions approximation is possible to take two approaches: one is approximating the Value function of a particular state or approximating the q-value. Being more explicit the *Value* of a state is the optimal recursive sum of rewards form that state following the optimal policy π^*. The *q-value* instead refers to a pair state-action and is the optimal sum of discounted rewards (γ as discount parameter) associated with such such input pair. This is easily summarized by Equation 7.

$$v^*(s) = \max_a q^*(s,a) \tag{7}$$

Such model can be complex as preferred, but usually is better to start with a model as simple as possible. On the contrary a overly complicated model could overfitts the the current scenario, making it hard to understand some behaviours, and also harder to train. In fact going from a small linear model to a complex Deep Neural model would require much more time to train and different techniques. In this project some experimentation were also done with some small fully connected networks, but it was observed that linear approximation would reach good performance depending on the function ϕ which was one of the main focuses in this project.

3.2 Parameters

3.2.1 Agent Parameters

The models proposed for the function approximation are all united by some common parameters that are listed below.

- **Learning rate** lr: learning rate used to update the weights of model, this parameter had to be tuned depending on the agent, a value to big would bring the model to diverge while a value to small would not bring convergence in reasonable times.
- **Discount** γ: discount factor used to adjust the long term reward considered by the agent. It is a value between 0 and 1 where if zero the agent would be shortsighted, unaware of future rewards. Increasing γ would allow the model to consider also future rewards.
- **Exploration rate** ϵ: probability that the agent will explore the environment rather than exploit it. This parameter is part of the $\epsilon - greedy$ strategy where with probability $1-\epsilon$ the

action taken is the one with higher q-value from that state (exploitation), while with probability ϵ the action suggested by the agent will be pandemic (exploration).

3.2.2 Environment Parameters

The Environment used through all the testing is the user that decide if he would follow or not the suggestion of the agent with a certain probability. The environment used is inherited from the previous group working on this project. The main parameters of such environment are listed below.

- **Number of states** N_s: Number of items in the catalogue **S**.
- **Number of actions** N_a: Number of possible action that can be taken from state s. It's equal to $N_s - 1$ given that the agent can not recommend the current state.
- **Cached items** N_c: Number of items cached: a cached item has a lower cost on the network load.
- **Related item** N_r: Number of items correlated with each state.
- **Leave** l: probability to leave the experience thus to end the current episode.
- **Follow recommendation** α: parameters used to decide whether or not to accept the suggested item. Given user type, the parameters is used in the following way:
  - **Random**: accept the item with probability α;
  - **Quality aware**: accept the item only if $U(s,a) > \alpha$ (in words the correlation between state s and action suggested a is greater than a threshold).
- **Reward type**: reward type used to compute the score of the agent suggestion. Given R_{corr} and R_{cache} respectively the reward for a correlated item and a cached item, the final reward can be computed in two different ways:
  - **Constant**: return a fixed reward based on the item characteristics
    $$R = \begin{cases} 0, & \text{if } a \text{ is not correlated nor cached} \\ R_{corr}, & \text{if } a \text{ is correlated} \\ R_{cache}, & \text{if } a \text{ is cached} \\ R_{corr} + R_{cache}, & \text{if } a \text{ is correlated and cached} \end{cases}$$
  - **Variable**: takes in account the actual correlation $U(s,a)$
    $$R = \begin{cases} 0, & \text{if } a \text{ is not correlated nor cached} \\ R_{corr}, & \text{if } a \text{ is correlated} \\ R_{cache}U(s,a), & \text{if } a \text{ is cached} \\ R_{corr}U(s,a) + R_{cache}, & \text{if } a \text{ is correlated and cached} \end{cases}$$

3.3 Benchmark

In order to evaluate the performances of the approximation, the first thing coming to mind is getting the full Q-table and compare it with the tabular result.

Cached	Recommended	State representation	State-Action Representation
T	T	0	0
T	F	0	1
F	T	0	1
F	F	1	1

Table 1: $computePenalty$ values

The optimal ground truth used through all the experiments an agent is implemented using the Tabular Q-Learning developed by the Previous Group working on this problem. Such method is able to converge to the optimum, but its convergence times were getting higher too quickly when increasing the catalogue size. Since it is not fundamental to reach the same Q-Table for all agents, but what really matters is the policy learned two metrics to evaluated the goodness of the agent are proposed: Average Reward and Penalty. These two metrics are calculated with a benchmark function that runs for N episodes, with one episode corresponding to the environment continuously running with a probability l to terminate after each suggestion.

The benchmark function is shown in Algorithm 2. Note that d can either be the state s_t or the state-action tuple (s_t, a_t), according to the representation better explained in the following sections.

Algorithm 2: Benchmark

$Reward = 0, Penalty = 0$;
for $episode \leftarrow 0$ **to** N **do**
  reset s_0;
  $done = false$;
  $i = 0, R = 0, P = 0$;
  while *not* $done$ **do**
    select $a_t = \arg\max_a Q(\phi(d))$;
    Take action a_t, observe $(r_{t+1}, s_{t+1}, done)$;
    $I = i + 1,\ R = R + r_{t+1},\ P = P$+computePenalty$(r_{t+1})$;
  $Reward = Reward + R/i,\ Penalty = Penalty + P/i$
$Reward = Reward/N,\ Penalty = Penalty/N$

The Penalty score represents how many times the model is not able to suggest a good action, where the goodness differ for the two representation types as can be seen in table 1

3.4 State Representation

The first approach was to use an approximation of the value function, such that starting from state s the approximation model $f_v(\phi(s))$ would output a vector v_a of size $(n \times 1)$, each component suggesting the the q-value of the pair (s, a_i). This would allow to easily get the value of such state s and pick the action that had maximum value, always with a $\epsilon - greedy$ policy. The first step the was to find a good function ϕ to represent each state, the attempted procedure are shown in the following Section 3.4.1.

3.4.1 State Feature

The attempts made to properly represent the states are listed below.

ID Each state is represented by a integer $i \in [0, N_a - 1]$.
Number of features: 1

One hot Each state is replaced by an array of zeros except for one entry of the array: in particular the i^{th} state will be represented by an array of n elements with the i^{th} entry equal to one.
Number of features: N_s

U Since a state is highly dependent on the states that are related to, each state is represented by the corresponding row of the correlation matrix between the states. This matrix is referred as $U(N_s \times N_a)$ with zero diagonal: in particular the i^{th} state becomes $U[i,:]$ (in Python notation). This array has real values between 0 and 1, and 0 for $U[i,i]$.
Number of features: N_s

U ceil Same representation as fore in *U-feature encoding* but this time instead of having real numbers inside the arrays the state s_i is a vector having having the j^{th} entry set to 1 if $U[i,j] > 0.5$ and 0 otherwise $\forall\, j \in [0, n]$.
Number of features: N_s

Reward Each state is represented as an array having the same size as the number of states. This idea starts from the *U Ceil-feature encoding* representation but then the j^{th} element of the s_i states is increased by one if the j^{th} state is also cached.This is the first encoding also taking in consideration the possibility of caching.
Number of features: N_s

Valuable This representation is another step forward from the previous encoding. Having the state s_i represented with an array as before it is then increased according to the following rule: the j^{th} element of the array is increased with the number of cached elements related to the j^{th} state. A state is related to another if its correlation value is bigger than 0.5.
Number of features: N_s

With those $\phi(s)$ defined, the model to be trained f_v is a linear model is explicitly defined in Equation 8.

$$\begin{gathered} f_v(\phi(s)) = C\phi(s)^T + b^T \\ C \in \mathbb{R}^{n \times |\phi(s)|},\ b \in \mathbb{R}^n \end{gathered} \tag{8}$$

The focus is to find a a combination of weights C and b to get good approximation of f_v: how the goodness of such model defined is explained later. Now the iterative training procedure is shown in Algorithm 3.

Algorithm 3: Q - Memory Replay training

initialize $memory$ with capacity C;
initialize $model$ with random weights;
for $episode \leftarrow 0$ **to** N **do**
  reset s_0;
  $done = false$;
  while *not* $done$ **do**
    With probability ϵ select random action a;
    otherwise select $a_t = \arg\max_a Q(\phi(s_t), a)$;
    Take action a_t, observe $(r_t, s_{t+1}, done)$;
    Add to $memory$ (s_t, a_t, r_t, s_{t+1});
  for $i \leftarrow 0$ **to** M **do**
    $batch \leftarrow minibatch(memory)$;
    $\bar{y} = [\,]$;
    for $(s_i, a_i, r_i, s_{i+1}) \in batch$ **do**
      $y_i = r_{i+1} + \gamma \max_a Q(\phi(s_{i+1}), a')$;
    AdamW optimization on $(\bar{y} - Q(\phi(\bar{s}), \bar{a})^2$;

3.4.2 Memory Replay

In Algorithm 3 it also mentioned Memory Replay. This technique is usually used in Deep-Learning problems and it is also known as *Experience Replay*. Instead of running a Q-learning algorithm on the state-action pairs as they occur during simulation, the systems stores the data discovered in a tuple $(state, action, reward, next\ state)$ in a table referred as *memory*. During this stage no calculation are done, but just simple simulation are run. The *memory* is a fixed size list of experience tuples FIFO behaviours: once the limit of is exceeded new experiences are added while the older one are discarded. Once are enough samples in the memory the learning phase starts sampling randomly M mini batches of experiences and the weighs of the model described by the Equation 8 are updated: in our experiments we used AdamW optimizer that yielded slighter better results then the Gradient Descent. This process is repeated N times (each time is called an Episode). The memory gets updated with new experiences at each episode because improving the policy will lead to different behaviour that should explore actions closer to optimal ones, the one that are more interesting for the model.
The *Memory Replay* de-correlates single experience making each on as an i.i.d. since during the training they are randomly sampled: this usually results in better convergence but in this simple case did not bring the expected outcome.

3.4.3 Python Implementation

All the data structures, from the the Memory to the single states were done with the *Numpy* library and some basic structures such are lists, to store data during training. For the model instead it was used the *PyTorch* library. In particular the class representing the function approximation was done via the *NN* module, creating a net of only one layer, without activation function, obtaining a result like the one described in Equation 8. The advantage of such approach was having a model

that dealt with the updates of the weights indifferently from the optimizer used through the back-propagation implemented by the *PyTorch* library.

3.4.4 Results

The Figures 3 and 4 shows a comparison of the state representations methods alongside with the $Random$ agent (suggest actions randomly) and the $Correlated$ agent (suggest the most correlated item from that state). The following Table instead lists the parameters used for training.

N_s	N_r	N_c	Train episodes	ϵ	Reward type	R_{cache}	R_{corr}	User type
50	10	5	3000	0.3	constant	1	1	Random

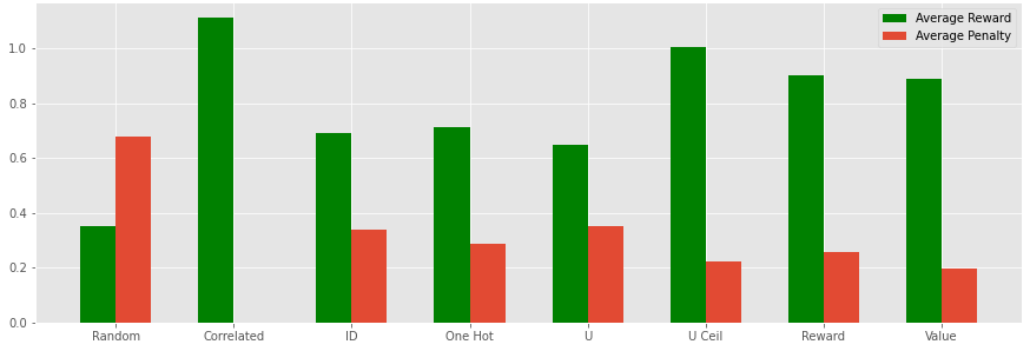


Figure 3: $\gamma = 0$

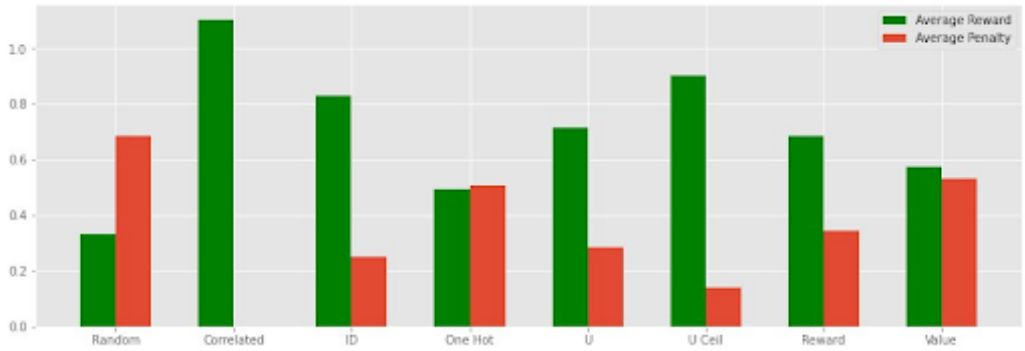


Figure 4: $\gamma = 0.9$

As can be seen in Figure 4, the state models surpass the random agent for both reward (the higher the better) and penalty (the lower the better). Unfortunately these representations never reach rewards bigger than the Correlated Agent. In fact both with γ equal to 0 and 0.9 the following formulations were giving unsatisfactory results. In order to seek for better results it was decided to go for a more straight forward approach since those models were not even able to reach the performance of a simple Agent suggesting just the more correlated items. In the following section is described how the q-value of each pair function is computed.

3.5 State-Action Representation

3.5.1 State-Action Feature

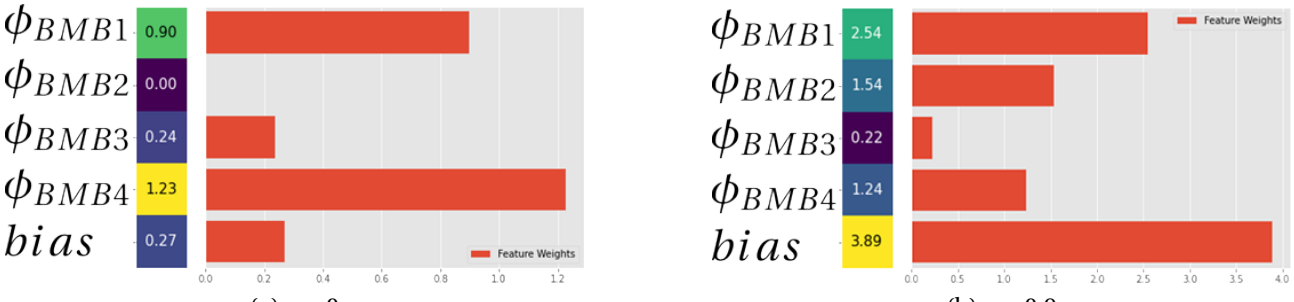
A different approach to approximate the behaviour of the Tabular agent is to model directly the q-value of the state-action pair. The ϕ representation now takes (s,a) as input and the model produces $q(s,a)$ with f_q.
Different models were built trying to explore possible features and analyze the result obtained also analyzing the weight of such feature in the final result. Reaching the same performance of the tabular for $\gamma = 0$ (short sighted agent) was not difficult, the problem arose when having to deal with $\gamma \neq 0$. One model that stood out was the *Best Mean Bi*. The features it was composed are listed below.

- ϕ_{BMB1} → **Correlation state-action**: self explanatory indicating the correlation between the current state and the one suggested;
- ϕ_{BMB2} → **Mean of correlation of [all states]-action**: this feature was designed aiming to capture the importance of an action in relation with all other states (as expected this feature had no impact for $\gamma = 0$ but revealed to be important for $\gamma = 0.9$;
- ϕ_{BMB3} → **State is cached**: Flag set to one if the current item is cached;
- ϕ_{BMB4} → **Action is cached**: Flag set to one if the suggested item is cached.

Figure 5 show the importance of each feature in a small scenario with $N_c = 50$, $N_r = 10$ and $N_c = 5$ after a training of 40000 episodes. As expected ϕ_{BMB2} had no impact when $\gamma = 0$ while showed to be relevant for $\gamma = 0.9$. On the contrary ϕ_{BMB3} had unstable behaviour and it was discarded. It is also important to notice how the $bias$ changes, getting higher values when the agent has the ability to see over the immediate step.

After having tested different few more experiments, ϕ_{BMB2} was transformed to better accommodate parallelization (addressed in Section 3.5.4) of the computations and as result the features effectively useful are summarised below.

- ϕ_1: Correlation between state s and suggested item by action a ($U[s,a]$).
- ϕ_2: Flag set to 1 if the action a is cached.



(a) $\gamma = 0$ (b) $\gamma = 0.9$

Figure 5: Feature weights *Best Mean Bi*

- ϕ_3: Number of actions in the N_r most correlated with $s_{t+1} = a_t$ and also cached, normalized by the number of cached states. This feature showed to be relevant for $\gamma \neq 0$ giving the agent a vision over the next immediate state.

To get an idea of how much each feature contributes to the approximation, the features are grouped in two different representation: **Corr-Cache** (ϕ_1, ϕ_2) and **Miriam** (ϕ_1, ϕ_2, ϕ_3).

Now for each pair state-action the result will a scalar number the new function approximating the q-value is as follows in Equation 9.

$$\begin{aligned} f_q(\phi(s,a)) &= w\phi(s,a)^T + b \\ w &\in \mathbb{R}^{|\phi(s,a)|},\ b \in \mathbb{R} \end{aligned} \tag{9}$$

The goal is now to find w and b, in total $|\phi(s,a)| + 1$ parameters to update, and in this case the method used during training is as explained in the next section, Gradient Descent.

3.5.2 Gradient Descent training

Given the simplicity of the current representations, the memory replay could result in a overkill or even worsen the situation by adding complexity were not needed. Therefore the model has been trained performing a gradient descent on the q-value after each action during simulation. The Algorithm 4 shows in detail how weights are updated. Thanks to the use of the $PyTorch$ library was not need to implement the Gradient Descent update step by hand but it was already given by the library.

Algorithm 4: Q - Gradient descent training

initialize $model$ with random weights;
for $episode \leftarrow 0$ **to** N **do**
  reset s_0;
  $done = false$;
  while *not* $done$ **do**
    With probability ϵ select random action a;
    otherwise select $a_t = \arg\max_a f_q(\phi(s_t, a))$;
    Take action a_t, observe $(r_{t+1}, s_{t+1}, done)$;
    $y = r_{t+1} + \gamma \max_a f_q(\phi(s_{t+1}, a'))$;
    Gradient descent on $(y - f_q(\phi(s_t, a_t)))^2$

3.5.3 Python Implementation

Initially the algorithm was implemented through the $Numpy$ and $PyTorch$ libraries. In fact ϕ was implemented through a class that will compute on the fly the value a certain pair. This approach created an excessive overhead, and slowed the training phase considerately. Therefore to allow running multiple and in depth test it was decided to pre-compute all the values of ϕ once, and store them in memory: this brought an overwhelming improvement in the training time at the cost of storing in memory a matrix $\Phi \in \mathbb{R}^{N_s^2 \times |\phi(s,a)|}$. With the resources provided by Colab it was possible to simulate catalogues of more than 10000 items. Despite such improvements one

training would take around 7/8 hours for an environment with $N_c = 5000$.
To tackle this problem and be able to reach more acceptable training times and do more simulations two further optimizations of the code were done and the the total training time was reduced significantly.
First of all, the data structures implemented with $Numpy$ got substituted with equivalents, wherever possible, of the $Tensor$ module of the $PyTorch$ library: this allowed to transfer all the computations from CPU to the GPU, taking full advantage of the CUDA Cores offered by Colab. This significantly reduce the timings in training process, bringing them down to about 20 minutes for thousands of episodes, independently of how big the catalogue was.
The main problem remaining to be solved is to address the actual computation of ϕ. This is done by parallelizing the process. More in detail, instead of calculating $\phi(s,a)$ the function was implemented such that all the action are treated together and it is computing $\phi(s) \in \mathbb{R}^{N_s \times |\phi(s,a)|}$. Such resulting matrix have as i^{th} row $\phi(s,a_i)$. The main challenge then become finding for each feature of a way of computing them in one step.

3.5.4 Parallelization

The U matrix, also referred as correlation matrix is made elements $U[i,j]$ that indicates how much the i^{th} item is related to the j^{th} one. To obtain $\phi_1(s)$ for a whole state was enough to take the row corresponding to such state. Considering that ϕ_2 depends only on the action, and from each state it is possible to take any action for each state, $\phi_2(s)$ is a vector of zeros and ones: with one if the action suggested points to a cached item. This allowed to compute the such vector only once and reuse it for all the states. The most challenging part was vectorizing $\phi_3(s)$ since relied more on the action. First, for each state the are computed the N_r most correlated items, to select then the action both correlated and cached was enough to take the vector containing the items cashed ($\phi_2(s)$) and intersect it with the matrix of most correlated for each state. This was possible thanks to the fact that the number of actions corresponds also to the number of states. Through $PyTorch$ indexing was possible to compute such value. Lastly the vector was normalized and eventuality obtained $\phi_3(s)$.

At the end of all of these optimizations was possible to obtain feature extraction times in the order of few seconds, allowing to train and obtain satisfactory results after training in an Environment with $N_c = 10000$ for 5000 episodes in about 14 minutes, while the tabular agent would take more then an hour to train for that many episodes, without even reaching the similar results.
In the following section the achieved results are analyzed more in detail.

3.6 Results

The first experiments were performed for smaller environments with $N_c = 20$ (20 items) with 4 cached and with $N_r = 4$. After training the two representations and confronted it with tabular method (all trained for the same number of episodes, 10000 in this case) the Figure 6 shows how the benchmark values stack against each other. Since the environment is really small, the difference is not big, so the approximation methods perform both pretty well with the **Miriam** model surpassing the **Corr-Cache** one, but without reaching the tabular agent.

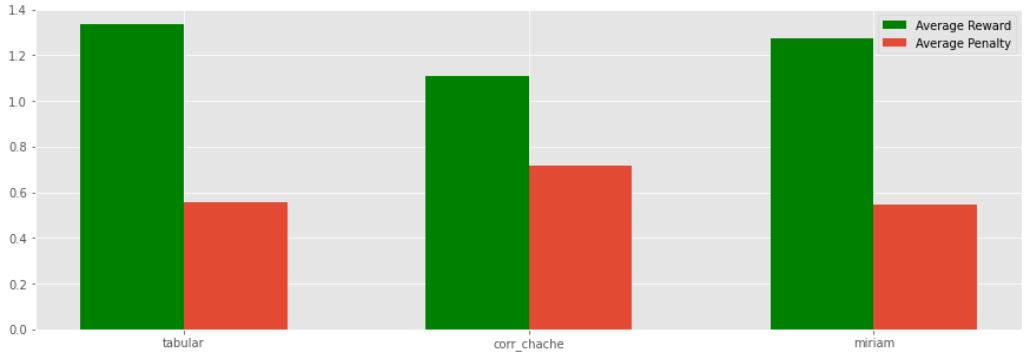


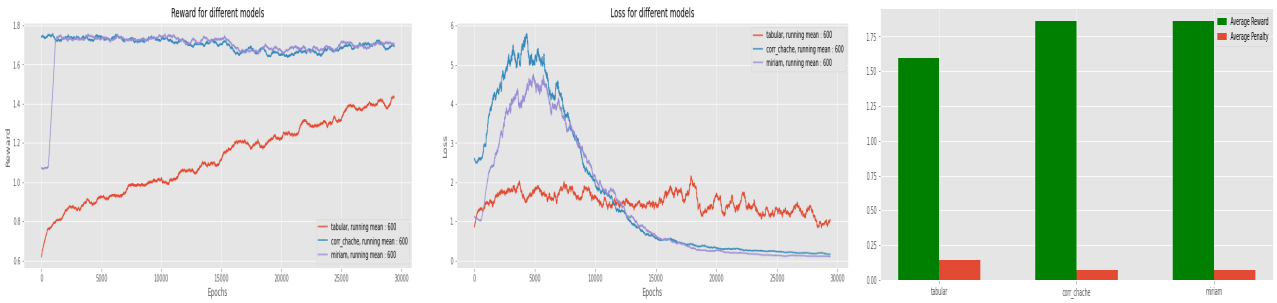
Figure 6: Reward higher the better, Penalty the lower the better

3.7 Increasing the problem size

ϵ	Reward type	R_{cache}	R_{corr}	User type	γ
0.1	variable	1	1	Random	0.9

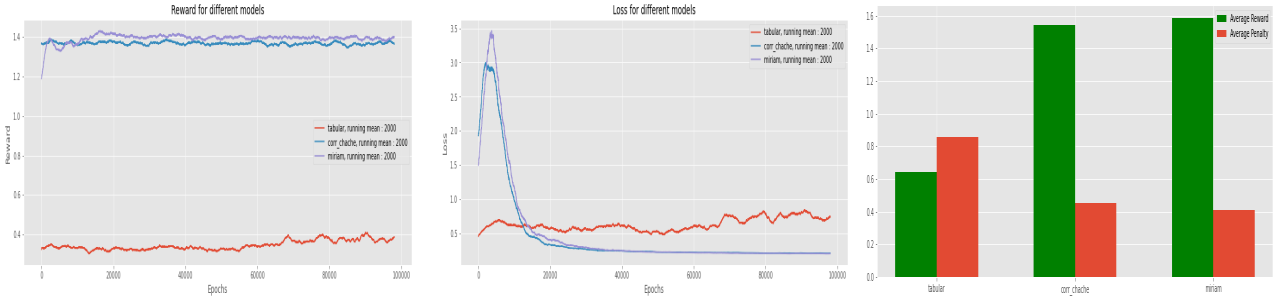
The models developed have no problem in approximating a short-sighed agent ($\gamma = 0$), but the problem arise when the number of states increased to the point that the tabular method would require too much time in order to give proper suggestion. The next step in fact was to limit test such function approximations. In the flowing Figures the **Tabular** agent is also kept as reference.

The results in Figures 7 and 8 demonstrate how the approximations are able to scale better over the catalogue size and reach results similar to the tabular with far less episodes. In particular in Figures 7a and 8a the average reward during training is plotted.Thre Red line is the Tabular version while the blue is the Corr-Cache and Miriam is violet (same coloring schema is preserved through all the graphs). The environment with 100 items was trained for 30000 episodes and it is possible to see that the approximating agents reach an optimal neighbourhood way faster than the tabular algorithm. In Figure 8a for 1000 items, 20000 episodes are not enough to see a noticeable improvement in the tabular model, that does not imply that the model is not learning but it is doing it way slower than the two approximations. On Figures 7 and 8 are displayed the benchmarks in the following order: **Tabular**, **Corr-Cache** and **Miriam**. As predicted for 1000 states the Tabular does not reach performances similar to the other two agents, keeping the same number of training episodes (Figure 8).



(a) Rewards (b) Losses (c) Scores

Figure 7: $N_s : 100,\ N_c : 10,\ N_r : 10$



(a) Rewards (b) Losses (c) Scores

Figure 8: $N_s : 1000,\ N_c : 33,\ N_r : 33$

3.7.1 Custom Correlation Matrix

N_s	N_r	N_c	Train episodes	ϵ	Reward type	R_{cache}	R_{corr}	User type
10	2	1	20000	0.3	variable	1	1	Random

Despite both representation performing fairly well with the standard environment, the feature ϕ_3 is able to improve the results in more complex scenarios like the one shown in Figure 9. Given the size of this test, the scores are compared with the optimal agent created by solving the MDP. As can be seen in figure 10, $Miriam$ achieved a better penalty score, despite having the same average reward as $Corr-Cache$. This disparity is coming from the policy of these models (Table 2), where $Miriam$ is able to capture the goodness of the state 4, the only one that can reach the state 0 and obtain a reward of 1.6, thus with no penalty.

Model/State	0	1	2	3	4	5	6	7	8	9
Optimal	2	4	5	1	0	1	8	4	4	0
Tabular	2	4	5	0	0	1	8	4	4	0
Corr-Cache	2	3	5	0	0	0	8	0	3	0
Miriam	2	4	4	1	0	1	8	4	4	2

Table 2: Models' policies

	0	1	2	3	4	5	6	7	8	9
0	0	0.8	0.9	0.7	0.6	0.6	0.5	0.4	0.3	0.2
1	0.1	0	0	0.8	0.7	0.5	0.2	0.2	0.3	0.4
2	0	0.1	0	0.9	0.8	1	0.9	0.3	0.5	0.2
3	0.5	0.9	0.8	0	0.2	0.2	0.2	0.2	0.2	0.2
4	0.8	0.9	0.7	0.2	0	0.2	0.2	0.2	0.2	0.2
5	0.4	0.9	0.8	0.2	0.2	0	0.2	0.2	0.2	0.2
6	0.3	0.2	0.2	0.2	0.2	0.2	0	0.2	0.9	0.8
7	0.2	0.2	0.2	0.2	0.7	0.6	0.1	0	0.1	0.2
8	0.1	0.1	0	0.8	0.7	0.2	0.2	0.2	0	0.2
9	0	0.3	0.4	0.2	0.1	0.2	0.1	0.2	0.2	0

Figure 9: green: cached, blue: correlated, yellow: cached and correlated

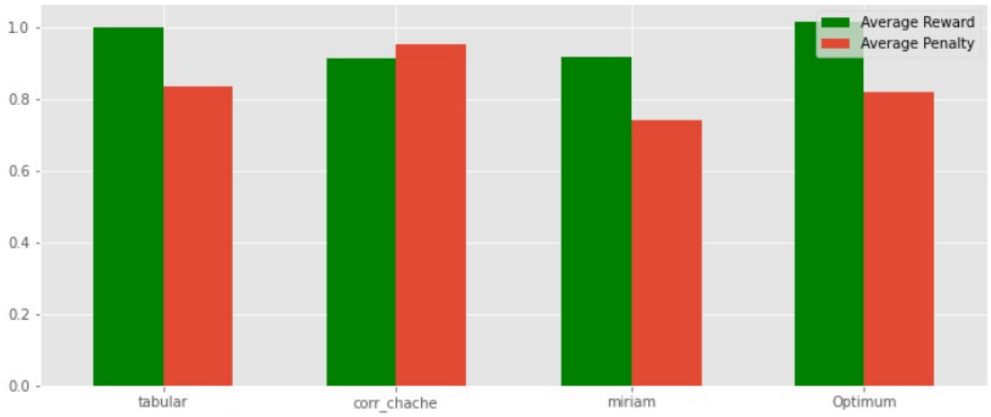


Figure 10: Results on custom matrix

3.8 Future work

The next step of this part of the project would be to implement more complex agents going from a simple Linear Approximation to some Deep Approximation with neural networks. In that case it could be useful to maybe implement the learning stage with the Experience Replay that in the linear case did not achieve good results.
All the tests run were done with a Random Environment where the user would pick a suggested content with a certain probability. It would be interesting to make further experiments with different kinds of users and see how the proposed approximations behaves.
A further improvement in training the model could using Double-q-learning instead of simple q-learning. This consist in maintaining two f_q and updating each one from the other next state. This approach in theory converges faster and further improves the training times.

4. Multiple Recommendations

The analysis that has been done so far aimed to find the best policy for recommending a single element of the catalog, and so, the size of the set of states that are possible in the MDP and the total number of actions that the algorithm needs to search for each state is the same. In other words, we were constantly dealing with a square matrix of the form $K \times K$ until now. In this section, we extend this work to the problem of multiple recommendations. This area is interesting because when we look at the problem of multiple recommendations, the search space for actions for each state is not limited to K actions, but now is a space of all possible combinations of N recommendations. Thus, as this number grows, the search-space grows considerably with each added recommendation and so, a better way to deal with this issue would be highly beneficial.

In this work, we tackle the problem of multiple recommendations by introducing a method that can split the problem of searching through N combinations of action-recommendations for each state into a problem of N simultaneous updates on a $K \times K$ Q-table. Section 4.1 explains the theory behind this algorithm and section 4.2 explains the implementation. We then show how this algorithm performs against the vanilla Q-Learning for multiple combinations of actions in different cases.

4.1 Theory

To understand the issue of tractability, let us look at the Vanilla Q-learning algorithm described in section 2.3 (that we will call from now on **Oracle**). We can see how the number of possible actions for each state A grows drastically in an N-batch recommender. Since we have to take into account all the possible combination of suggestions in the N size batch, given a state-space K, our action size $dim(A)$ becomes:

$$dim(A) = \binom{K}{N} = \frac{K!}{N!(K-N)!} \tag{10}$$

This increases the computations that need to be done, and the memory required to perform these computations, which in turn, increases the convergence time for the algorithm. Since we have to map each possible batch in the Q-table to a set of actions and Q-value updates, a novel algorithm was introduced that could work with any value of N, while keeping the size of the Action space limited to that of the single recommendation K. This algorithm (that we will refer to as **ALGO-QF**) has a Q-table size independent from the batch size N, and achieves this by performing parallel updates of the batch items by setting N Q-Values in each iteration.

To explain the working of ALGO-QF, we need to first define the notion of Frequency of Q-values. The frequency calculation is based on the realization that as the algorithm converges, the distribution over the Q-values should peak at the values that occur more frequently. Thus, inspired from the Boltzmann distribution [3] we define the probability of a state-action pair as being proportional to the Q value of this pair as shown in equation 11.

$$\mathbb{P}(S,u) \propto \exp(\frac{Q(S,u)}{kT}) \tag{11}$$

This is normalized for all the state-action pairs to get the frequency $F(S,u)$ as a soft-max function depending on an action item u given a state S and parameterized by the temperature T and $k = 1$, as shown in equation 12.

$$F(s,u) = \frac{\exp(\frac{Q(s,u)}{T})}{\sum_{j=1}^{K} \exp(\frac{Q(s,j)}{T})} \in [0,1] \tag{12}$$

The frequencies should, thus, satisfy equation 13.

$$\sum_{u=1}^{K} F(s,u) = 1 \tag{13}$$

The temperature T is a hyper-parameter that should start from a very high value and cool-down to a value very close to 0 as the Q-learning iterations evolve. This notion of frequency is then used to define a Q-learning target as shown in equation 14.

$$H(Q(S')) = \sum_{j=1}^{K} F(S',j)Q(S',j) \tag{14}$$

We implement this practically by keeping a track of the frequencies corresponding to each (S,u) tuple in the Q-table through an F-table of size $K \times K$. Thus, whenever any update to the frequency needs to be made, we update the F-table and use this compute H(), which is then used in Q-Learning as shown in equation 15, where α_t is the learning rate, γ is the discount factor and $R(S')$ is the reward that the agent receives when transitioning from state S to state S'.

$$Q(S,u) \leftarrow Q(S,u) + \alpha_t[R(S') + \gamma.H(Q(S')) - Q(S,u)] \tag{15}$$

It is important to tune the temperature in such way, that as the system cools down $(T \rightarrow 0)$, we expect:

$$H(Q(S')) = \sum_{j=1}^{K} F(S',j)Q(S',j) \rightarrow \frac{1}{N}\sum_{n=1}^{N} \max_{u} Q(S',u) \tag{16}$$

The full algorithm has been detailed in Algorithm 5. As we can see from the algorithm, the Q-table keeps the $K \times K$ size and the update is done in parallel for each item u in the batch suggested.

4.2 Implementation

4.2.1 Oracle

The Oracle was implemented using the algorithm 1, but with the additional extension to N recommendation. Thus, for each state, the algorithm searches a $K \times dim(A)$ action-space, where $dim(A)$ is defined based on N recommendation as shown in Equation 10. Out of all these combinations of actions, the algorithm searches for the combination that gives the highest reward on state transition. The corresponding Q-value is updated and this is repeated till either a maximum number of iterations is reached, or a certain difference is obtained.

Algorithm 5: ALGO-QF maximization

Initialize $Q(S,u) = 0$, for all $S \in \mathscr{S}$, $u \in \mathscr{A}(s)$, so that all items are equi-probable at the start.
Set starting temperature T_0 to a very large value.
Initialize F-table: $(F(S,u) = \frac{1}{K} \forall (S,u))$.
Set $t = 0$
for *each episode* **do**
  Set $l \leftarrow 0$ and $t \leftarrow t+1$
  Initialize S_0
  for *each step in episode* **do**
    Choose the actions a from S using ϵ-greedy approach a N-sized batch by choosing N items i with highest $Q(s,i)$ for state s
    Set $l \leftarrow l+1$, $t \leftarrow t+1$
    Observe S'
    Compute $R(S')$
    for *each item u in a* **do**
      Update Q:

$$Q(S,u) = Q(S,u) \leftarrow Q(S,u) + \alpha_t[R(S') + \gamma.H(Q(S')) - Q(S,u)] \tag{17}$$

      where

$$H(Q(s',:)) = \sum_{j=1}^{K} F(S',j)Q(S',j) \tag{18}$$

    Update F-table
    Reduce temperature T:

$$T = \frac{T_0}{t^{\delta}} \tag{19}$$

    set $S \leftarrow S'$

4.2.2 ALGO-QF

The ALGO-QF uses Algorithm 5 to make the Q-updates on a $K \times K$ table with the Frequencies being tracked in another F-Table as mentioned previously. We defined 3 variants on ALGO-QF to perform an ablation study of the features that make-up this algorithm, using equation 16 as a reference.

ALGO-QF with temperature decay : This is the algorithm proposed in 5, where the formula for Q update uses H(Q(S') with a starting temperature T_0 that decreases to a value very close to 0 towards the end. We tested this for multiple values of starting temperature T_0 and decay factor δ.

ALGO-QF with max: Exploiting the equation 16, we defined a variant of ALGO-QF that uses the final value to which it should converge by removing the need for the frequency update and directly using this value to get the Q-update as:

$$Q(S,u) = Q(S,u) \leftarrow Q(S,u) + \alpha_t\Big[R(S') + \gamma.\frac{1}{N}\sum_{n=1}^{N}\max_{u} Q(S',u) - Q(S,u)\Big]$$

ALGO-QF with fixed temperature: Exploiting the equation 16, we defined a variant of ALGO-QF that uses the final value to which it should converge by setting a very low $T_0 = 10^{-6}$ to begin with. This was used to understand the impact that a cool-down of temperature has on the convergence of the algorithm and the general performance on reward accumulation and catalog recommendations.

4.2.3 Common settings

To compare the results, the different implementations were tested on the same environment, with a data-set of 50 items, with 5 items cached and 7 correlated items for each one. We decided to run all the experiments for different values of γ and different exploration settings, so all the algorithms were tested with a fixed ϵ value and with a decaying ϵ strategy. This strategy updated the ϵ value according to equation 20, where the epsilon is decayed from episode D_{min} to episode D_{max}. The starting value of ϵ is 1, while the ending value was 10^{-6}.

$$\epsilon = \epsilon - \frac{\epsilon}{D_{max} - D_{min}} \tag{20}$$

We also checked the consistency of the results by verifying the performance for each parameter on multiple runs, and through averaging certain values.

4.3 State Transitions and Rewards

For each batch of actions A suggested to the user, S' represents the state in which the user ends-up based on the actions selected. Thus, according to equation 17, the state S' depends totally on user choices. The algorithm takes this into account by using the rewards generated from this state transition while updating the Q-values.

The reward that is given to the RL agent is conditioned on this new state in the following ways:

- If the state is neither cached, nor correlated, then the agent receives no reward i.e $R(S') = 0$
- If the new state is either cached or correlated, then $R(S') = 1$
- If the new state is both cached and correlated, then the agent receives the highest reward $R(S') = 2$

Note: It is important to highlight that the previous versions and tests were erroneously made since even though the equation conditioned the reward on the state transitions, the code of the previous version had a mechanism in which the rewards were conditioned on the actions that were proposed by the system instead of the state in which the user ends after making a decision on whether they want to accept the proposal or go differently. Thus, the same scheme of rewards was applied for checking if the action was cached, correlated, or both. This was leading the algorithm to learn the database instead of the user choice since the actions were always being presented by the agent.

4.4 Tunable Parameters

Learning rate α_t: Learning rate that the system uses to make a Q-update.

Discount Factor γ: The discount factor used to adjust the long term reward considered by the agent. Different runs were made, focusing on a γ value of 0.9 and 0.99.

Rate of Exploration ϵ: The rate at which the system explores new states while exploiting an already learned policy. We used a decay strategy according to equation 20 and controlled the decay by adjusting the values of (D_{min}, D_{max}) for our tests. The standard values were (0, max_iter) where max_iter is the upper limit on the number of episodes. However, to check how fast our algorithm was able to learn high rewards, we used custom values for D_{max}, like 500, 1000, 50000 episodes, etc. For the tests in which ϵ was fixed, we kept it at 0.1, which corresponded to exploring roughly 10% of the time.

Initial Temperature T_0: Starting Temperature value. Different runs were made changing this value, we decided to set it to 100 in the normal ALGO-QF and to $1e-6$ in the ALGO-QF with fixed temperature.

Temperature decay parameter δ: Parameter tunable to set the temperature decay, following the equation 19. Tests were made with different values, such as 0.1, 0.5, and 0.9.

[User]α: Probability of the user to choose randomly among the recommended items. Several runs were made with different α values, such as 0.6, 0.8, and 1.

Environment settings: In all the tests done, we considered a dataset of 50 items, where 5 items where cached and each item had 7 correlated items.

4.5 Tests on temperature value and temperature decay

To understand the effect of temperature on our algorithm, we tried out multiple tests with different temperature settings. the main idea was to have an ablation study of the different parameters that impact the temperature and how does that impact the reward accumulation process. All these tests were conducted initially for the case of 2 recommendations and re-evaluated for the case of 3 recommendations, with similar results being seen.

Tests for different values of δ: The first parameter we tested out was the exponent by which the episodes impacted the temperature i.e δ. We chose three values of δ for this: 0.9, 0.5, 0.1. Figure 11 shows this comparison for the case of 2 recommendations, with the initial temperature set to 100.

This showed us that setting δ to 0.1 gave us the best accumulation of rewards, and as we increased δ this reward decreased for the same temperature setting. We observed this same result for other temperature settings too. Thus, we decided to keep the value of δ to 0.1.

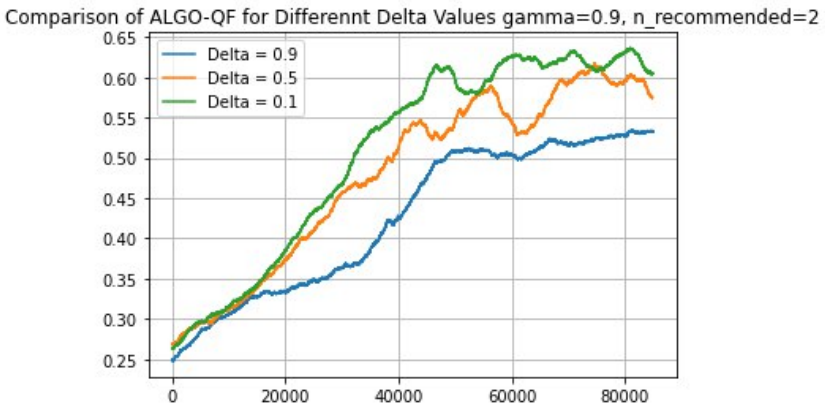


Figure 11: Test results for temperature settings with 2-item batch size for different values of δ

Tests for different values of T_0: After understanding the best value of δ, we went ahead and tested the effect of changing initial temperature with the same temperature decay profile. Figure 12 shows this comparison for initial temperatures of 10, 100 and 1000. We see that the accumulation profile does not vary significantly with different initial temperatures, but even in this slight variation, we saw that at a temperature value of 100, the reward turns out to be the best on average.

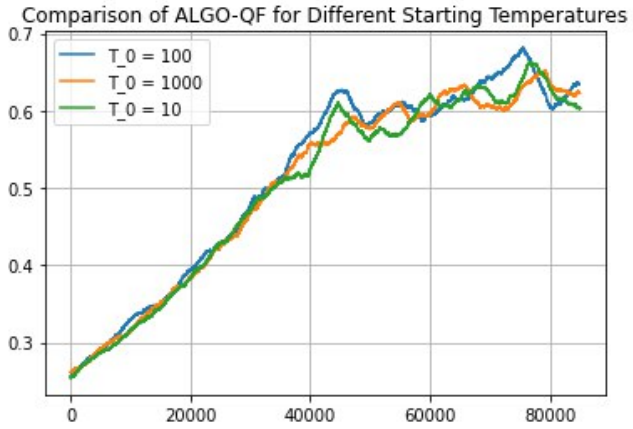


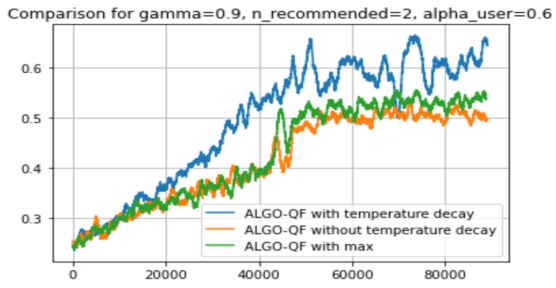
Figure 12: Test results for different initial temperature settings with 2-item batch size for same value of δ

The interesting thing we understood through this analysis is that the temperature decay profile, which is controlled through δ is what determines the final performance of ALGO-QF, instead of the initial and the final values of temperature. As seem in figure, 12, the variation between different starting values for the same δ are minimal, but the performance changes a lot between different delta values. Thus, we decided to stick to an initial temperature of 100 for the δ value of 0.1 for our tests.

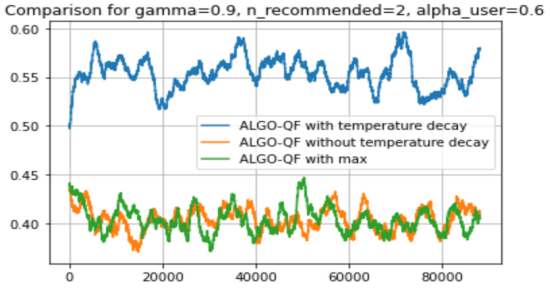
4.6 Tests on the different ALGO-QF implementations

In the first tests done, we did not notice the issue with the rewards being based on action values, as highlighted in section 4.3. So, we introduced the two ALGO-QF implementations, the one with fixed temperature and the one with the max. After fixing the update part of the algorithm by changing the rewards to be conditioned on states, however, we compared again their behavior, to see if the δ value found in section 4.5 was able to outperform also those implementations:

4.6.1 2-items batch recommendations



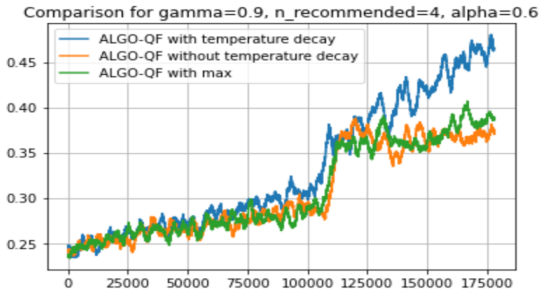
(a) Reward with ϵ decay



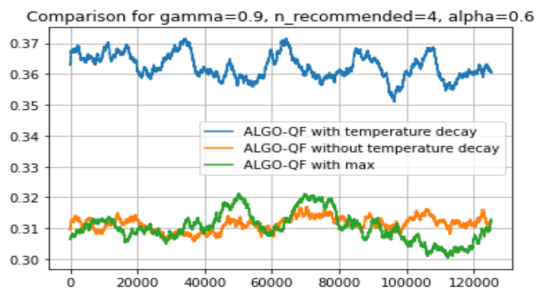
(b) Reward with ϵ fixed to 0.1

Figure 13: Test results with 2-item batch size

4.6.2 4-items batch recommendations



(a) Reward with ϵ decay



(b) Reward with ϵ fixed to 0.1

Figure 14: Test results with 4-item batch size

As we can see from the results, the ALGO-QF with a fine-tuned temperature decay behaves better than the other implementations for both fixed and decaying epsilon profiles, so we decided to stick with this agent for further testing. We postulate that the reason for this could be that ALGO-QF can gradually learn a better policy due to the gradual temperature decay with a δ value of 0.1 and initial temperature set to 100, as explained in 4.5, when pitted against the other variants. Hence, we decided to use this agent for further tests.

Another fact to note is that, while in the 2-items batch analysis ALGO-QF can have similar performances of an "only related" agent benchmark, increasing the batch size led to a general worsening of performances, which we estimate is either due to the estimation properties of ALGO-QF or due to further tuning required for each batch of recommendations.

Tests with a 4-item batch recommendation, however, were possible to be done only with this implementation, because the Oracle was too slow and time/memory-consuming to be able to make runs.

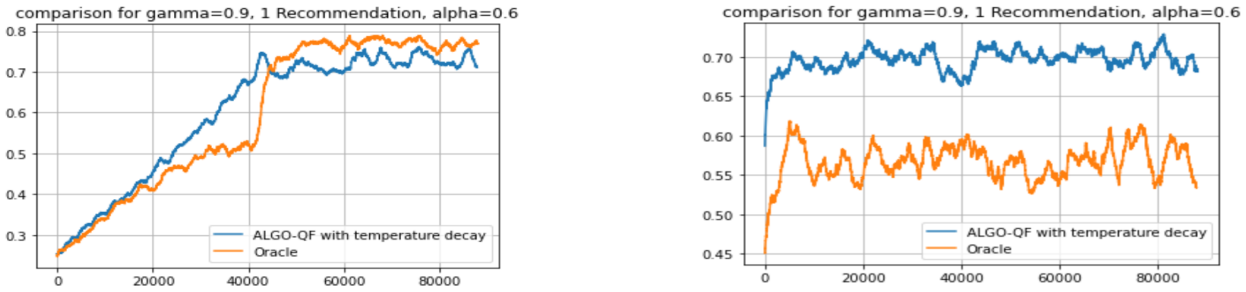
4.7 Tests on different α values

From the results obtained in section 4.6, we will now focus only on the Oracle and the ALGO-QF implementation with temperature decay and $\delta = 0.1$.

We decided to run our algorithms with the random user with different α values, to see how this randomness affect our agent's results. Following this, we compared the average result achieved in each episode, to understand the effectiveness of the agent's suggestions.

4.7.1 Single recommendation

We first focused on $N = 1$, comparing the Oracle with the ALGO-QF algorithm, with an α probability of 0.6. Figure 15 shows this for fixed and decaying epsilon strategies.



(a) Reward with ϵ decay

(b) Reward with ϵ fixed to 0.1

Figure 15: Test results with $\alpha = 0.6$

As we can see from the reward comparison, in the ϵ-decay implementation both algorithms seem to converge to a policy that enables them to gain similar rewards. This is important to highlight since the average accumulated reward is higher in the ϵ-decay case than the average reward in the fixed-ϵ case. This is because, in the latter, there is a constant 10% exploration which doesn't allow the agent to explore as much at the start and forces it to explore even after learning a decent policy. It seems, however, that the ALGO-QF can converge faster to a better policy than the Oracle in this time frame analysed.
When we compare the results achieved for a known benchmark, such as a random recommender or an "always-correlated" recommender, we can see that ALGO-QF is indeed able to outperform. As the random recommender will have a very bad performance, the "always-correlated" recommender will have a fixed +1 reward whenever the user accepts a suggestion, leading to an average reward of around 0.6, which ALGO-QF can beat.

Figure 16 show the classes that are recommended by the 2 algorithms. From this image, we can see how it seems that despite the good performance, the database is not learned in the best way, as in some cases both algorithms suggest some elements that are neither cached nor correlated. This could be due to the effect of having a lower value of α that introduces some randomness in the learning process.

It could also be that some "bad" recommendations are done to get a bigger future reward due to $\gamma = 0.9$. However, given the data-set size (50 items) we would exclude that possibility. When we remove the randomness of the user by setting $\alpha = 1$ (The user always accepts the suggestion) we see that ALGO-QF recommends content that is either correlated or cached and correlated as compared to the Oracle, as shown in figure 17. In this case, we can highlight that all recommendations

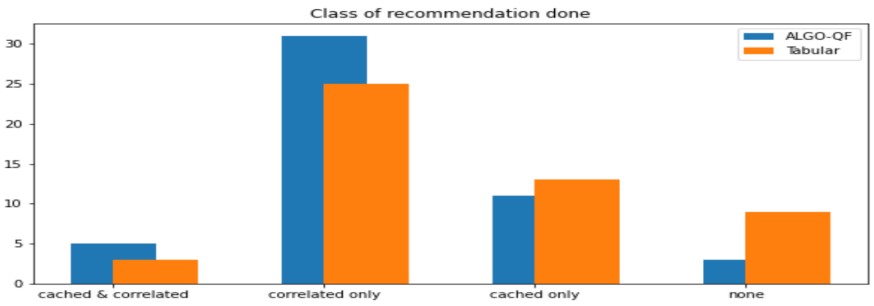


Figure 16: Class of Item suggested

are meaningful, with the difference in suggested item type that can be associated with the same reward earned if the item is cached or if it is correlated.

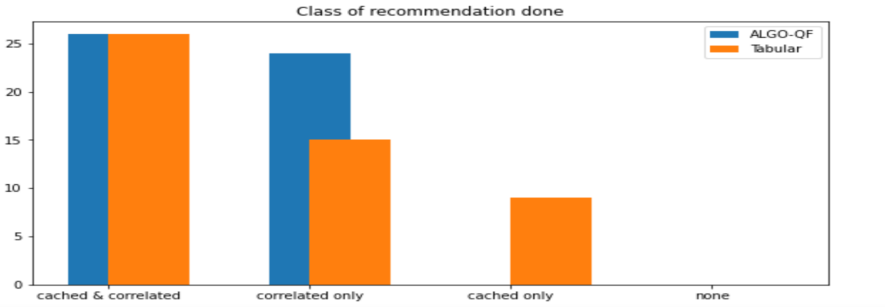
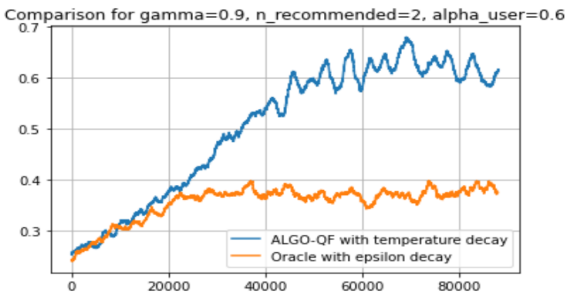


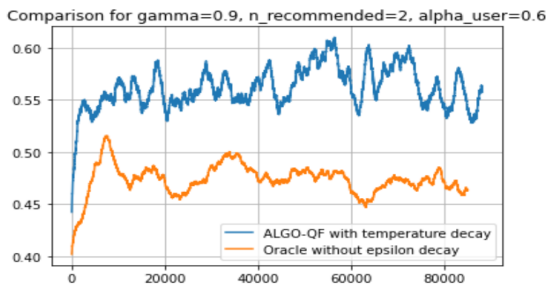
Figure 17: Class of item suggested

4.7.2 2-recommendations batch

Once we observed a similar behavior of the two algorithms in the single recommendation case, we moved to a more interesting analysis on the different behaviors in a 2-items batch recommendation, testing the ALGO-QF with different values of alpha, ranging from 0.6 to 1. Figure 18 shows this comparison for fixed and decaying epsilon strategies. As we can see, with $\alpha = 0.6$ in both cases the ALGO-QF outperforms the Oracle, that in the ϵ-decay case seems to have not yet reached an optimal policy. In order to solve this issue, new runs with a slower exploration decay may be useful, but will lead to a bigger amount of episodes needed. The behavior in the fixed-ϵ test, instead, seem to confirm what observed in the single recommendation case.



(a) Reward with ϵ decay

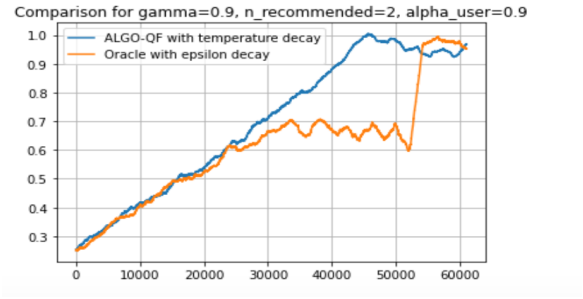


(b) Reward with ϵ fixed to 0.1

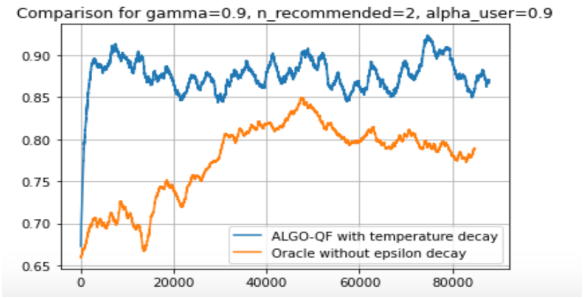
Figure 18: 2-items recommendations with $\alpha = 0.6$

When we ran further tests, as shown in figure 19, we found out that with higher α - and, thus, less

randomization - the Oracle performs better, getting same performance in the ϵ-decay implementation:



(a) Reward with ϵ decay, high α value



(b) Reward with fixed ϵ , high α value

Figure 19: Tests with high α value

We postulate, based on the results above, that the Oracle struggles to fully understand the dataset composition when the user introduces too much randomness, while the ALGO-QF can converge to a better policy. This can be attributed to either the problem having a mode that is simple to understand and, thus, does not require a lot of exploration, or the ALGO-QF being inherently superior in terms of its exploration capabilities. Considering the simplification introduced by the $K \times K$ Q-table, which allows a faster convergence, we did not conclude on either possibility without further tests. It is important to notice that while the Oracle performed well when there is not much randomness in the user choices, the ALGO-QF implementation seemed to behave well in all the tests done, outperforming or getting close to the performance of an "only correlated item" recommender, which we used as a benchmark.

4.8 Tests to highlight different convergence speed

In the case of recommendations on a 1-item batch, the two algorithm have same convergence speed in the ϵ-decay setting, since the two Q-tables have same size. However, for a 2-items batch of recommendations we already have a big difference in Q-table size, as highlighted in 4.1. We, thus, decided to test the difference between the rate of convergence by changing multiple parameters. Following were the setups we tested:

- Increasing the complexity of the problem
- Increasing γ
- Limiting the ϵ-decay exploration by setting manual thresholds

4.8.1 Tests with fixed ϵ and high γ

In order to check the faster convergence tha ALGO-QF was supposedly displaying, we ran the tests with a γ value of 0.99 and a fixed ϵ strategy. Figure 20 shows these results. As we can clearly see, the oracle is still learning even after 100,000 episodes, while ALGO-QF seems to have achieved convergence after only a few episodes.

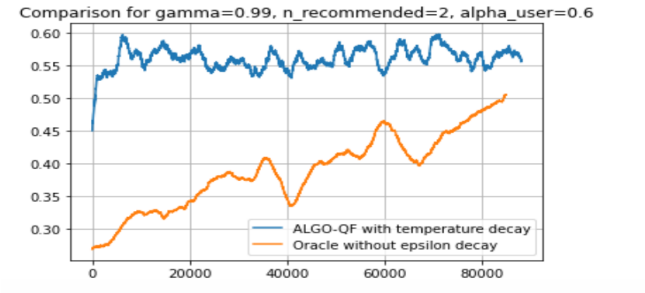


Figure 20: Reward with fixed ϵ value

4.8.2 Tests with ϵ-decay and early exploration stoppage

To highlight this faster convergence, we ran the test pertaining to figure 18a again, but with a controlled exploration through an upper threshold. Figure 21 shows the results for these tests. the behavior displayed by ALGO-QF shows us that even with an exploration stoppage at 15000 episodes, ALGO-QF achieves a policy that is very similar to the one achieved with an exploration decay until 50,000 episodes, previously shown in 18a. This highlights that ALGO-QF can understand the user very fast, in the case of 2-recommendations.

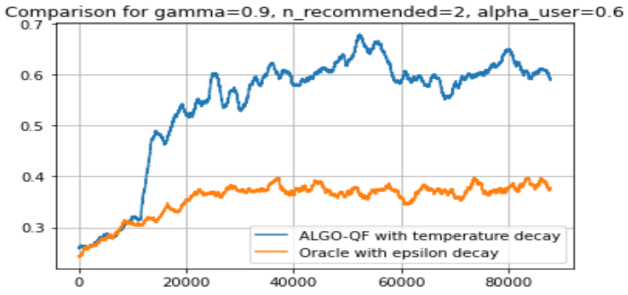


Figure 21: Reward with different ϵ-decay stopping criteria

4.9 Tests with different users

Our next step was to understand what ALGO-QF is learning in its exploration phase. We postulated that this might depend on the nature of 'relevant' information that the problem contains. To understand this, we will formalize the notion of an optimal final volume, taken from [2] as follows:

Definition 1. *Let M be an MDP with state-space $\mathbb{S}$. If the set $\mathbb{G} = \cap_{\mathbb{G}_a \in \mathbb{A}} \mathbb{G}_a$ is non-empty, we call $\mathbb{G}$ the **optimal final volume of M**. Here, $\mathbb{A}$ is the set of non-empty sets $\mathbb{G}_a \subseteq \mathbb{S}$ such that*

$$\exists t_0 > 0 : \quad P_{s_t \sim q_{\pi^*}}(s_t \in \mathbb{G}_a) = 1 \quad \forall t \geq t_0 \quad . \tag{21}$$

Now, if most of the information required for a relevant recommendation that leads to a higher reward is in this optimal final volume, then the ability of an agent to converge to high rewards depends on its inherent ability to discover this optimal volume of the MDP. We postulated that ALGO-QF is probably better at doing this than the Oracle for the given problem, and so, can converge faster. Thus, we decided to test this out further by segregating the data based on user-preferences and restricted availability.

To test how ALGO-QF performs on users that have preference for content-types, we decided to test it out on two kinds of users, i.e $n(\mathbb{G}) = 2$. Let's call these users Sporty - The user that prefers sport related content - and Artzy - The user that prefers art related content. To simulate this, we performed the following steps:

- **Case A:** We simulated a dataset containing two types of content by splitting it into two halves, where all the items of the same half were highly correlated with each other, and uncorrelated to items of the other half. The amount of total items that were correlated, however, was still fixed to the hyper-parameter set while initializing the environment, and similar to our previous tests, this value was 7. Thus, out of 7 items that were supposed to correlated in the whole dataset, an unbalanced half of these items would be correlated to each other and totally uncorrelated with the other part.
- **Case B:** We simulated a data-set with restricted availability, which could be due to some server issues. In this case the correlation matrix does not change with respect to the tests done in other sections i.e elements in the first half may be correlated to elements of the second half and vice versa. The user, however, is able to access to only a given half of the entire data-set.

This analysis is useful for not just understanding the ability of ALGO-QF to discover the final volume, but also to see how much the correlation factor has a role in the behaviour analysis and if our agents are able to understand user behavior even when it's purely arbitrary, as in case B. However, even in the case A: setting the number of correlated items as a fixed number will lead to have a 0 reward even if two items of same type are highly correlated, so further investigation on the reward received by the correlation factor still needs to be be done, exploring the possibility to introduce a minimum correlation value to obtain the full reward, or setting the reward as a function of the correlation.

An iteration on the above lines has been summarized in algorithm 6. We tested those two cases with N=1 and N=2 and our results have been summarized in the following sections. All runs were made with an ϵ-decay approach, with a fixed exploration stoppage after around 50000 episodes. It could be interesting to dig deeper into those settings, analysing the behavior recognition with an ϵ-fixed approach or changing the exploration threshold.

4.9.1 Case A

Figure 22 shows the results for the classes recommended for case A. As we can see from those class plots, the two algorithms have understood almost completely the user preferences. The high correlation between the items of same class can be one of the reasons of this great result.

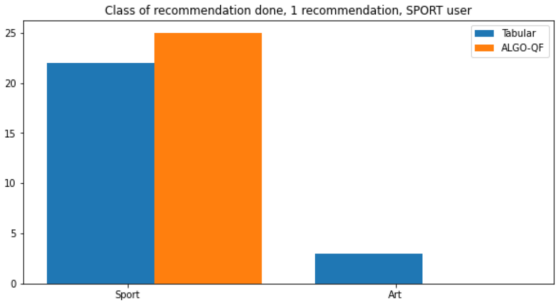
the results for the case of 2-recommendations are shown in figure 23. In this case, also, we have quite good results: given the user behavior it is enough to have one right item in the proposed batch, so on average all the implementations succeed in recommending right items. Moreover, we see that the recommendations on the other class by ALGO-QF that has a finely tuned temperature profile are less as compared to the others.

Algorithm 6: Sporty/Artzy user implementation

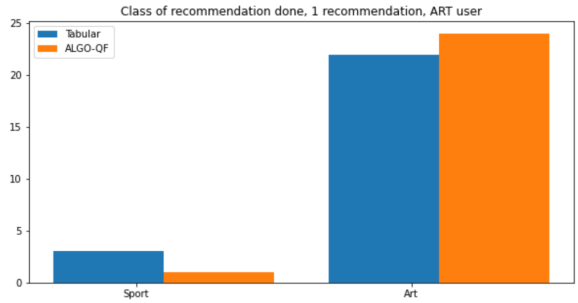
Environment for each user - sporty and artzy - receives indexes of the initial and final item of the class, in order to create a p0 uniform probability to pick only items of interest.
If no offset is set, then the initial item is item 0 and the final item is item $K-1$.
Case A only: receive and set inner correlation matrix u
sample u matrix for a 6-item dataset in case A:

$$u_{caseA} = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

for *each step* **do**
User leaves episode with probability *to_leave*
If Agent suggests only items of the wrong class
then User picks randomly an item of its class
else:
User will choose among the correct items suggested with a tunable probability α, else will pick randomly an item of its class.

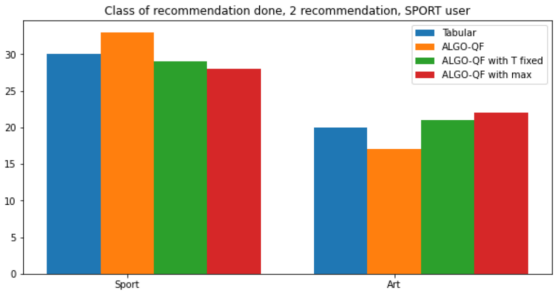


(a) Sport user recommendation when in Sport state

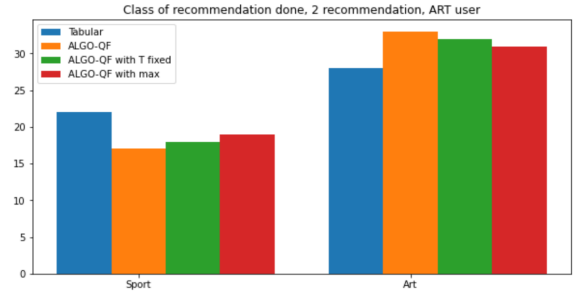


(b) Art user recommendation when in Art state

Figure 22: Item type suggestion when single recommendation done



(a) Sport user recommendation when in Sport state



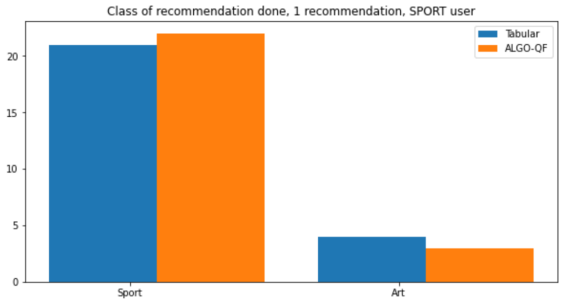
(b) Art user recommendation when in Art state

Figure 23: Item suggested in a 2-items batch recommendation

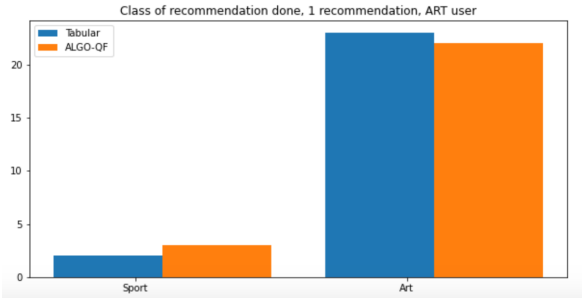
In order to have better results it could be investigated to set a minimum correlation value, in order to increase the number of possible items that give a positive reward, as well as adding a reward for

the user click, that could improve further the results.
Moreover, it can be noticed that ALGO-QF with temperature decay is the implementation that has better class recognition, beating the other two implementations. This further bolsters the analysis done in section 4.6. The oracle being beaten in this analysis could be due to proper convergence not reached (all algorithms ran with an ϵ-decay implementation), as it could be a possibility, as highlighted in previous tests.

4.9.2 Case B



(a) Sport user recommendation when in Sport state

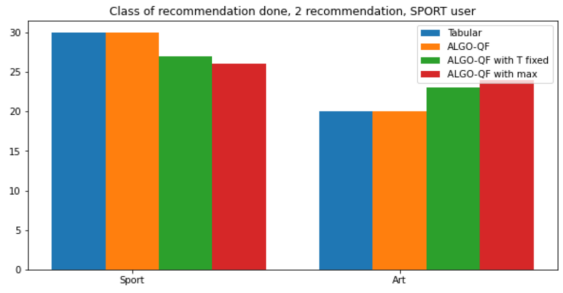


(b) Art user recommendation when in Art state

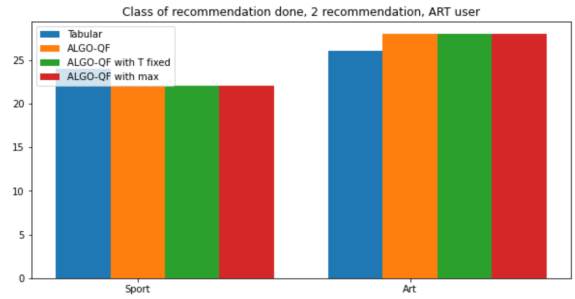
Figure 24: Item type suggestion when single recommendation done

In the case of partial-visibility of the dataset - Case B - we expected a slightly worse performance than in case A, which is what we observed as shown in figure 24. We attributed this to the less correlated situation we have in this case. Adding a reward for the user click could be a possible way to alleviate this problem.

Similar to the previous case, our results for the multiple recommendation case, shown in 25 are very close to the tests done in Case A. We can then be quite confident on the effectiveness of our algorithms, since they seem to be able to properly face both the situations.



(a) Sport user recommendation when in Sport state



(b) Art user recommendation when in Art state

Figure 25: Item suggested in a 2-items batch recommendation

As already noted, it could be interesting to modify the reward policy, in order to see if we can improve this behavior analysis further. While these results do indicate that ALGO-QF is able to understand a subset of data if it has items correlated to each other, the difference in recommendations is not very large as we had expected. Hence, we cannot confirm whether exploration of user-preferences on sparse episodes is, in fact, the strength of this algorithm, and would suggest further testing that should be done to prove/disprove this.

4.10 Results, comments and conclusions

The tests in the previous section have been very useful in discovering the potential of ALGO-QF and highlighting its limitations along with the problems of the Oracle implementation. It was highlighted that with a more random user, the Oracle struggled to achieve a satisfying policy in a satisfying number of episodes, while the ALGO-QF was still able to perform quite well.

Overall, it seems that the approximation of the max. introduced by the ALGO-QF that allows it to learn the same behaviour on a $K \times K$ matrix has a slight impact on the performance in a 2-items batch recommendation case, while with higher batches it starts to behave poorly. All the implementations, however, were able to recognize the user preferences in the experiments done in section 4.9, showing us that they are able to understand and adapt to a more specific user.

Further experiments and research needs to be done in parameter tuning as well as in the testing with other user types.

4.11 Future work

As highlighted in section 4.9.2, a new reward policy may be explored, in order to improve the item differentiation and have a better view on the real data-set configuration, but in general further experiments on different settings and user types will be fundamental to get a better view on the effectiveness of those algorithms. Currently ,the rewards are the Mae for items that are either correlated, or cached, and changing them to nudge the agent to learn more correlated or more cached items is a direction worth exploring. Moreover, shaping the rewards based on user-related metrics, in addition to the current set-up, is also a direction worth exploring. Modeling the correlation or cached reward as a floating value within a range, could help to reflect more properly the data-set settings, adding depth in the analysis that our Agents can do.

A new type of user can be analysed too, for example, one that accepts the suggestion only if a certain minimum correlation threshold is passed. Another possible way to proceed is to progressively increasing the data-set size, analysing how performance and results change and if the ALGO-QF implementation is able to scale well. In conclusion, we believe our work has taken a significant step towards understanding the space in which techniques that aim to improve search through decomposition, similar to [6], might make an impact.

5. Conclusion

To conclude the Semester Project it can be said that both approaches made significant steps towards the expected goals listed in Section 1.2, in the direction of exploring the improvements that can be made to a problem that scalably becomes intractable. The results are viewed in detail in their respective sections. Although these are promising steps forward, there is still a lot to explore in each of the two approaches and more work needs to be done on these to validate the ideas.

An interesting direction that can be explored is a way to merge the two works. This would allow evaluating the applicability of the approximation approach to a multiple-Recommendation case, trying to test this approach on more representatives data-set. If possible, a combination of these two approaches would be a very interesting, and potentially highly efficient, way to solve the search problem in higher dimensions that was explained in section 1..

Acknowledgement

We want to thank our supervisors Dr. Spyropoulos, Dr. Giovanidis and Dr. Giannakas, for guiding us through this project. The help and guidance that they provided during the project were conducive to understanding the background related to the work of this project. Their continuous and valuable feedback during our work helped us to better understand concepts and implementations, and allowed us to work more efficiently.

References

[1] A. Andrew. “Reinforcement Learning: An Introduction by Richard S. Sutton and Andrew G. Barto, Adaptive Computation and Machine Learning series, MIT Press (Bradford Book), Cambridge, Mass., 1998, xviii + 322 pp, ISBN 0-262-19398-1, (hardback, £31.95)”. In: *Robotica* 17 (1999), pp. 229–235.

[2] Anonymous. “Plan-Based Asymptotically Equivalent Reward Shaping”. In: *Submitted to International Conference on Learning Representations*. under double-blind review. 2021. URL: https://openreview.net/forum?id=w2Z2OwVNeK.

[3] Alexander Bach. “Boltzmann’s Probability Distribution of 1877”. In: *Archive for History of Exact Sciences* 41.1 (1990), pp. 1–40. ISSN: 00039519, 14320657. URL: http://www.jstor.org/stable/41133876.

[4] RICHARD BELLMAN. “A Markovian Decision Process”. In: *Journal of Mathematics and Mechanics* 6.5 (1957), pp. 679–684. ISSN: 00959057, 19435274. URL: http://www.jstor.org/stable/24900506.

[5] Theodoros Giannakas, Pavlos Sermpezis, and T. Spyropoulos. “Show me the Cache: Optimizing Cache-Friendly Recommendations for Sequential Content Access”. In: *2018 IEEE 19th International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)* (2018), pp. 14–22.

[6] Eugene Ie et al. “SlateQ: A Tractable Decomposition for Reinforcement Learning with Recommendation Sets”. In: *Proceedings of the Twenty-eighth International Joint Conference on Artificial Intelligence (IJCAI-19)*. See arXiv:1905.12767 for a related and expanded paper (with additional material and authors). Macau, China, 2019, pp. 2592–2599.

[7] M. Manhart. “Markov Processes”. In: 2012.

[8] Pravati Swain, Purandar Bhaduri, and Sukumar Nandi. “Probabilistic model checking of IEEE 802.11 IBSS power save mode”. In: *International Journal of Wireless and Mobile Computing* 7 (May 2014), pp. 465–474. DOI: 10.13140/2.1.1204.0640.

[9] M. Wooldridge and N. Jennings. “Intelligent agents: theory and practice”. In: *Knowl. Eng. Rev.* 10 (1995), pp. 115–152.