Performance Assessment of Context-aware Online Learning for Task Offloading in Vehicular Edge Computing Systems

Mutaz A. B. Al-Tarawneh[1]
Computer Engineering Department
Faculty of Engineering
Mutah Univesity, Jordan

Saif E. Alnawayseh[2]
Electrical Engineering Department
Faculty of Engineering
Mutah Univesity, Jordan

***Abstract*—Vehicular Edge Computing (VEC) systems have recently become an essential computing infrastructure to support a plethora of applications entailed by smart and connected vehicles. These systems integrate the computing resources of edge and cloud servers and utilize them to execute computational tasks offloaded from various vehicular applications. However, the highly fluctuating status of VEC resources besides the varying characteristics and requirements of different application types introduce extra challenges to task offloading. Hence, this paper presents, implements and evaluates various task offloading algorithms based on the Multi-Armed Bandit (MAB) theory for VEC systems with predefined application types. These algorithms seek to make use of available contextual information to better steer task offloading. These information include application type, application characteristics, network status and server utilization. The proposed algorithms are based on having either a single MAB learner with application-dependent reward assignment, multiple application-dependent MAB learners or dedicated contextual bandits implemented as an array of incremental learning models. They have been implemented and extensively evaluated using the EdgeCloudSim simulation tool. Their performance has been assessed based on task failure rate, service time and Quality of Experience (QoE) and compared to that of recently reported algorithms. Simulation results demonstrate that the proposed contextual bandit-based algorithm outperforms its counterparts in terms of failure rate and QoE while having comparable service time values. It has achieved up to 73.4% and 21.7% average improvements in failure rate and QoE, respectively, among all application types. In addition, it efficiently utilizes the available contextual information to make appropriate offloading decisions for tasks originating from different application types achieving more balanced utilization of the available VEC resources. Ultimately, employing incremental learning to implement the proposed contextual bandit algorithm has shown a profound potential to cope with dynamic changes of the simulated VEC systems.**

Keywords*—*Vehicular edge computing; task offloading; multi-armed bandits; contextual bandits

I. Introduction

Recently, the emergence of smart and connected vehicles has excelled the development of various types of vehicular applications such as infotainment and autonomous driving services [1], [2]. These applications are usually supported by the on-board computing and storage hardware resources. However, the ever-increasing spectrum of compute-intensive vehicular applications and services has rendered the on-board computational resources inadequate. Hence, Vehicular Edge Computing (VEC) systems have emerged as a baseline for providing high-performance and reliable computing services for in-vehicle applications [3]. In these systems, vehicles, edge servers - instantiated at the road side units (RSUs) and cloud servers can contribute their resources to process computational tasks generated from on-board mobile devices or vehicular driving systems [4]. Hence, computational tasks within VEC systems can be offloaded to any of the available hardware resources to ensure their correct and timely execution. While task offloading can enhance task execution and improve user-perceived Quality of Service (QoS), designing an efficient task offloading scheme is not straightforward. First, the VEC environment encompasses different application classes each with different processing demands, network bandwidth requirements, timing constraints and delay sensitivity. Such diversity in application characteristics besides the unpredictable behavior of offloading requests will cause the heterogeneous computational and network resources contained in VEC infrastructure to exhibit transient and dynamic operational characteristics. These characteristics are mostly related to the utilization levels of available computational servers and the availability of network bandwidth. Second, VEC systems entail the collaboration of various entities such as vehicles, local edge servers and global cloud servers. While such a multi-component environment can lead to more versatility in task offloading, it also increases the state-space of task offloading complicating the decision to select the most appropriate entity to handle an offloaded task [5], [6]. As the dynamic changes to the VEC systems are difficult to predict or model in advance, an efficient offloading scheme should be able to learn while offloading; it should utilize its historical offloading data to steer its future offloading decisions considering both application-salient characteristics and current status of the VEC system [7]. This work targets task offloading in VEC systems with a predefined set of applications with each application having different processing, network bandwidth and timing requirements. The essence of task offloading in such systems is to enable vehicles or offloading decision makers to interact with potential offloading destinations via task offloading, learn their suitability to handle the offloaded tasks and utilize recent offloading history to guide current offloading decisions. As the set of possible offloading destinations in the considered VEC systems remain unchanged, task offloading can be formulated as a multi-armed bandit

(MAB) in which each possible offloading destination (i.e., computational server) is considered as an independent arm. Hence, pulling an arm at each round is equivalent to selecting a particular computational server to receive the offloaded task. This requires maintaining a reasonable trade-off between the exploitation (i.e., selecting the current best computational server based on past offloading decisions) and exploration (i.e., trying other servers to gain more useful and accurate information). In this regard, classical MAB solutions such as the Upper-Confidence Bound (UCB) and soft-max [8], [9], [10] become ineffective from multiple facets. On the one hand, offloading requests originate from different applications with distinct timing requirements and delay sensitivity levels hindering the process of reward formulation in the underlying MAB problem. On the other hand, the candidate arms (i.e., computational servers) may encounter dynamic changes due to their varying resource utilization levels and network connections status. To address these issues, this paper presents and evaluates three different approaches that leverage some contextual information about different application types and current status of computational servers to make offloading decisions. First, as the considered applications have different timing requirements and delay sensitivity levels, a MAB-based approach with application-dependent reward assignment is implemented and evaluated. Second, in order to ensure that an offloading decision for a particular task type is influenced only by the offloading history of similar tasks, another MAB approach, in which a dedicated bandit learner is maintained per each application type, is proposed and evaluated. Third, to cope with the dynamic and continuous changes of the VEC environment, two variations of a contextual-bandit algorithm are also proposed. This algorithm leverages incremental (online) learning to continuously adjust offloading decisions based on current environment changes. The rationale behind contextual bandits is to compute expected rewards as function of some contextual information. In order to capture variations between different arms for the same application type or variations of the same arm for different application types, this work implements contextual-bandits as an array of incremental learners with either one separate learner per arm (i.e., computational server) or one dedicated learner per each combination of arm and application type.

The rest of this paper is organized as follows. Section II discusses related research efforts. Section III presents the proposed algorithms. Section IV shows and discusses simulation results and Section V summarizes and concludes this paper.

II. Related Work

Task offloading in VEC environments has recently gained a noticeable interest among researchers. Several research efforts with different decision variables and optimization goals have been proposed. In these efforts, vehicles are assumed to offload some or all of their tasks using vehicles to everything (V2X) communication technologies. Typically, V2X is a general term that indicates different communication models used by the vehicles to offload their tasks. In this context, Vehicle to Vehicle (V2V), Vehicle to RSU (V2R), Vehicle to Pedestrian (V2P) and Vehicle to Infrastructure (V2I) can be utilized [11]. Sun et. al. [12] have proposed an adaptive learning based task offloading (ALTO) algorithm for the dynamic VEC systems. They have proposed a MAB-based solution that works in a distributed manner and targets minimizing the average delay of task offloading. However, their proposed algorithm focused on V2V task offloading. On the other hand, Zhang et. al. [7] have formulated task offloading as a mortal MAB problem in which tasks can be offloaded to neighboring edge nodes. While contextual information obtained from various edge nodes were considered when making an offloading decision, the presence of different applications with distinct characteristics and timing requirements was not considered. On the other hand, Xu et. al. [13] have formulated task offloading in VEC environments as a multi-objective optimization problem. They have solved the optimization problem using genetic algorithm with the goal of minimizing offloading latency and improving resource utilization. In their proposed method, tasks can be offloaded either to edge servers or other vehicles. However, since task offloading is an online problem and its constituent task characteristics and environment dynamics are not known in advance, finding an offline task offloading solution may not be effective in real-life.

Dai et al. [14] have formulated offloading destination selection and load balancing in VEC systems as a mixed-integer nonlinear programming problem. They have proposed an approximation heuristic algorithm to solve this problem. Their proposed algorithm is assumed to run on the vehicles in a distributed manner. In addition, they have assumed that some parts of the tasks can be executed locally using in-vehicle resources while the rest can be offloaded to the VEC server. However, dividing task execution into several parts and then combining the results is error-prone and may not be suitable for delay-sensitive applications such as accident prevention services.

Wang et al. [15] have employed a game theory-based technique to find the offloading probability of each vehicle in the VEC system. Their primary goal was to maximize the utility of each vehicle. The vehicles adjust their offloading probabilities by considering the offloading probability of other vehicles in the previous stage. Based on the computed probabilities, tasks can be executed locally, or offloaded to the edge server. However, they did not consider task offloading to the global cloud server neither did they consider the presence of applications with different requirements.

Liu et al. [16] have utilized a matching-based approach for minimizing the network delay associated with task offloading. In their work, the VEC system is composed of three layers that include the vehicles, RSUs and a macro base station (MBS). The MBS is responsible for performing task offloading and handover operations. Hence, all the vehicles and RSUs are assumed to be connected to the MBS. The matching algorithm operates iteratively based on matching requests sent from the vehicles to the RSUs. However, their work was based on a fixed-latency network model in which the latency of the wide-area network (WAN) is assumed to be fixed. Hence, their work did not consider the impact of network status on task offloading especially that the matching requests will create extra load on the available network bandwidth.

Feng et al. [17] have proposed a hybrid vehicular cloud (HVC) framework to increase the computing capacity of vehicles by utilizing computational resources of other neighboring vehicles, RSUs and the cloud. The goal of their proposed online algorithm is to increase the number of successfully offloaded and executed tasks while minimizing cellular net-

work usage. Their proposed algorithm seeks to first find the idle slots on other neighboring vehicles and RSUs considering both the estimated transmission and execution delays. If no idle slots are found on the neighbouring vehicles or the RSUs, the cellular network is used to access the cloud. All devices in the VEC system are assumed to work collaboratively by broadcasting a beacon message. Computational tasks are scheduled consecutively based on their anticipated transmission and processing delays. However, incorrect or misleading information provided by some malicious vehicles may cause some tasks to fail. On the other hand, Jiang et. al. [18] have introduced task replication technique to improve service reliability in VEC systems. In their proposed approach, task replicas can be simultaneously offloaded to multiple vehicles to be processed. However, one drawback of their proposed framework is that it needs frequent state information update and can place significant overhead on the network bandwidth.

Sonmez et al. [6] have recently proposed a machine learning-based task offloading scheme for VEC systems. They have considered a multi-access, multi-tier VEC architecture that consists of three main layers, namely, the vehicles, RSUs and cloud servers. In addition, they have also assumed a multi-access communication framework in which vehicular wireless local area network (WLAN), wide-area network (WAN) and cellular network can be used for V2I task offloading. Their task offloading scheme is based on a two-stage process in which dedicated regression and classification models are maintained per each potential offloading destination. During the first stage, the classification models are consulted to predict which devices could successfully handle the offloaded task. In the second stage, the regression models are employed to predict the time required to execute the offloaded task (i.e., service time) on each of the devices identified during the first stage. Thereafter, the device with the lowest predicted service time is chosen to receive the offloaded task. However, their work is based on having a static dataset to train the regression and classification models. However, such a static dataset may not be available in real-life as the VEC environment from which the data is collected changes continuously. In addition, their used regression and classification models remain static and do not acquire any new knowledge from the dynamic changes of VEC environment. In other words, when the VEC environment conditions to which the static models are exposed differ from those used for model training, their proposed offloading scheme may fall short and lead to poor performance.

In this work, three different online MAB-based task offloading schemes are implemented and evaluated based on the VEC architecture presented in [6]. The common theme among these schemes is to account for the presence of applications with different requirements and dynamically adjust the offloading decisions based on the dynamic conditions of the VEC system.

III. Online Vehicular Task Offloading

As task offloading requests are sequentially generated in a dynamic manner, task offloading becomes an online sequential decision making process that cannot be handled using traditional offline optimization tools. Instead, it can be formulated and solved using MAB theory. Hence, this work implements several MAB-based task offloading algorithms. In addition, it evaluates their performance in terms of the percentage of satisfied task offloading requests, task response time and QoE - under dynamically changing server utilization levels and network conditions.

A. VEC System Overview

A typical VEC system is composed of multiple edge servers augmented by the global cloud resources. In addition, the underlying communication infrastructure encompasses multiple technologies such as WLAN, MAN and WAN [19]. Such a heterogeneous architecture with time-varying offloading patterns leads to a dynamic scene that requires proper management of task execution. In this regard, the task offloading engine tries to preserve an efficient operation of the entire VEC system by selecting the best available computational server to receive an offloaded task. The decision on which server to choose is substantially demanding as it should consider both task characteristics, computational server utilization and network status. This work assumes a multi-tier and multi-access VEC system's architecture in which both local edge servers and global cloud servers can receive offloaded tasks [6]. In this architecture, vehicles can offload their computational tasks to the edge servers (i.e., V2R) or to the cloud servers (i.e., V2I). On the one hand, tasks can be offloaded to the edge servers using a short-range WLAN communication protocol such as the IEEE 802.11 used in [20], [21]. On the other hand, vehicles can offload their tasks to the cloud servers using the Internet connection (WAN), which provides a more flexible and high-bandwidth network interface. Similar to the model proposed in [6], vehicular tasks can be offloaded to the cloud either through the serving RSUs, which are assumed to use fiber connection to the cloud, or using the cellular network's broadband connection. Furthermore, the RSUs in the considered VEC architecture are also connected through a Metropolitan Area Network (MAN). This allows RSUs to form a shared resource pool in which task migration can be performed to handle the handover problem as proposed in [16]. In the assumed handover scheme, when a vehicle leaves the range of its current serving RSU before the results of the offloaded task are received, those results are transmitted to that vehicle in a multi-hop manner via the other RSUs in the VEC system. The handover process only fails if the offloading vehicle leaves the range of its current serving RSU while uploading a task or downloading a result.

Therefore, the considered VEC system allows vehicles to offload their tasks either to the edge sever, cloud server through RSU or cloud server through cellular network.

B. Task Offloading Algorithms

In this work, the considered algorithms are based on the MAB theory which is a Reinforcement Learning (RL) approach to maximize the total cumulative reward through sequential decision making. As shown in Fig. 1, a typical RL problem is modelled as an environment whose state is continuously observed by an agent. As shown, the agent observes the environment state (S1) and takes an action (A). Consequently, the environment responds by transitioning to state (S2) and sending a reward (R) to the agent. The reward may be positive or negative. Over a series of such trials and

errors, the agent learns an optimal policy (i.e., a mapping from states to actions) to maximize the long-term reward.

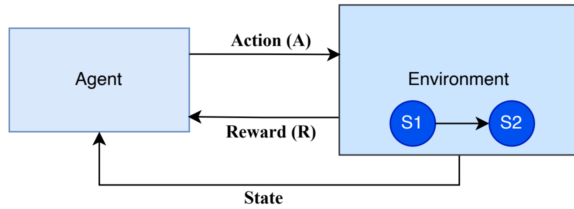


Fig. 1. Reinforcement Learning Flow.

In reality, the RL problem can be simply abstracted as a MAB problem. As shown in Fig. 2. MAB problems do not account for the environments and their state changes. In other words, an agent observes only the actions it takes and the associated rewards it receives and tries to compose the optimal strategy accordingly. The rationale behind solving MAB problems is to try and explore the actions involved in the action space and realize the unknown distributions of the rewards. Therefore, in MAB problems, the agents will ultimately try different actions and maintain a trade-off between exploration and exploitation to devise the optimal policy.

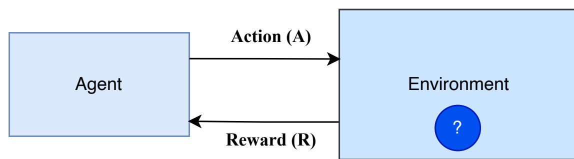


Fig. 2. Multi-armed Bandit Problem Flow.

Evidently, the main drawback of MABs is that the agents totally ignore the environment state when making an action. The environment state can provide significantly useful insights that can help the agent in devising an efficient policy much faster. Utilizing some useful elements of the environment state has introduced a new class of algorithms know as contextual or context-aware bandits [22], [23], [24], illustrated in Fig. 3. Here, instead of managing the trade-off between exploration and exploitation randomly, the agent obtains some context (i.e., contextual information) about the environment and utilizes that information to properly manage the actions. The notion of context is different from that of the state used in the RL problem formulation. A context is simply some useful knowledge about the environment that helps the agent take a proper action. For example, in the case of task offloading, the context may provide some information about the application type to which an offloaded task belongs.

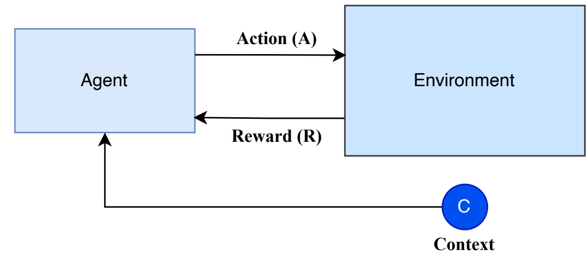


Fig. 3. Context-aware Multi-armed Bandit Problem Flow.

In the task offloading problem - formulated as a MAB or contextual MAB problem, the task offloading engine is the agent, the environment represents the VEC system while the reward is a numeric representation of user's perception of the quality of the offloading service. On the other hand, the action space consists of all offloading options i.e., offloading to the edge servers, offloading to the cloud severs via the RSU or offloading to the cloud via the cellular network. The following sections (i.e., III-B1, III-B2 and III-B3) describe each of the implemented context-aware task offloading algorithms.

1) Task offloading with application-dependent rewards: This section explains the implemented MAB-based task offloading algorithms in which the reward assigned to the agent is application-dependent. The reward assigned after receiving the results of an offloaded task is computed based on the observed task's response time, maximum tolerable delay of the application to which the offloaded task belongs and that application's delay sensitivity. The response time of an offloaded task can be computed as shown in equation 1.

$$\begin{aligned} T_R &= t_u + t_p + t_d \\ t_u &= \frac{T_s + T_{IN}}{Network_{ub}} \\ t_p &= \frac{T_{IC}}{Server_{MIPS}} \\ t_d &= \frac{T_{OUT}}{Network_{db}} \end{aligned} \tag{1}$$

Where T_R is the total response time of the offloaded task in seconds, t_u is the time required to upload the task and its input file to the selected server, t_p is the execution time of the offloaded task, t_d is the time required to download the results to the offloading vehicle, T_s is the size of the offloaded task's binary in Megabyte (MB), T_{IN} is the task's input file size in MB, $Network_{ub}$ the uplink bandwidth of the network connection associated with the selected arm (i.e., computational server) in MB/s, T_{IC} represents the instruction count of the offloaded task, $Server_{MIPS}$ is the processing capacity of the associated server in million instructions per second (MIPS), T_{OUT} is the task's output file size and $Network_{db}$ is the downlink bandwidth of the used network connection.

Assuming that the response time observed after the agent has offloaded a task (i), generated from an application (A) whose maximum delay requirement is T_{max} and delay sensitivity is α_A, is T_i. Then, the reward assigned to the agent (R_i) is computed as shown in equation 2. This formulation is based on the notion of Quality of Experience (QoE) proposed in [6].

$$R_i = \begin{cases} 0, & \text{if } i \text{ has failed} \\ 0, & \text{if } T_i \geq 2T_{max} \\ (1 - \frac{T_i - T_{max}}{T_{max}}).(1-\alpha_A), & \text{if } T_{max} \leq T_i < 2T_{max} \\ \alpha_A.R_{max}, & \text{if } T_i \leq T_{max} \end{cases} \quad (2)$$

Where R_{max} is the maximum possible reward, $\alpha_A \in [0, 1]$ and $1 - \alpha_A$ refers to the delay tolerance of the associated application. In other words, a high value of α_A indicates that the associated application is a delay-sensitive application while a low value of α_A represents a delay-tolerable application. Apparently, the value of the reward is directly linked to application characteristics (i.e., the maximum delay requirement and delay sensitivity). Hence, two similar response time values obtained for two different applications - with distinct requirements will be viewed differently by the agent. Consequently, the agent will reasonably scale the cumulative reward associated with a particular arm in proportion to application characteristics; while a selected arm (i.e., computational server) might be suitable for a particular application type, it may not satisfy the requirements of other application types.

In a MAB problem with k possible arms, there are k possible actions i.e., arm selection choices. Each action has an expected reward provided that the action is selected. This expected reward is known as the value of that action and denoted as $q_*(a)$. The action selected at time instant t is denoted as A_t. Hence, the value of an arbitrary action a, is the expected reward given that a is selected by the agent, as shown in equation 3 [25].

$$q_*(a) \doteq \mathbb{E}[R_t | A_t = a] \quad (3)$$

If the agent knew the value associated with each action, then it would be trivial to solve the MAB problem: the agent would always select the action with highest value. However, the agent does not know the action values with certainty, although it may have estimates. The estimated value of action a at time instance t is denoted as $Q_t(a)$. If the agent maintains estimates of the values of different actions, then at any time instant there exist at least one action whose estimated value is the highest. This action - with the highest estimated value is known as the greedy action. Hence, a simple action selection policy is to always pick the greedy action, as given in equation 4 [25].

$$A_t \doteq \underset{a}{\operatorname{argmax}}\, Q_t(a) \quad (4)$$

Where the $argmax$ operator returns the action that maximizes the enclosed expression. If the agent selects a greedy action, the agent is said to be exploiting its current knowledge of the values of the actions. If instead the agent picks one of the non-greedy actions, then the agent is said to be exploring, because this enables the agent improve its estimates of the values of the non-greedy actions. While exploitation is the right thing that the agent can do to maximize the expected reward on the one step, exploration may yield greater total reward in the long run. For example, suppose the value of the greedy action is known with certainty, while some other actions are anticipated to be nearly as good but with some high degree of uncertainty. The uncertainty is such that at least one of the other actions is probably better than the greedy action, but the agent does not know which one. If the agent has many time steps ahead on which to choose among actions, then it may be better to explore the non-greedy actions and identify which of them are better than the greedy action. Because the agent is not able to both explore and exploit with any single action selection, the conflict or trade-off between exploration and exploitation should be properly addressed. This work considers two possible MAB algorithms that handle the exploitation-exploration dilemma taking into account the uncertainty in the estimates of action values. These algorithms are the Upper-Confidence Bound (UCB) and the soft-max bandit algorithms [8], [9], [10]. The UCB action selection policy works based on the premise that it would be better to choose from the non-greedy actions in accordance with their potential for actually being optimal, considering both how close their estimates are to being maximal besides the uncertainties in those estimates. This action selection policy is given in equation 5 [8], [26].

$$A_t \doteq \underset{a}{\operatorname{argmax}} \left[Q_t(a) + c\sqrt{\frac{ln(N)}{N_t(a)}} \right] \quad (5)$$

Where N is the number of action selections performed by the agent, $N_t(a)$ is the number of times action a has been selected so far and $c > 0$ is the exploration control parameter. The rationale behind the UCB action selection is that the square root term is a measure of the uncertainty in the estimate of the value of action a. Hence, the quantity being max'ed over is therefore a kind of upper bound on the potential true value of action a, with parameter c determining the confidence level. When action a is selected by the agent, $N_t(a)$ increases and its associated uncertainty is reduced, and, since $N_t(a)$ appears in the denominator, the uncertainty term decreases as well. On the other hand, every time an action other than a is selected by the agent, the value of t increases but $N_t(a)$ does not. Hence, as t appears in the numerator, the estimate of uncertainty - associated with a increases. In addition, the use of the natural logarithm indicates that the increases in uncertainty get smaller over time, but are unbounded; all actions will ultimately be selected. However, actions with lower value estimates, or that have already been selected more often, will be selected by the agent with decreasing frequency over time.

On the other hand, the soft-max algorithm picks each action with a probability that is proportional to its current estimated value $Q_t(a)$ as shown in equation 6 [25], [26].

$$Pr\{A_t = a\} \doteq \frac{e^{Q_t(a)/\tau}}{\sum_{j=1}^{k} e^{Q_t(j)/\tau}} \quad (6)$$

Where τ is a temperature parameter used to control the randomness of action selection. When $\tau = 0$, the algorithm acts greedily. When τ increases to infinity, the algorithm will select actions uniformly at random. In other words, the soft-max algorithm learns a numerical preference for each action a, which is proportional to the action value (i.e., $Q_t(a)$). The larger the preference, the more frequently that action is selected.

In the two algorithms, after an action a is selected and a reward is received, the estimated value of that action is incrementally updated as shown in equation 7 [25].

$$Q_{t+1}(a) \doteq Q_t(a) + \frac{1}{N_a(t)}\left[R_t - Q_t(a)\right] \qquad (7)$$

Where $Q_{t+1}(a)$ is the new estimate of the action value, $Q_t(a)$ is the old estimate of that action's value, R_t (equation 2) is the recently received reward.

Algorithms 1 and 2 show high-level pseudo-codes of the UCB and soft-max algorithms, respectively. As shown, the two algorithms keep track of dynamically changing variables such as the number of times each arm/action has been selected (i.e., the N_a array) and the action value estimates (i.e., the Q array). On the other hand, each algorithm implements three basic functions, namely, $initialize$, $chooseArm$ and $updateArmValue$. The $initialize$ procedure is used to initialize algorithm's variables and data structures. The $chooseArm$ procedure is responsible for arm selection while the $updateArmValue$ is used to update the selected action's value after receiving the reward from the environment. While the $initialize$ and $chooseArm$ functions are similar in the two algorithms, the $chooseArm$ procedures are different. In the UCB algorithm, the $chooseARM$ procedure computes the UCB values of different actions based on their current value estimates and their uncertainty measures. Thereafter, it acts greedily on the computed UCB values. On the other hand, the $chooseArm$ procedure in the soft-max algorithm computes action probabilities - in proportion to their current value estimates and performs categorical draw to select actions based on their computed probabilities. Algorithm 3 shows a high-level abstraction of the main functions involved in the proposed MAB-based task offloading algorithm with application dependent reward assignment. First, the task offloading agent is initialized as either a UCB or soft-max algorithm (line 1). The utilized MAB algorithm is configured to have three possible actions , namely, offloading to the edge server (Edge), offloading to the cloud server via the RSU (cloudRSU) and offloading to the cloud server via the cellular network (cloudCN), as shown in lines 2 and 3 of algorithm 3. Every time an offloading request is received by the offloading agent, the $selectOffloadingDestination$ procedure (lines 4-9) is invoked. This procedure will call the associated MAB algorithm to select a particular server to handle the offloaded task (lines 5-6), update the task's meta-data to maintain an association between the task and the selected server (line 7) and then offload the task to the selected server (line 8). When the results of the offloaded task are returned from selected server, the $taskCompleted$ procedure (lines 10-22) will be called. This procedure will first check the completed task's meta-data to determine the server that has handled the task (lines 11-20) and then ask the MAB learner (i.e., algorithm) to update the value of that server based on the obtained reward (line 21). On the other hand, if the offloaded task fails, the $taskFailed$ procedure (lines 23-35) can be called to update the value of the associated server accordingly. As shown, the $taskCompleted$ and $taskFailed$ procedures will eventually call the $updateArmValue$ procedure defined in algorithms 1 and 2.

Algorithm 1 UCB Algorithm

Input: $Task$
Output: $selectedArm$

```
1: numArms ← number of arms
2: N_a[numArms] ← array of individual arm pulls
3: Q[numArms] ← array of action/arm values
4: procedure INITIALIZE(n)
5:     numArms ← n
6:     N ← 0
7:     for i ← 1 to numArms − 1 do
8:         N_a[i] ← 0
9:         Q[i] ← 0
10:    end for
11: end procedure
12: procedure CHOOSEARM()
13:    N ← 0
14:    for i ← 0 to numArms − 1 do
15:        count ← N_a[i]
16:        if count = 0 then
17:            N_a[i] ← 1
18:            return i
19:        end if
20:        N ← N + N_a[i]
21:    end for
22:    ucbQ[numArms] ← temporary array of UCB values
23:    for i ← 0 to numArms − 1 do
24:        ucbQ[i] ← Q[i] + sqrt(2*ln(N) / N_a[i])
25:    end for
26:    selectedArm ← 0
27:    for i ← 1 to numArms − 1 do
28:        newValue ← ucbQ[i]
29:        if newValue > ucbQ[selectedArm] then
30:            selectedArm ← i
31:        end if
32:    end for
33:    N_a[selectedArm] ← N_a[selectedArm] + 1
34:    return SelectedArm
35: end procedure
36: procedure UPDATEARMVALUE(arm, task, success)
37:    if success = False then
38:        reward ← 0
39:    else
40:        α_A ← task.delaySensitivity
41:        T_i ← task.responseTime
42:        T_max ← task.maxDelayRequirement
43:        R_max ← 1
44:        reward ← result of equation 2
45:    end if
46:    Q[arm] = Q(arm) + 1/N_a(arm) [reward − Q(arm)]
47: end procedure
```

2) Task offloading with application-dependent bandits: This section presents the application-dependent MAB-based task offloading algorithm. The basic idea behind this algorithm is to ensure that the offloading decision for a particular task is influenced by the offloading history of similar tasks i.e., tasks originating from similar application type. Algorithm 4 gives a pseudo-code of the application-dependent task offloading algorithm. As shown, the algorithm proceeds (lines 3-6) by

Algorithm 2 Soft-max Algorithm

Input: $task$
Output: $selectedArm$
1: $numArms \leftarrow$ number of arms
2: $N_a[numArms] \leftarrow$ array of individual arm pulls
3: $Q[numArms] \leftarrow$ array of action/arm values
4: $\tau \leftarrow$ *temperature value*
5: **procedure** INITIALIZE(n)
6: $numArms \leftarrow n$
7: $N \leftarrow 0$
8: **for** $i \leftarrow 1$ to $numArms - 1$ **do**
9: $N_a[i] \leftarrow 0$
10: $Q[i] \leftarrow 0$
11: **end for**
12: **end procedure**
13: **procedure** CHOOSEARM()
14: $sumQ \leftarrow 0$
15: **for** $i \leftarrow 0$ to $numArms - 1$ **do**
16: $sumQ \leftarrow sumQ + e^{Q(i)/\tau}$
17: **end for**
18: $probabilities[numArms] \leftarrow$ temporary array of soft-max probabilities
19: **for** $i \leftarrow 0$ to $numArms - 1$ **do**
20: $probabilities[i] \leftarrow \frac{e^{Q(i)/\tau}}{sumQ}$
21: **end for**
22: **return** $categoricalDraw(probabilities)$
23: **end procedure**
24: **procedure** CATEGORICALDRAW($probabilities$)
25: $rand \leftarrow$ *random double* $\in [0, 1]$
26: $cumulativeP \leftarrow 0.$ ▷ cumulative probability
27: **for** $i \leftarrow 0$ to $numArms - 1$ **do**
28: $cumulativeP \leftarrow cumulativeP + probabilities[i]$
29: **if** $cumulativeP > rand$ **then**
30: **return** i
31: **end if**
32: **end for**
33: **return** $numArms - 1$
34: **end procedure**
35: **procedure** UPDATEARMVALUE($arm, task, success$)
36: **if** $success = False$ **then**
37: $reward \leftarrow 0$
38: **else**
39: $\alpha_A \leftarrow task.delaySensitivity$
40: $T_i \leftarrow task.responseTime$
41: $T_{max} \leftarrow task.maxDelayRequirement$
42: $R_{max} \leftarrow 1$
43: $reward \leftarrow$ result of equation 2
44: **end if**
45: $Q[arm] = Q(arm) + \frac{1}{N_a(arm)}[reward - Q(arm)]$
46: **end procedure**

Algorithm 3 Task Offloading Algorithm with Application-dependent Rewards

Input: $task$
Output: *Selected Offloading Destination*
1: $taskOffloder \leftarrow MAB$
2: $servers$ []={Edge,cloudRSU,cloudCN}
3: $taskOffloder.Initialize(n = 3)$
4: **procedure** SELECTOFFLOADINGDESTINATION($task$)
5: $arm \leftarrow taskOffloder.chooseARM()$
6: $server \leftarrow servers[arm]$
7: *task.setAssociatedServer(server)*
8: *offload task to server*
9: **end procedure**
10: **procedure** TASKCOMPLETED($task$)
11: $server \leftarrow task.getAssociatedServer()$
12: **if** $server = Edge$ **then**
13: $arm = 0$
14: **end if**
15: **if** $server = cloudRSU$ **then**
16: $arm = 1$
17: **end if**
18: **if** $server = cloudCN$ **then**
19: $arm = 2$
20: **end if**
21: $taskOffloder.updateArmValue(arm, task, True)$
22: **end procedure**
23: **procedure** TASKFAILED($task$)
24: $server \leftarrow task.getAssociatedServer()$
25: **if** $server = Edge$ **then**
26: $arm = 0$
27: **end if**
28: **if** $server = cloudRSU$ **then**
29: $arm = 1$
30: **end if**
31: **if** $server = cloudCN$ **then**
32: $arm = 2$
33: **end if**
34: $taskOffloder.updateArmValue(arm, task, False)$
35: **end procedure**

initializing each offloading engine, associated with each application type, as a 3-arm MAB. This MAB can be either a UCB or a soft-max algorithm. In other words, the offloading agent maintains a separate offloading engine for each application type.

As shown, algorithm 4 implements three main procedures. The $selectOffloadingDestination$ procedure (lines 7-13) - used for server selection will first identify the application type (line 8), utilize the associated MAB learner to select a particular server (lines 9-10), record the task-to-server association (line 11) and then offload the task to the selected server (line 12). On the other hand, the $taskCompleted$ procedure (lines 14-27) - invoked upon the receipt of the offloaded task's results will use that task's type (line 15) and other meta-data (lines 16-25) to ask the related MAB learner to update the value of the server to which the task was offloaded (line 26). In addition, the $taskFailed$ procedure (lines 28-41) is responsible for handling task failure; it identifies the task type and that task's associated server (lines 29-39) and updates the server value accordingly (line 40). As algorithm 4 defines a dedicated MAB learner for each application type, the $updateArmValue$ procedure called in lines 26 and 40 is redefined to compute the value of the reward as shown in equation 8 instead of equation 2; tasks with the same application type are assumed to have the same value of delay sensitivity (α_A).

Algorithm 4 Task Offloading Algorithm with Application-dependent MAB Learners

Input: $task$
Output: *Selected Offloading Destination*

1: $numApps \leftarrow$ *Number of application types*
2: $servers$ []={Edge,cloudRSU,cloudCN}
3: **for** $i \leftarrow 0$ to $numApps - 1$ **do**
4: $taskOffloder[i] \leftarrow MAB$
5: $taskOffloder[i].Initialize(n = 3)$
6: **end for**
7: **procedure** SELECTOFFLOADINGDESTINATION($task$)
8: $type \leftarrow task.getApplicationType()$
9: $arm \leftarrow taskOffloder[type].chooseARM()$
10: $server \leftarrow servers[arm]$
11: *task.setAssociatedServer(server)*
12: *offload task to server*
13: **end procedure**
14: **procedure** TASKCOMPLETED($task$)
15: $t \leftarrow task.getApplicationType()$
16: $server \leftarrow task.getAssociatedServer()$
17: **if** $server = Edge$ **then**
18: $arm = 0$
19: **end if**
20: **if** $server = cloudRSU$ **then**
21: $arm = 1$
22: **end if**
23: **if** $server = cloudCN$ **then**
24: $arm = 2$
25: **end if**
26: $taskOffloader[t].updateArmValue(arm, task, True)$
27: **end procedure**
28: **procedure** TASKFAILED($task$)
29: $t \leftarrow task.getApplicationType()$
30: $server \leftarrow task.getAssociatedServer()$
31: **if** $server = Edge$ **then**
32: $arm = 0$
33: **end if**
34: **if** $server = cloudRSU$ **then**
35: $arm = 1$
36: **end if**
37: **if** $server = cloudCN$ **then**
38: $arm = 2$
39: **end if**
40: $taskOffloader[t].updateArmValue(arm, task, False)$
41: **end procedure**

$$R_i = \begin{cases} 0, & \text{if } i \text{ has failed} \\ 0, & \text{if } T_i \geq 2T_{max} \\ (1 - \frac{T_i - T_{max}}{T_{max}}), & \text{if } T_{max} \leq T_i < 2T_{max} \\ 1, & \text{if } T_i \leq T_{max} \end{cases} \tag{8}$$

3) Task offloading with incremental learning: This section presents two variations of an algorithm in which incremental (i.e., online) learning is used to guide task offloading agents. The presented algorithm is inspired by the idea of contextual bandits used in some domains such as recommendation systems [27], [28]. As shown in [22], the main principle in contextual bandits in to construct a linear model that can be used to predict the expected reward of choosing a particular action considering some contextual information. The parameters of this model are continuously updated based on the true observed reward. Hence, this work presents a new algorithm that employs the idea of contextual bandits for task offloading. The rationale behind this algorithm is to develop and maintain an online model that predicts a task response time based on contextual information such as task processing requirements, server utilization and network latency, as shown in equation 9 [22].

$$\mathbb{E}[T_t|\mathbf{x}_t] = f_{\boldsymbol{\theta}}(\mathbf{x}_t) = {\mathbf{x}_t}^T \boldsymbol{\theta} = \sum_{j=0}^{n} \theta_j x_{tj} \tag{9}$$

Where T_t is predicted response time, $\mathbf{x}_t$ is the context vector at time t and $\boldsymbol{\theta}$ is the model coefficients vector and n is the number of parameters in the context vector. The vector $\mathbf{x}_t$ contains contextual information related to application characterises and the VEC environment status. Hence, the goal of online learning process is to find the coefficients vector $\boldsymbol{\theta}$ that would minimize the error between the predicted response time (i.e, T_t) - computed before task offloading and the actual response time (T_{a_t}) observed after offloading. In other words, the learning process seeks to find the values of $\boldsymbol{\theta}$ that would minimize a particular cost function $C_{\boldsymbol{\theta}}$. As the model given in equation 9 represents a liner regression model, the squared loss function given in equation 10 is a suitable choice for the cost function [29]. Evidently, this function computes the squared error or difference between the predicted and observed response times. The average cost per training instance can be computed as given in equation 11, which computes the mean squared error (MSE) [29].

$$C_\theta = \frac{1}{2}(f_{\boldsymbol{\theta}}({\mathbf{x}_t}^{(i)}) - {T_{a_t}}^{(i)})^2 \tag{10}$$

$$MSE_\theta = \frac{1}{m}\sum_{i=1}^{m} \frac{1}{2}(f_{\boldsymbol{\theta}}({\mathbf{x}_t}^{(i)}) - {T_{a_t}}^{(i)})^2 \tag{11}$$

Where m is the number of training instances. Hence, the goal of model training is to find the values of $\boldsymbol{\theta}$ that would minimize the MSE over the whole training set. In traditional machine learning settings, the values of $\boldsymbol{\theta}$ are usually computed offline using some optimization techniques such the gradient decent - assuming the presence of a static training dataset. However, in the task offloading problem, a static training dataset of offloading requests does not usually exist; offloading requests appear as a dynamic stream of instances. Hence, the model parameters (i.e., values of $\boldsymbol{\theta}$) should be incrementally learned and adjusted based on the incoming offloading requests. In this regard, stochastic gradient decent (SGD) provides a viable tool to incrementally adjust the values of $\boldsymbol{\theta}$ based on individual training instances [30]. It can perform a parameter update for each training instance (${\mathbf{x}_t}^{(i)}$, ${T_{a_t}}^{(i)}$). After each training instance, the values of θ will be updated as shown in equation 12.

$$\theta_j = \theta_j - \eta \nabla_{\theta_j} C_\theta({\mathbf{x}_t}^{(i)}, {T_{a_t}}^{(i)}) \tag{12}$$

Where η is the learning rate and ∇_{θ_j} is the gradient of the cost function with respect to θ_j. Hence, this work employs SGD to dynamically fit and adjust linear models for response time prediction. Apparently, as the VEC environment contains different offloading options i.e., servers, a single linear model would not suffice for all possible servers. Hence, the first variant of the SGD-based offloading algorithm maintains three separate linear models for response time predictions; a model for edge server (SGD_{edge}) and two other models for cloud via RSU (SGD_{cRSU}) and cloud via CN (SGD_{cCN}), respectively. Table I summarizes the contextual features used by each model. Each model uses its own context vector to make response time predictions. While different context vectors contain some similar application characteristics such as the task's instruction count, each vector contains server-specific features such as that server's utilization level and the upload and download latencies of the associated network connection. It is worth noting that the SGD_{edge}, unlike cloud-related models, uses the current utilization of the edge server to predict response time. In general, edge servers are not as resource-enriched as cloud servers and their utilization levels could have a significant impact on the response time.

TABLE I. CONTEXTUAL FEATURES OF SGD-BASED MODELS

Model	Contextual features
SGD_{edge}	Task instruction count (T_{IC}), edge server utilization (U_e), WLAN upload latency ($WLAN_u$), WLAN download latency ($WLAN_d$)
SGD_{cRSU}	Task instruction count (T_{IC}), WAN upload latency (WAN_u), WAN download latency (WAN_d)
SGD_{cCN}	Task instruction count (T_{IC}), CN upload latency (CN_u), CN download latency (CN_d)

Algorithms 5 and 6 show pseudo-codes of the main parts of the incremental learning-based task offloading algorithm. These algorithms follow the notation of the WEKA API used for implementing the algorithm in the used simulation tool [31]. As show in algorithm 5, the algorithm initializes all models as an SGD-based linear models (lines 1-3). These models are initialized with arbitrary values of the model coefficients $\boldsymbol{\theta}$.

In order to allow the constructed models to make educated predictions of response time, the algorithm will first utilize a round-robin-based offloading until a batch of instances, for each server, with a predefined size is obtained. Each instance of the batch records the server's contextual features (Table I) - at the time of offloading besides the observed response time value under that context. Thereafter, a model, for each server, is trained on the associated batch (lines 4-6). When an offloading request is received, the $selectOffloadingDestination$ procedure (lines 7-15) is invoked. This procedure will first observe the context associated with each possible offloading option and call the constructed models to make response time predictions, with a prediction per each offloading option (lines 8-10). Then, the offloading option with the least predicted response time is chosen for task offloading. In addition, the incremental learning-based algorithm maintains a dictionary to keep track of the selected server's context at the time of offloading (lines 12-13). This dictionary will later be used for updating the respective model parameters when the result of offloading is disclosed.

On the other hand, when the results of offloading are suc-

Algorithm 5 Task Offloading Algorithm with Incremental Learning - 1

Input: $task$
Output: *Selected Offloading Destination*

```
1:  SGD_edge ← new SGD()
2:  SGD_cRSU ← new SGD()
3:  SGD_cCN ← new SGD()
4:  SGD_edge.buildModel(Batch_edge)
5:  SGD_cRSU.buildModel(Batch_cRSU)
6:  SGD_cCN.buildModel(Batch_cCN)
7:  procedure SELECTOFFLOADINGDESTINATION(task)
8:      t_edge ← SGD_edge.predict(Context_edge)
9:      t_cRSU ← SGD_cRSU.predict(Context_cRSU)
10:     t_cCN ← SGD_cCN.predict(Context_cCN)
11:     server ← server with minimum predicted t_server
12:     task.setAssociatedServer(server)
13:     taskDictionary.put(task.id, Context_server)
14:     offload task to server
15: end procedure
16: procedure TASKCOMPLETED(task)
17:     server ← task.getAssociatedServer()
18:     id ← task.getID()
19:     t_o ← observed response time
20:     if server = Edge then
21:         Context_edge = taskDictionary.remove(id)
22:         SGD_edge.update(Context_edge, t_o)
23:     end if
24:     if server = cloudRSU then
25:         Context_cRSU = taskDictionary.remove(id)
26:         SGD_cRSU.update(Context_cRSU, t_o)
27:     end if
28:     if server = cloudCN then
29:         Context_cCN = taskDictionary.remove(id)
30:         SGD_cCN.update(Context_cCN, t_o)
31:     end if
32: end procedure
33: procedure TASKFAILED(task)
34:     server ← task.getAssociatedServer()
35:     id ← task.getID()
36:     t_o ← observed response time
37:     if server = Edge then
38:         Context_edge = taskDictionary.remove(id)
39:         SGD_edge.update(Context_edge, t_p)
40:     end if
41:     if server = cloudRSU then
42:         Context_cRSU = taskDictionary.remove(id)
43:         SGD_cRSU.update(Context_cRSU, t_p)
44:     end if
45:     if server = cloudCN then
46:         Context_cCN = taskDictionary.remove(id)
47:         SGD_cCN.update(Context_cCN, t_p)
48:     end if
49: end procedure
```

cessfuly returned from the selected server, the $taskCompleted$ procedure (lines 16-32) will be called. This procedure will first retrieve the associated task's meta-data (i.e., the server chosen for offloading and taskID) (lines 17-18). It also makes use of the true observed response time (t_o) (line 19). Once the associated server is identified, the task dictionary will be

Algorithm 6 Task Offloading Algorithm with Incremental Learning - 2

Input: $task$
Output: *Selected Offloading Destination*
1: $numApps \leftarrow$ *Number of application types*
2: **for** $i \leftarrow 0$ to $numApps - 1$ **do**
3: $SGD_{edge}[i] \leftarrow$ new SGD()
4: $SGD_{cRSU}[i] \leftarrow$ new SGD()
5: $SGD_{cCN}[i] \leftarrow$ new SGD()
6: $SGD_{edge}[i].buildModel(Batch_{edge}[i])$
7: $SGD_{cRSU}[i].buildModel(Batch_{cRSU}[i])$
8: $SGD_{cCN}[i].buildModel(Batch_{cCN}[i])$
9: **end for**
10: **procedure** SELECTOFFLOADINGDESTINATION($task$)
11: $type \leftarrow task.getApplicationType()$
12: $t_{edge} \leftarrow SGD_{edge}[type].predict(Context_{edge})$
13: $t_{cRSU} \leftarrow SGD_{cRSU}[type].predict(Context_{cRSU})$
14: $t_{cCN} \leftarrow SGD_{cCN}[type].predict(Context_{cCN})$
15: $server \leftarrow$ server with minimum predicted t_{server}
16: *task.setAssociatedServer(server)*
17: $taskDictionary.put(task.id, Context_{server})$
18: *offload task to server*
19: **end procedure**
20: **procedure** TASKCOMPLETED($task$)
21: $server \leftarrow task.getAssociatedServer()$
22: $id \leftarrow task.getID()$
23: $t_o \leftarrow$ observed response time
24: $type \leftarrow task.getApplicationType()$
25: **if** $server = Edge$ **then**
26: $Context_{edge} = taskDictionary.remove(id)$
27: $SGD_{edge}[type].update(Context_{edge}, t_o)$
28: **end if**
29: **if** $server = cloudRSU$ **then**
30: $Context_{cRSU} = taskDictionary.remove(id)$
31: $SGD_{cRSU}[type].update(Context_{cRSU}, t_o)$
32: **end if**
33: **if** $server = cloudCN$ **then**
34: $Context_{cCN} = taskDictionary.remove(id)$
35: $SGD_{cCN}[type].update(Context_{cCN}, t_o)$
36: **end if**
37: **end procedure**
38: **procedure** TASKFAILED($task$)
39: $server \leftarrow task.getAssociatedServer()$
40: $id \leftarrow task.getID()$
41: $t_o \leftarrow$ observed response time
42: $type \leftarrow task.getApplicationType()$
43: **if** $server = Edge$ **then**
44: $Context_{edge} = taskDictionary.remove(id)$
45: $SGD_{edge}[type].update(Context_{edge}, t_p)$
46: **end if**
47: **if** $server = cloudRSU$ **then**
48: $Context_{cRSU} = taskDictionary.remove(id)$
49: $SGD_{cRSU}[type].update(Context_{cRSU}, t_p)$
50: **end if**
51: **if** $server = cloudCN$ **then**
52: $Context_{cCN} = taskDictionary.remove(id)$
53: $SGD_{cCN}[type].update(Context_{cCN}, t_p)$
54: **end if**
55: **end procedure**

accessed to obtain the context associated with the received task. Then, the associated model will be updated using SGD-based parameter update (equation 12) (lines 20-31). In order to maintain a relatively small size of the task dictionary, the entry associated with the returned task will be deleted from dictionary upon the receipt of that task's results. When the offloaded task fails, the $taskFailed$ procedure (lines 33-49) is called. This procedure operates in a manner that resembles that of the $taskCompleted$ procedure. However, the associated model is updated with a penalty value (t_p). This value is set such that the associated model is updated in a way that forces it to predict a high value of response time for upcoming offloading requests. Such a high predicted value would potentially prevent the offloading engine from choosing the respective server for subsequent tasks.

The other variant of the incremental learning-based algorithm (algorithm 6) maintains, for each server, an array of SGD-based models. Each model in the array can be used to make response time predictions for a particular application type. As shown, this algorithm initializes and fits preliminary models for different application types (lines 2-9). On the other hand, the three essential procedures in this algorithm are similar to those of algorithm 5; the only distinction is that these procedures will first identify the application type to which the task belongs and then use the associated model accordingly.

IV. RESULTS AND ANALYSIS

In order to assess the performance of the proposed algorithms, they have been implemented and evaluated using the EdgeCloudSim simulation tool [32]. EdgeCloudSim provides a simulation environment for Mobile Edge computing (MEC) and VEC systems [33], [6]. It allows modeling of computational servers, network infrastructure and mobile vehicles. It also allows users to defined different application types with varying characteristics. The vehicular mobility model assumed in this work is similar to that of [6]. In this model, its is assumed that a 16 km road is divided into 40 400-meter segments with each segment having a dynamic velocity value and covered by a single RSU. Hence, the speed of a vehicle dynamically varies based on the type of segment it moves on. This allows to differentiate the traffic density on each segment of the road and, consequently, the demand placed on the associated vehicular resources especially the edge servers (i.e., RSUs) covering different road segments and their associated network connection's bandwidth. When the simulation is started, vehicles are assigned random locations on the road and move in a single direction with a predefined segment-dependent speed. In addition, the road is defined as a circular route keeping the number of vehicles the same for the entire simulation time. This work assumes a VEC system with three representative application types, namely, traffic management, danger assessment and infotainment applications. Application characteristics are shown in Table II. Configuration parameters of the computational servers and network resources are shown in Table III.

The presented algorithms are compared to other existing algorithms that include the MAB-based algorithm (MAB) [12], game theory-based (Game-Theory) [15], machine learning-based (ML_based) besides the time series forecasting-based

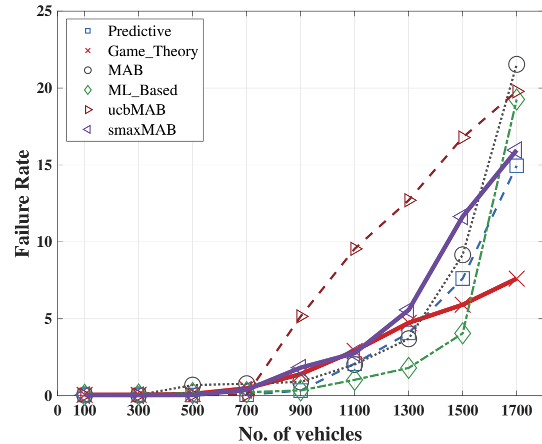
TABLE II. VEHICULAR APPLICATION CHARACTERISTICS

	Traffic Management	Danger Assessment	Infotainment Application
Vehicles percentage	30%	35%	35%
Input file size (KB)	20	40	20
Output file size (KB)	20	20	80
Inter-arrival time (second)	3	5	15
Instruction count ($\times 10^9$)	3	10	20
Utilization on Edge VM (%)	6	20	40
Utilization on Cloud VM (%)	1.6	4	8
Delay sensitivity (α)	0.6	0.90	0.35
Maximum delay requirement (second)	0.50	1.25	1.75
Penalty value (t_p)	1.25	2.25	2.5

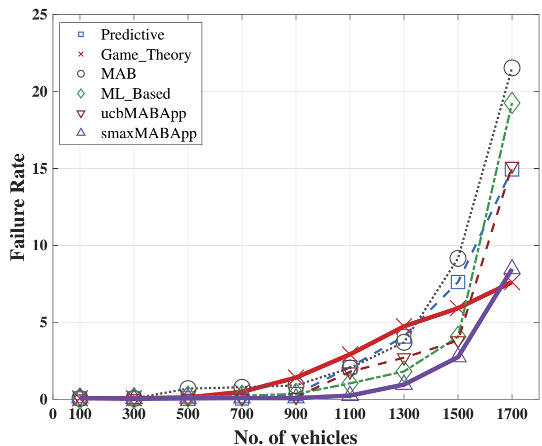
TABLE III. SIMULATION PARAMETERS

Parameter	Value
Simulation time (minutes)	60
Number of vehicles	100 - 1700
Vehicles counter size	200
No. of virtual Machines(VMs) on cloud server	20
No. of VMs on edge server	2
Processing capacity of cloud VM (MIPS)	75000
Processing capacity of edge VM (MIPS)	20000
WLAN range (meter)	200
WLAN bandwidth (Mbps)	100
MAN bandiwdth (Mbps)	1000
WAN bandwidth (Mbps)	50
WAN propagation delay (second)	0.15
CN bandwidth (Mbps)	20
CN propagation delay (second)	0.16
Maximum reward (R_{max})	1
Pre-training batch size (instances)	100

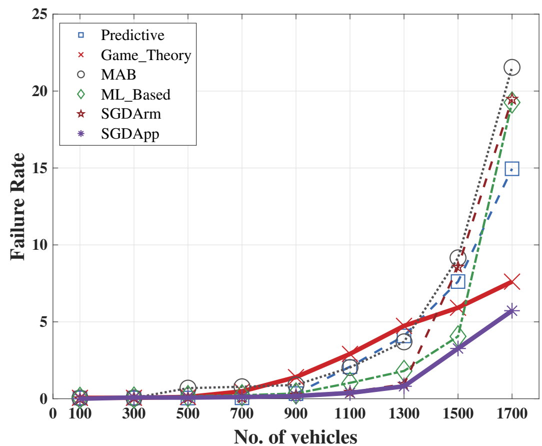
(Predictive) algorithms [6]. Comparison results cover all possible implementations of algorithms 3 and 4. On the one hand, there are two possible implementations of algorithm 3; using either UCB or soft-max algorithms, denoted as $ucbMAB$ and $smaxMAB$, respectively. On the other hand, algorithm 4 can also be implemented using either UCB (denoted as $ucbMABApp$) or soft-max (denoted as $smaxMABApp$). In addition, algorithm 5 is denoted as $SGDArm$ while algorithm 6 is denoted as $SGDApp$. In VEC systems, failure rate is an important factor in assessing the performance of different task offloading schemes. Typically, an offloaded task would fail if a virtual machine (VM), on the selected server, has very high utilization that prevents it from executing the offloaded task, or if the available network bandwidth of the selected server is not sufficient to upload/download the input/output of the offloaded task. In other words, an offloaded task can fail due to unavailability of computational or networking resources. Fig. 4 shows and compares the average task failure rate, among all application types, under different offloading algorithms, as the number of vehicles is increased from 100 to 1700. In general, the failure rate increases as the number of vehicles is increased. As shown in Fig. 4a, the proposed MAB-based offloading algorithms with application-dependent reward (i.e., $ucbMAB$ and $smaxMAB$) perform reasonably well - in terms of failure rate for small to moderate number of vehicles ($\leq$ 700), as compared to other competitor algorithms. However, their associated failure rates increase significantly as the number of vehicles increases beyond 700 vehicles. On the other hand, Fig. 4b compares the failure rate of the MAB-based offloading with separate application-dependent MAB learners (i.e., algorithm 4) to that of other existing algorithms.



(a)



(b)



(c)

Fig. 4. Failure Rate Comparison: (a) Algorithm 3, (b) Algorithm 4, (c) Algorithms 5 and 6.

As shown, the $smaxMABApp$ has outperformed the vast majority of other algorithms in terms of failure rate; having a dedicated MAB learner per application type allows the agent to devise an efficient offloading policy in which offloading decisions are based on the offloading history of similar tasks. While the $ucbMABApp$ algorithm has achieved relatively low failure rate when the number of vehicles is less than 900, its performance has deteriorated in response to increasing the number of vehicles. Furthermore, Fig. 4c compares the failure rate under the proposed $SGDArm$ and $SGDApp$ algorithms to that under other algorithms. As shown, the $SGDArm$ algorithm, in which an online SGD learner is maintained per each offloading option, has outperformed all competitor algorithms when the number of vehicles is less than 1300. However, its performance has significantly dropped beyond 1300 vehicles; as a single model is shared among all application types, updating the SGD-based model with a penalty value associated with a particular application type may adversely affect the offloading decision for subsequent tasks with different types. On the other hand, the $SGDApp$ algorithm, in which an array of SGD-based learners is maintained, has shown significant improvement in failure rate especially under high number of vehicles (i.e., $\geq$ 1500). This can be due to that fact that $SGDApp$ maintains an array of learners per each server with a dedicated model for each application type and, consequently, ensures that model updates are performed due to tasks with similar characteristics. Hence, the $SGDApp$ algorithm has maintained consistent learning pattern achieving an efficient utilization of available contextual information and previous offloading history to steer current offloading decisions. For the other competitor algorithms, the failure rate of ML_Based and the Predictive algorithms [6], MAB [12] have significantly increased for high number of vehicles (i.e., $\geq$ 1500). On other hand, the game theory-based algorithm [15] has witnessed a noticeable linear increase in failure rate as the number of vehicles is increased beyond 900 vehicles. As compared to other algorithms especially in the more congested situation (i.e., no. of vehicles = 1700), $SGDApp$ has achieved a failure rate reduction that ranges from 24.81%, as compared to GAME_Theory [15], to 73.43% as compared to MAB [12]. As for the other competitor algorithms, they tend to have acceptable failure rate values when the number of vehicles is less than 900. However, their associated failure rates start increasing after 900 vehicles. For instance, the ML_based algorithm offloads tasks with small instruction count to the nearby edge servers and the tasks with larger instruction count to the remote cloud server. However, the ML_based algorithm relies on statically trained models that do not gain any knowledge from the outcomes of the online offloading decisions. Hence, as the VEC system becomes more congested, its failure rate become worse. On the other hand, the MAB-based algorithm in [12] is also aware of the task's instruction count. However, it falls short when the VEC system becomes congested, and consequently, the failure rate starts increasing. In addition, the game theory-based algorithm tends to offload the majority of tasks to the edge servers regardless of their type. Hence, the lack of computing capacity at the edge servers lead to more task failure. Therefore, the task failure situation is noticeable after 900 vehicles and increases linearly, with respect to the number of vehicles. Furthermore, the predictive algorithm does not consider task characteristics when making offloading decisions; it increases the probability of selecting an offloading destination that has recently provided better results. Hence, such a strategy would become inadequate when the VEC system becomes overloaded.

Admittedly, service time (a.k.a. response time) is another important evaluation metric. It represents the total time required to offload a task and obtain its results. Fig. 5 shows and compares the average service time values obtained under different offloading algorithms, with respect to the number of vehicles, for the successfully executed tasks.

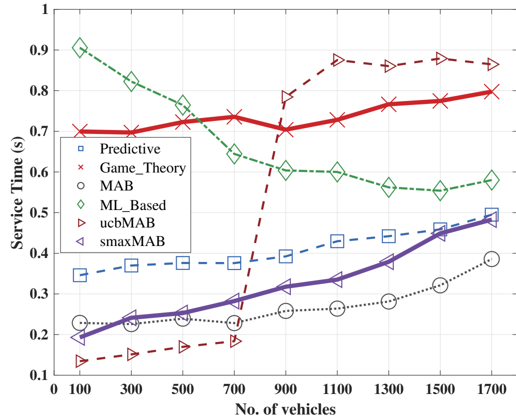
In general, the average service time increases with respect to the number of vehicles with the MAB algorithm [12] yielding the lowest service time among all competitor algorithms. As shown in Fig. 5a, the proposed $ucbMAB$ algorithm has outperformed all other counterparts when the number of vehicles is less than 700 but its performance has dropped significantly after 700 vehicles. Similarly, the proposed $smaxMAB$ algorithm has attained relatively acceptable service time values for systems in which the number of vehicles is less than 700. On the other hand, Fig. 5b illustrates that the proposed $ucbMABApp$ and $smaxMABApp$ algorithms have shown a profound ability to minimize average service time values for systems with less than 700 vehicles.

While the performance of $ucbMABApp$ has dropped after 700 vehicles, $smaxMABApp$ has consistently maintained a comparable performance to that of its MAB counterpart in [12]. Furthermore, Fig. 5c shows that the proposed $SGDArm$ and $SGDApp$ algorithms surpass all their counterparts for systems with at most 700 vehicles. However, as the VEC system becomes more congested, their obtained service time values increase. Nevertheless, the $SGDApp$ algorithm has maintained a steadily comparable service time to that of the MAB algorithm in [12]. It is worth noting that although the MAB algorithm presented in [12] has achieved low average service time for the successfully executed tasks, it has suffered significant increases in failure rate in congested VEC systems as shown in Fig. 4.

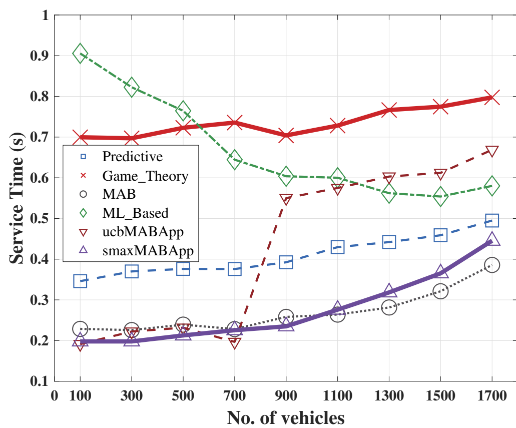
Although the failure rate and average service time provide adequate measures to evaluate task offloading algorithms, considering them as individual metrics may provide misleading results in systems where offloaded tasks may be lost. For instance, it may not be acceptable to have a low average service time for successfully executed tasks while having a high failure rate. Therefore, this work utilizes the Quality of Experience (QoE) formula proposed in [6], which combines both the service time and task failure as shown in equation 13.

$$QoE_i = \begin{cases} 0, & \text{if } i \text{ has failed} \\ 0, & \text{if } T_i \geq 2T_{max_i} \\ (1 - \frac{T_i - T_{max_i}}{T_{max_i}}).(1-\alpha_A) \times 100\%, & \text{if } T_{max_i} \leq T_i < 2T_{max_i} \\ 100\%, & \text{if } T_i \leq T_{max_i} \end{cases} \quad (13)$$

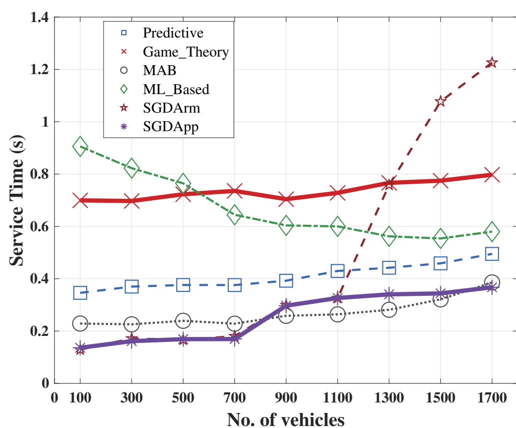
where T_i is the response time of task i, T_{max_i} is the maximum delay requirement of that task and α_A is that task's delay sensitivity. Evidently, the average QoE value decreases when task i is completed later than its associated delay requirement. If the observed service time exceeds twice the tasks' delay requirement or if task i fails, the QoE value is



(a)

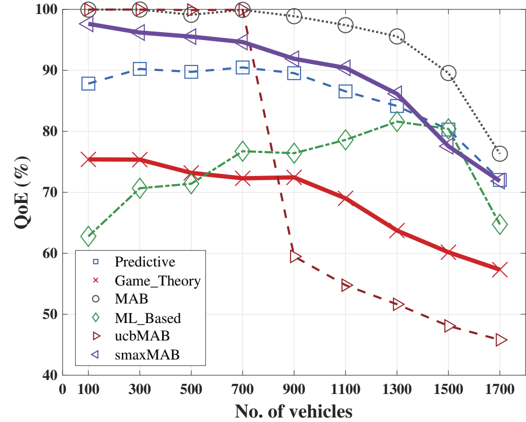


(b)

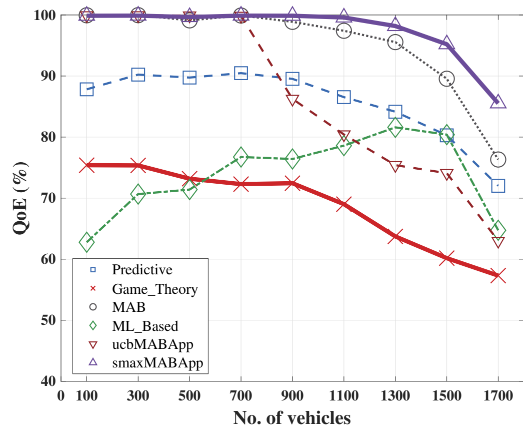


(c)

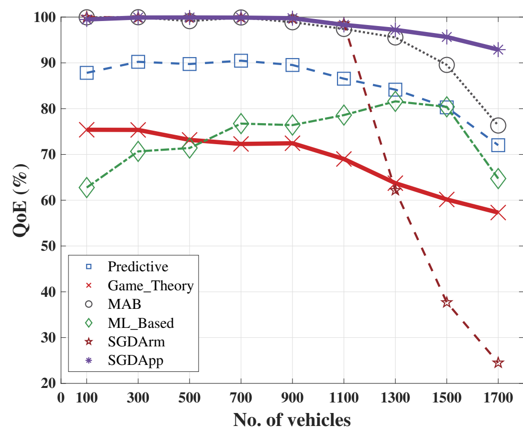
Fig. 5. Service Time Comparison: (a) Algorithm 3, (b) Algorithm 4, (c) Algorithms 5 and 6.



(a)



(b)



(c)

Fig. 6. Quality of Experience (QoE) Comparison: (a) Algorithm 3, (b) Algorithm 4, (c) Algorithms 5 and 6.

set to 0. Hence, the QoE metric provides a unified metric to match the observed service time to the delay requirements of different tasks besides assessing the balance between service time and failure rate. Fig. 6 depicts the average QoE values as a function of the number of vehicles.

As shown in Fig. 6a, the proposed $ucbMAB$ and its MAB counterpart have provided the maximum QoE (100%) when the number of vehicles is low i.e., less than 700; because of their low response time and failure rate. Similarly, the proposed $ucbMAXApp$ and $smaxMABApp$ algorithms besides the

MAB algorithm in [12] have outperformed other algorithms in terms of QoE when the number of vehicles is less than 700, as shown in Fig. 6b. In addition, the $smaxMABApp$ algorithm has maintained a reasonably higher QoE values for more congested systems i.e., when the number of vehicles exceeds 700. Furthermore, Fig. 6c demonstrates the ability of the proposed $SGDArm$ and $SGDApp$ algorithms to achieve a 100% QoE for systems with up to 1100 vehicles, with the $SGDApp$ algorithm maintaining its superiority over other algorithms beyond 1100 vehicles. As shown in Fig. 6b and 6c, the proposed $smaxMABApp$ and $SGDApp$ algorithms have achieved 12.05% and 21.70% improvement in QoE as compared to their MAB counterpart, respectively, when the number of vehicles is equal to 1700.

Considering the competitor algorithms, the ML_based, predictive and game-theory-based algorithms provide the lowest QoE values. On the one hand, the ML_based algorithm is not able to respond to the dynamic changes of the VEC environment such as network bandwidth and server utilization. In other words, it is not able to dynamically adjust its offloading policy as it depends on statically trained models that would yield poor performance if the run-time conditions differ from those observed during offline model building. On the other hand, the game theory-based provides poor QoE readings because the main objective of the game model is neither to minimize the service time nor to improve failure rate but rather to attain a stable equilibrium. For the predictive algorithm, its QoE values never exceed 90% even with no task failure as it does not essentially minimize service time. Furthermore, The MAB-based algorithm (i.e., MAB [12]) does not guarantee the delay requirements of different task types as the number of vehicles exceeds 900. This can be due to the mismatch between the offloaded task's processing demand and the selected offloading destination violating that task's delay requirements.

Apparently, the failure rate, service time and QoE results of the proposed $SGDApp$ algorithm prove the ability of the proposed contextual incremental learning scheme to handle task offloading in VEC systems with diversified application characteristics. In fact, incremental learning allows the task offloading algorithm to dynamically construct a robust offloading policy that efficiently utilizes the available contextual information and offloading history to guide subsequent offloading decisions. In other words, incremental learning allows the constructed models to gain new knowledge at run-time and vary model parameters in accordance with recently observed offloading outcomes. In order to prove the ability of the $SGDApp$ algorithm to behave in a VEC system with different application type, its behaviour has further been analyzed and compared to its MAB counterpart [12], as the later has shown almost the best performance among other competitor algorithms. In this regard, Fig. 7 shows the task offloading distribution, considering all application types, under the $SGDApp$ and the MAB algorithms. It shows the percentage of tasks offloaded to each of the edge and cloud servers. As shown, the proposed $SGDApp$ algorithm has maintained more balanced utilization of edge and cloud units (Fig. 7a) as compared to its MAB counterpart (Fig. 7b); it has offloaded an almost identical proportion of tasks to each of the edge and cloud servers.

In order to gain further insight about the offloading behavior for different application types. Fig. 8 and 9 show the task offloading distributions for the traffic management and infotainment applications, respectively, under the $SGDApp$ and the MAB algorithms. As shown in Table II, the traffic management application is characterized by having small tasks i.e., tasks with lower instruction count and average input/output file sizes. On the other hand, the infotainment application has larger tasks i.e., tasks with higher instruction count and average input/output file size, as compared to other application types.

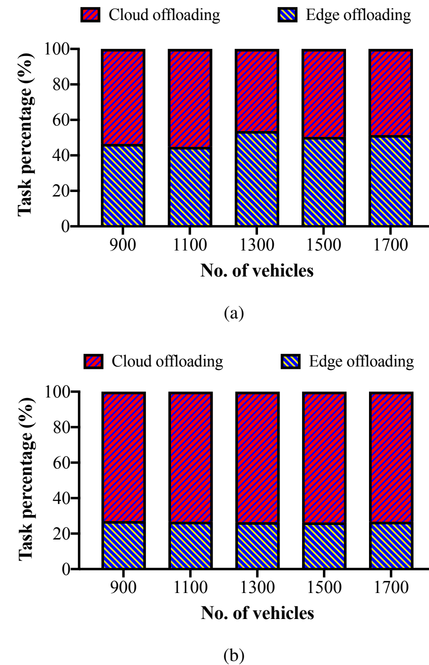
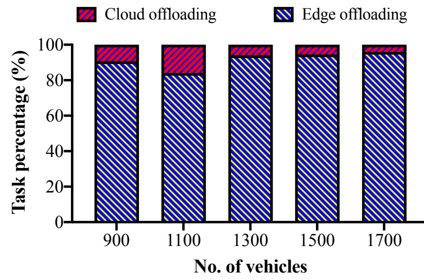


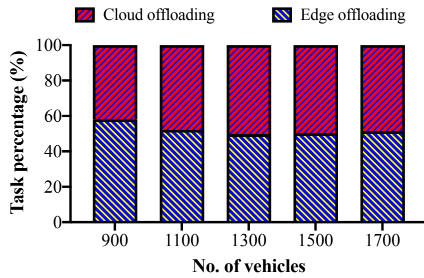
Fig. 7. Offloaded Tasks Distribution - all Application Types: (a) SGDApp, (b) MAB [12].

As shown in Fig. 8a, the $SGDApp$ algorithm tends to send the vast majority of the small tasks - generated from the traffic management application to the nearby edge servers. However, the MAB algorithm sends an almost equal proportion of the small tasks to each of the edge and cloud servers, as shown in Fig. 8b. On the other hand, Fig. 9a illustrates that the $SGDApp$ algorithm sends the vast majority of the large infotainment tasks to the cloud server as opposed to MAB algorithm that sends all infotainment tasks to the cloud server, as shown in Fig. 9b.

Therefore, it can be observed that $SGDApp$ is able to construct for each application type a model that would utilize the available contextual information to better steer that application's offloading decisions. In other words, the relatively small instruction count and file size of the traffic management tasks align with the processing and bandwidth capabilities of the edge servers. Consequently, sending these tasks to the edge servers has saved more cloud's processing capacity and network bandwidth for the processing- and bandwidth-hungry tasks such as infotainment tasks. On the other hand, sending small tasks to the cloud server in case of the MAB algorithm was harmful for all applications; it has caused higher service times for the small tasks and resulted in more resource contention with the large tasks on the cloud resources.

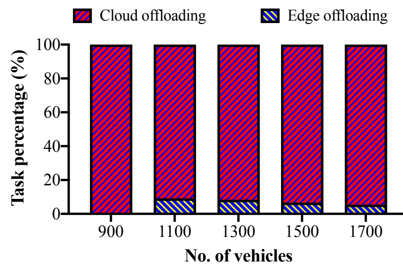


(a)

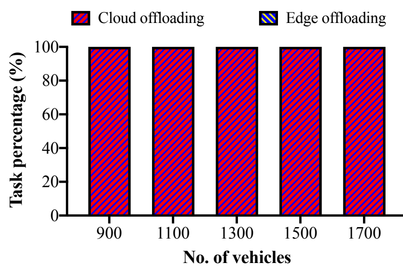


(b)

Fig. 8. Offloaded Tasks Distribution - Traffic Management: (a) SGDApp, (b) MAB [12].



(a)

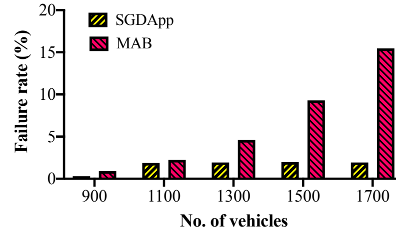


(b)

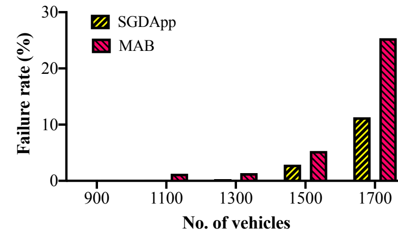
Fig. 9. Offloaded Tasks Distribution - Infotainment: (a) SGDApp, (b) MAB [12].

Ultimately, the offloading decision made for each application type has a direct consequence on that application's failure rate and QoE metrics. As shown in Fig. 10, the proposed $SGDApp$ algorithm has obtained noticeable improvement in failure rate as compared to its counterpart. It has achieved better failure rate values for both the small traffic management tasks (Fig. 10a) and the large infotainment tasks (Fig. 10b). The $SGDApp$ algorithm has achieved up to 87.5% and 55.4% improvement in failure rate for the traffic management and infotainment applications, respectively, when the number of vehicles is 1700.

Similarly, the proper offloading decisions made by the $SGDApp$ algorithm has achieved better QoE for both small and large tasks especially in more loaded VEC systems with the number of vehicles exceeding 1300, as shown in Fig. 11.

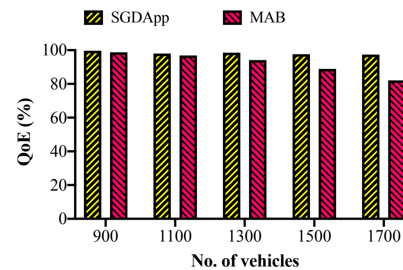


(a)

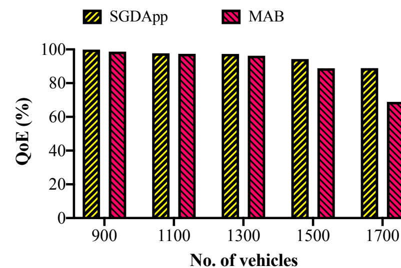


(b)

Fig. 10. Failure Rate of Different Applications: (a) Traffic Management, (b) Infotainment.



(a)



(b)

Fig. 11. QoE of Different Applications: (a) Traffic Management, (b) Infotainment.

The proposed $SGDApp$ algorithm has consistently maintained higher QoE values for both application types, with respect to the number of vehicles. As depicted in Fig. 11a, the $SGDApp$ algorithm has achieved up to 18.7% improvement in QoE, as compared to its counterpart, in a VEC system with 1700 vehicles. On the other hand, Fig. 11b illustrates the superiority of the $SGDApp$ algorithm; its QoE improvement has reached 29.02% as the VEC system becomes more congested, with 1700 vehicles.

V. Conclusion

Vehicular Edge Computing (VEC) systems have recently been introduced to provide a seamless integrated computing platform to execute various kinds of vehicular applications. In these systems, computational tasks generated from in-vehicle applications are offloaded to either the edge or the cloud servers. In addition, VEC systems are characterized by a dynamically changing resource utilization besides having to handle diversified application types. Hence, an efficient task offloading scheme is required to ensure appropriate selection of offloading destinations, considering both application characteristics and the status of VEC resources. Therefore, this paper has presented a number of Multi-Armed Bandit (MAB) algorithms for task offloading in VEC systems with a representative set of applications. The rationale behind the proposed algorithms is to utilize contextual information such as application type and current resource utilization to achieve efficient application-specific offloading decisions. The proposed algorithms were implemented based on either a single MAB learner with application-dependent reward formulation, multiple dedicated MAB learners with a specific learner for each application type or a contextual bandits approach - based on incremental learning methods. The proposed algorithms were thoroughly analyzed and compared to other closely related task offloading algorithms. Simulation results proved the ability of the proposed contextual bandits-based algorithm to surpass all other algorithms under the failure rate and QoE metrics besides achieving adequately comparable service time values. In addition, it demonstrated the ability to efficiently utilize the available VEC resources and make the most appropriate decision for each application type, considering the interplay between application characteristics, timing requirements, server's computational capacity and network status. Hence, utilizing contextual information to dynamically construct and adjust incremental learning models has proved its feasible applicability for task offloading in VEC systems.

References

[1] D.-K. Choi, J.-H. Jung, H.-B. Nam, and S.-J. Koh, "Agent-based in-vehicle infotainment services in internet-of-things environments," *Electronics*, vol. 9, no. 8, 2020.

[2] S. Mozaffari, O. Y. Al-Jarrah, M. Dianati, P. Jennings, and A. Mouzakitis, "Deep learning-based vehicle behavior prediction for autonomous driving applications: A review," *IEEE Transactions on Intelligent Transportation Systems*, pp. 1–15, 2020.

[3] L. Liu, C. Chen, Q. Pei, S. Maharjan, and Y. Zhang, "Vehicular edge computing and networking: A survey," *Mobile Networks and Applications*, Jul 2020.

[4] S. Raza, W. Liu, M. Ahmed, M. R. Anwar, M. A. Mirza, Q. Sun, and S. Wang, "An efficient task offloading scheme in vehicular edge computing," *Journal of Cloud Computing*, vol. 9, no. 1, p. 28, Jun 2020.

[5] Y. Wang, S. Wang, S. Zhang, and H. Cen, "An edge-assisted data distribution method for vehicular network services," *IEEE Access*, vol. 7, pp. 147 713–147 720, 2019.

[6] C. Sonmez, C. Tunca, A. Ozgovde, and C. Ersoy, "Machine learning-based workload orchestrator for vehicular edge computing," *IEEE Transactions on Intelligent Transportation Systems*, pp. 1–13, 2020.

[7] R. Zhang, P. Cheng, Z. Chen, S. Liu, Y. Li, and B. Vucetic, "Online learning enabled task offloading for vehicular edge computing," *IEEE Wireless Communications Letters*, vol. 9, no. 7, pp. 928–932, 2020.

[8] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine Learning*, vol. 47, no. 2, pp. 235–256, 2002.

[9] J. Vermorel and M. Mohri, "Multi-armed bandit algorithms and empirical evaluation," in *Machine Learning: ECML 2005*, J. Gama, R. Camacho, P. B. Brazdil, A. M. Jorge, and L. Torgo, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 437–448.

[10] Y.-L. He, X.-L. Zhang, W. Ao, and J. Z. Huang, "Determining the optimal temperature parameter for softmax function in reinforcement learning," *Applied Soft Computing*, vol. 70, pp. 80–85, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1568494618302758

[11] Z. MacHardy, A. Khan, K. Obana, and S. Iwashina, "V2x access technologies: Regulation, research, and remaining challenges," *IEEE Communications Surveys Tutorials*, vol. 20, no. 3, pp. 1858–1877, 2018.

[12] Y. Sun, X. Guo, J. Song, S. Zhou, Z. Jiang, X. Liu, and Z. Niu, "Adaptive learning-based task offloading for vehicular edge computing systems," *IEEE Transactions on Vehicular Technology*, vol. 68, pp. 3061–3074, 04 2019.

[13] X. Xu, Y. Xue, X. Li, L. Qi, and S. Wan, "A computation offloading method for edge computing with vehicle-to-everything," *IEEE Access*, vol. 7, pp. 131 068–131 077, 2019.

[14] Y. Dai, D. Xu, S. Maharjan, and Y. Zhang, "Joint load balancing and offloading in vehicular edge computing and networks," *IEEE Internet of Things Journal*, vol. 6, no. 3, pp. 4377–4387, 2019.

[15] Y. Wang, P. Lang, D. Tian, J. Zhou, X. Duan, Y. Cao, and D. Zhao, "A game-based computation offloading method in vehicular multiaccess edge computing networks," *IEEE Internet of Things Journal*, vol. 7, no. 6, pp. 4987–4996, 2020.

[16] P. Liu, J. Li, and Z. Sun, "Matching-based task offloading for vehicular edge computing," *IEEE Access*, vol. 7, pp. 27 628–27 640, 2019.

[17] J. Feng, Z. Liu, C. Wu, and Y. Ji, "Mobile edge computing for the internet of vehicles: Offloading framework and job scheduling," *IEEE Vehicular Technology Magazine*, vol. 14, no. 1, pp. 28–36, 2019.

[18] Z. Jiang, S. Zhou, X. Guo, and Z. Niu, "Task replication for deadline-constrained vehicular cloud computing: Optimal policy, performance analysis, and implications on road traffic," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 93–107, 2018.

[19] Q. Hu, C. Wu, X. Zhao, X. Chen, Y. Ji, and T. Yoshinaga, "Vehicular multi-access edge computing with licensed sub-6 ghz, ieee 802.11p and mmwave," *IEEE Access*, vol. 6, pp. 1995–2004, 2018.

[20] H. Peng and X. Shen, "Deep reinforcement learning based resource management for multi-access edge computing in vehicular networks," *IEEE Transactions on Network Science and Engineering*, vol. 7, no. 4, pp. 2416–2428, 2020.

[21] H. Sami, A. Mourad, and W. El-Hajj, "Vehicular-obus-as-on-demand-fogs: Resource and context aware deployment of containerized micro-services," *IEEE/ACM Transactions on Networking*, vol. 28, no. 02, pp. 778–790, mar 2020.

[22] L. Li, W. Chu, J. Langford, and R. E. Schapire, "A contextual-bandit approach to personalized news article recommendation," in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 661–670.

[23] C. Zhang, H. Wang, S. Yang, and Y. Gao, "A contextual bandit approach to personalized online recommendation via sparse interactions," in *Advances in Knowledge Discovery and Data Mining*, Q. Yang, Z.-H. Zhou, Z. Gong, M.-L. Zhang, and S.-J. Huang, Eds. Cham: Springer International Publishing, 2019, pp. 394–406.

[24] A. Krishnamurthy, J. Langford, A. Slivkins, and C. Zhang, "Contextual bandits with continuous actions: Smoothing, zooming, and adapting," in *Proceedings of the Thirty-Second Conference on Learning Theory*, ser. Proceedings of Machine Learning Research, A. Beygelzimer and D. Hsu, Eds., vol. 99. Phoenix, USA: PMLR, 25–28 Jun 2019, pp. 2025–2027.

[25] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018.

[26] C. Szepesvári, "Algorithms for reinforcement learning," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 4, no. 1, pp. 1–103, 2010.

[27] L. Tang, Y. Jiang, L. Li, and T. Li, "Ensemble contextual bandits for personalized recommendation," in *Proceedings of the 8th ACM Conference on Recommender Systems*, ser. RecSys '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 73–80.

[28] R. Cañamares, M. Redondo, and P. Castells, "Multi-armed recommender system bandit ensembles," in *Proceedings of the 13th ACM Conference on Recommender Systems*, ser. RecSys '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 432–436.

[29] G. James, D. Witten, T. Hastie, and R. Tibshirani, *Linear Regression*. New York, NY: Springer New York, 2013, pp. 59–126.

[30] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*, Y. Lechevallier and G. Saporta, Eds. Heidelberg: Physica-Verlag HD, 2010, pp. 177–186.

[31] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update," *SIGKDD Explor. Newsl.*, vol. 11, no. 1, p. 10–18, Nov. 2009.

[32] C. Sonmez, A. Ozgovde, and C. Ersoy, "Edgecloudsim: An environment for performance evaluation of edge computing systems," *Transactions on Emerging Telecommunications Technologies*, vol. 29, no. 11, p. e3493, 2018.

[33] C. Sonmez, A. Ozgovde, and C. Ersoy, "Fuzzy workload orchestration for edge computing," *IEEE Transactions on Network and Service Management*, vol. 16, no. 2, pp. 769–782, 2019.