November 21, 2018

REINFORCEMENT LEARNING BASED RECOMMENDATION SYSTEM FOR SOFTWARE UPGRADES

Parth Gaggar

Mansi Goel

Chandra Kamatam

Srinivasa Rao Aravilli

REINFORCEMENT LEARNING BASED RECOMMENDATION SYSTEM FOR SOFTWARE UPGRADES

AUTHORS:

Parth Gaggar
Mansi Goel
Chandra Kamatam
Srinivasa Rao Aravilli

ABSTRACT

A software recommendation system that uses Reinforcement Learning to recommend the most suitable upgradable software versions to a customer. The software recommendation system takes into consideration the user's feedback and applies that learning to unseen cases, resulting in a better customer experience.

DETAILED DESCRIPTION

Many of the most widespread and complex network issues are caused by devices using the wrong software. For example, some estimates indicate that more than twenty-five (25) percent of service requests submitted by customers are found to be caused by network devices with the wrong software version. These problems can also be quite serious, potentially exposing customers to software bugs, security risks, and high severity incidents. Enormous amounts of time and resources are spent resolving these service requests. Additionally, identifying the best software version for a given device is not always straightforward and different tools may provide different recommendations. This can confuse customers and reduce their confidence in the ability of vendors to accurately address their problems. As such, there is a need of a recommendation system that the user can upgrade to, without facing issues or downtime.

Presented herein is a software recommendation system that uses a sequential decision based process and Reinforcement Learning to recommend the most suitable upgradable software versions to a customer. The software recommendation system learns from user feedback and accommodates that learning into its own decision process, to allow for improved recommendations for unseen cases. For example, the software recommendation system is implemented, in part, in the form of a Markov Decision Process, which uses Reinforcement Learning to incorporate the user feedback in the form of rewards and policies. In certain examples, Q-learning with a function approximation technique, sometimes referred to as a Double Deep Q Network, is used to train the software recommendation system.

Recommendation Process

There are multiple factors that are considered while making a software recommendation, including:

1. *Currently running software* - Currently running software can provide a variety of information with respect to the device, namely the type of Image, configuration (running and startup configuration), device information, and supported software for that device. This information is fetched from the Service Request (SR) which is filed against the device for a support engineer to work on.
2. *Impacting Bugs* - These are the bugs associated with the request. These bugs are filed against the version running on the device, and it is expected that the recommended version should have all or most of those bugs fixed.
3. *Security Vulnerabilities* - It is important that the recommend software has no or minimal exposure to security vulnerabilities. Thus, the software vulnerabilities are checked with respect to the current software and a software that is least vulnerable is recommended.
4. *End of Life* - Every software has a defined lifecycle, which means it will go end of support at a particular date. That means the system should recommend software which is likely to last long and has the end of support date further into the future.
5. *Current Deployment* - A software which is currently deployed with a large number of customers is likely to be a preferred software. Thus, the system looks to deploy such software, which has a higher presence across the existing customers.

To recommend a software, the recommendation system first obtains the device details from the Service Request (SR) and then takes the current software version as the starting point. The recommendation system then identifies the candidate software versions based on where the bugs are fixed, and also the versions which have security vulnerability fixes. The recommendation system iterates over the same obtained software versions and identifies open security vulnerabilities, and again moves towards the versions containing the fix for those issues (Figure 1). As a result, the recommendation system is searching in a directed graph and the structure makes it inefficient to search across all possible states, in order to reach the final state. The user's past feedback is retrieved which acts as the correct recommendation (V_6 in Figure 1) and is incorporated within the algorithm to make it learn from the feedback.

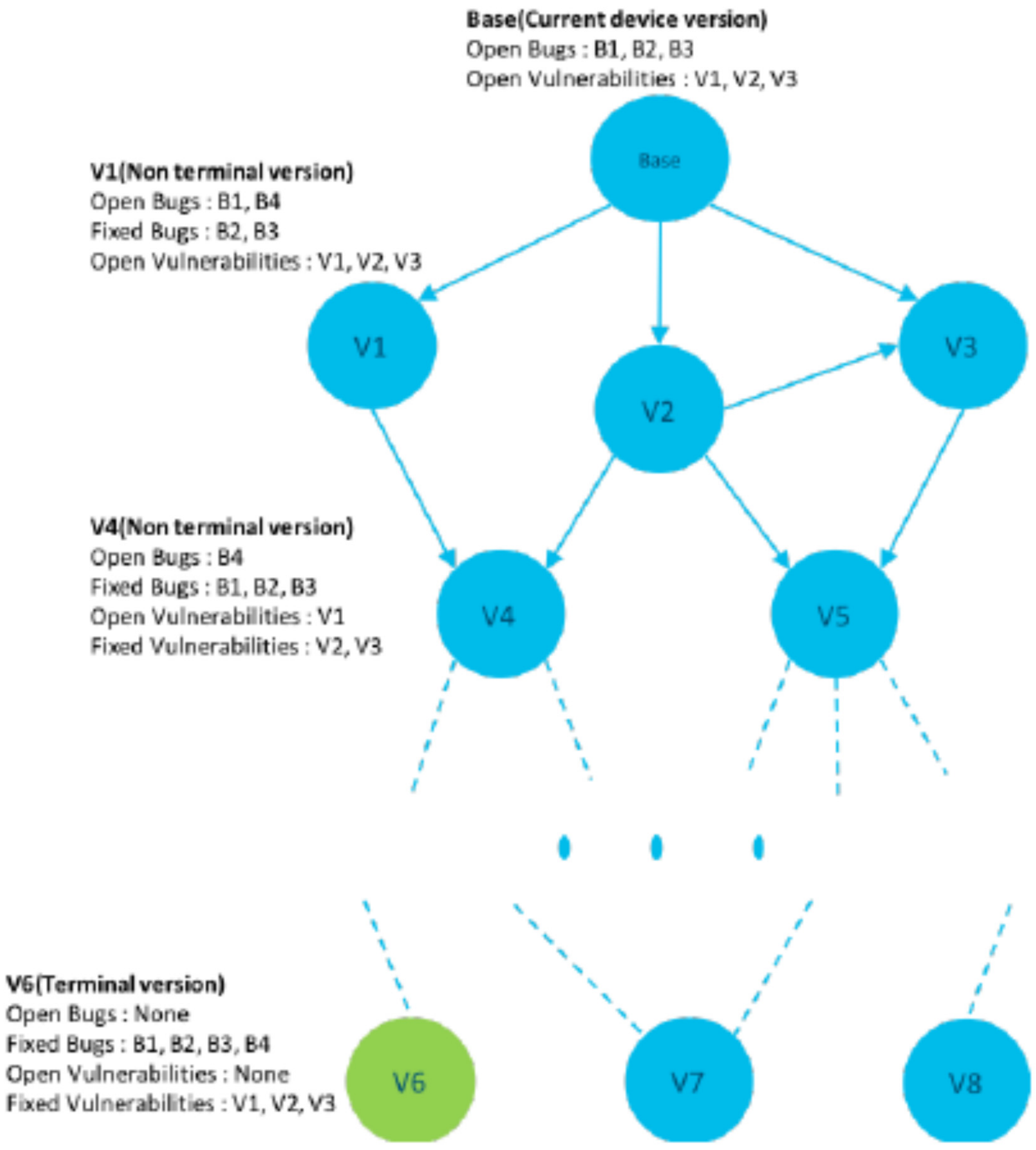
Base(Current device version)
Open Bugs : B1, B2, B3
Open Vulnerabilities : V1, V2, V3
Base
V1(Non terminal version)
Open Bugs : B1, B4
Fixed Bugs : B2, B3
Open Vulnerabilities : V1, V2, V3
V1
V2
V3
V4(Non terminal version)
Open Bugs : B4
Fixed Bugs : B1, B2, B3
Open Vulnerabilities : V1
Fixed Vulnerabilities : V2, V3
V4
V5
V6(Terminal version)
Open Bugs : None
Fixed Bugs : B1, B2, B3, B4
Open Vulnerabilities : None
Fixed Vulnerabilities : V1, V2, V3
V6
V7
V8

Figure 1- Recommendation as a Markov Decision Process.

In Figure 1, the version on the top (Base) is the initial version. The version at the bottom marked in green (V6) is suggested as the terminal version which is obtained as the recommendation through user feedback.

Reinforcement Learning

In general, there are two reasons for choosing Reinforcement Learning (RL) for this type of problem:

1. The main rationale is that finding a suitable software version is a sequential decision-making process which can be modelled as a Markov Decision Process (MDP). At every time step, given a software version V_1 and the list of versions (V_2 -> V_N) which contain the fixes of security vulnerabilities and bugs for V_1. The task is to find an optimal policy which decides which software version to choose among the list of software versions. The purpose of RL is to solve this MDP and come up with the optimal policy.
2. Also, RL enables to take into consideration the feedback given by the user in the past. If the user had provided a negative feedback for the given recommendation, the model will incorporate the negative feedback and try to improve the recommendations by penalizing the inappropriate version and the path that led to the inappropriate version.

Methodology

To apply reinforcement learning, it is appropriate to formulate the problem as a Markov Decision Process (MDP). An MDP is by definition a five-tuple: $<S,A,R,T,\gamma>$, where S is a set of states, A is a set of actions, R is a reward function that assigns a real value to each state/action pair, and T is the state-transition function, which provides the probability of a transition to a state, given an action taken from that state and γ is the discount factor that allows trade off b/w present and future rewards.

Here, each software version is represented as a state in the MDP. However, to preserve the Markovian property, the information that is needed at every stage is also folded in to determine the next state. As a result, each state has sufficient information to determine a stationary policy for that state.

Hence, the description of the MDP with respect to the application is as follows:

- States: Each state is defined by the Software, Security Vulnerabilities, Bugs, Publish Date, and Deployment related information for that software.
- Actions: Action from each state is defined as the next state to which the agent decides to move to.
- Transition Function: Transitions are taken as deterministic, and thus the actions are indicative of the next state.
- Reward: Rewards for any non-terminal state transition is taken to be slightly negative to incentivize shorter paths. Depending upon the user feedback, if the feedback is positive (i.e., one of the recommendations is correct), there is a large positive reward given for the terminal state transition, and if the feedback is negative (i.e., all recommendations are incorrect), there is a large negative reward given for the corresponding terminal state transitions.
- Discount Factor: This represents the horizon of the MDP. This value was taken on the higher side so that the agents move towards the best global target, and also avoids loops and longer paths.

Q learning with Function Approximation

Q Learning is an off-policy, model-free algorithm for solving MDPs. Due to its model-free nature, it does not assume prior knowledge of the transition probabilities and the rewards and explores the environment to learn them. The standard Q learning implementation requires explicit representation of the state space, which is often not feasible. There may be new additions to the state space by means of new software, and the algorithm needs to handle such cases. This would mean the creation of an entirely new MDP for every case. To avoid this, and generalize over these cases, a Double Deep Q Network is used, which is a Deep Learning based Function Approximation based method that uses samples from its past experience to learn the correct policy by estimating Q values for the state action pair. This is commonly referred to as Experience Replay.

Training

Figure 2, below, explains the end to end procedure followed for training the data. The data is taken from the customer network, and used to identify the current version, and the bugs and vulnerabilities that it is exposed to. Then subsequent versions are identified by looking up fixes for the bugs and vulnerabilities, and the MDP is created. Rewards are assigned based on customer feedback, and the network is trained with Double Deep Q Networks.

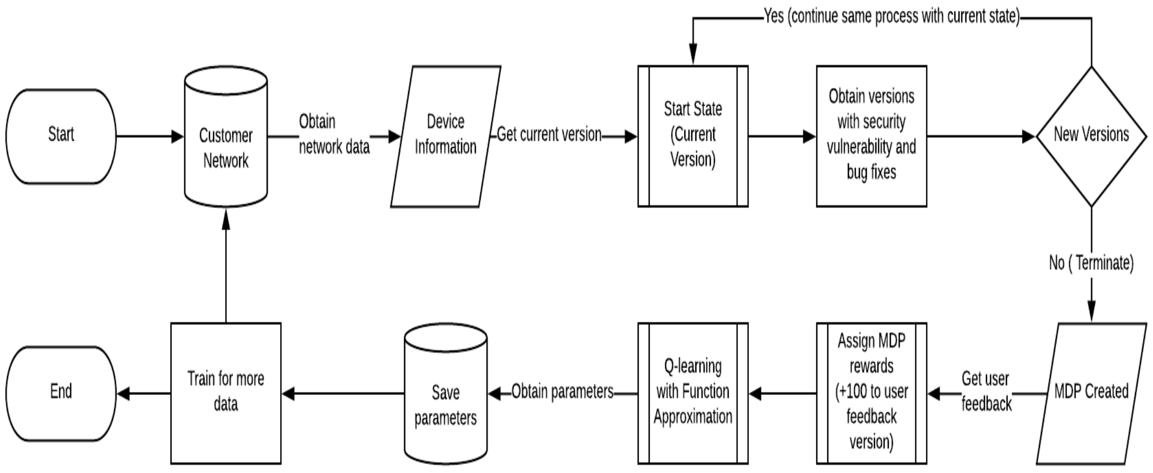
Start
Customer Network
Obtain network data
Device Information
Get current version
Start State (Current Version)
Obtain versions with security vulnerability and bug fixes
New Versions
Yes (continue same process with current state)
No (Terminate)
MDP Created
Get user feedback
Assign MDP rewards (+100 to user feedback version)
Q-learning with Function Approximation
Obtain parameters
Save parameters
Train for more data
End

Figure 2. Training flow. This includes fetching the data from customer network, creating an MDP, and training a double DQN network for identifying the correct recommendation.

Testing

Figure 3, below, illustrates the end to end procedure followed for testing the data. The starting point here is taken as the current version from the customer network as well, and the fixed versions are identified by the bug and vulnerability fixes. After that, the Deep Q learner is used to identify the state with the highest Q value, then that version is taken as the next base version, and this process is repeated until a terminal state is reached.

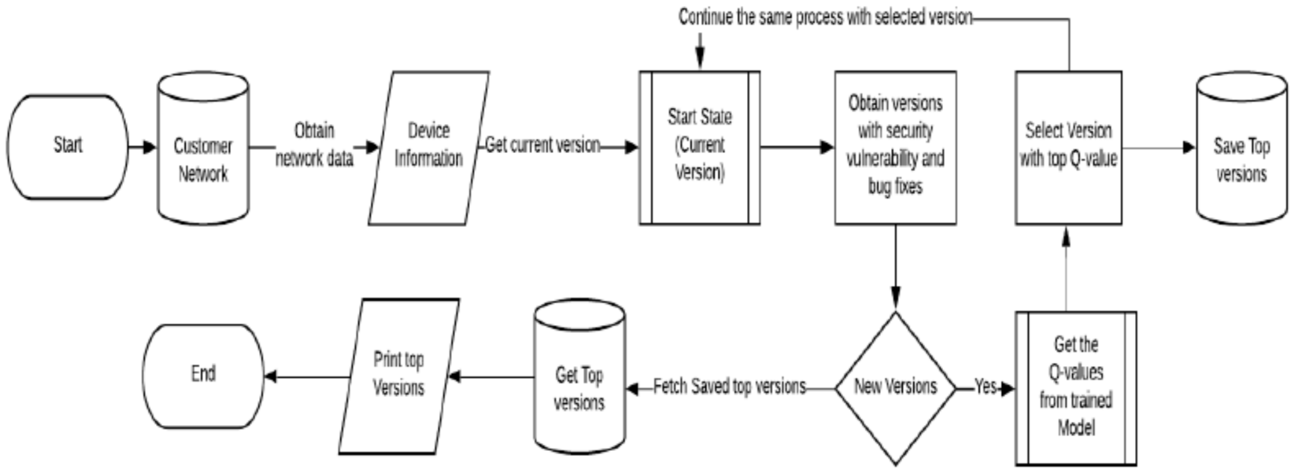
Continue the same process with selected version
Start
Customer Network
Obtain network data
Device Information
Get current version
Start State (Current Version)
Obtain versions with security vulnerability and bug fixes
Select Version with top Q-value
Save Top versions
End
Print top Versions
Get Top versions
Fetch Saved top versions
New Versions
Yes
Get the Q-values from trained Model

Figure 3. Testing flow. This includes fetching the data from customer network, identifying next versions, taking the ones with highest Q values, until the terminal version is reached

Technical Benefits

There are two key benefits of this approach:

1. Designing the MDP based on user feedback means that the recommendation engine learns from the user. This is really important because the objective of any recommendation engine is to predict results based on how a human would, given the exact same choices.
2. Using the Q values to decide which version/path to select compared to going down all paths saves a lot of time, by avoiding unnecessary deep searching. This in turn allows for faster, near real-time recommendations.

Business Benefit

This technique presented herein allow for a way to transform a software recommendation into a sequential decision based process. This is very useful for products that require calculated sequential decisions to be made, as they can be modelled as Markov Decision Processes. The technique presented herein allows for a way to account for user feedback as part of the process, and hence can be useful in any case where the system must continuously learn from the user.