Date: 06/20/2016

Algorithm document for Machine Learning on Caseta Smart Bridge:

This document contains information about the structure of the algorithm and the Design of the code along with high level ideas of how the algorithm works.

Classes:

EventDetail: Contains methods to get, set and parse the details associated with the event such as zoneId, time stamp, date, level and zoneSource. All these fileds are parsed and converted to integers for storing so that the algorithm can efficiently use these fields for computation.

EventCompare: This class has some helper methods to compare if two events are in the same window or if they have similar level.

ValueMaintainer: This class contains all the variables which are decided upon for the algorithm. These variables include recording intervals, threshold values, number of weeks, suggestion window and other such variables.

Mediator: This class contains methods to interact with all other modules of the code. (Goes with the name as well). It will take incoming in-coming json message (which will have the event details). This data will be converted to an EventDetail Object by the mediator and then sent to the list manager which will dynamically manage a list of events. The mediator will be responsible for calling the run_scene_suggestion() method in the core_algorithm.py.

__init__ : This script just inflates the data and calls the appropriate mediator methods. It is just a script to run the application, since for now, no json data is being received from the bridge. This file simulates the user behaviour.

CoreAlgorithm: This class contains most of the methods which generate recommendations from the event detail objects.

Working and Flow of the Algorithm:

run_scene_suggestion(event_details_list): This method is called by the mediator with all the previously collected data in form of a list.

1. **For** all the relevant days (on which we know that events have occurred) , **for** all the relevant time-stamps (time stamps at which the events have occurred and the timestamps within its window.) These lists of relevant days and time stamps are maintained dynamically by the EventDetail Class. This ensures that no additional filtering it needed since everytime an event occurs its day and time will be put in a set maintained by EventDetail.
    1. Get all the events associated with this time stamp (based on the window) and day from this list. Current window size is 5, which implies it will consider events at x -5 and x + 5 times for suggestion at x. This technically makes it a 15 min. Eg: Events at 10am take into consideration events at 9.55 am and 10.05 am. But events from 9.53am onwards are rounded to 9.55 while events before 10.07 am are rounded up to 10.05. This makes it a 15 min window which intuitively, should be good for practical considerations.
    2. If there are any associated events, go to step 3. Else increment values in the inner for loop and repeat from step 1.

3. This list might contain information about different zones (since 2 zones can be switched on at the same time or in the same window. WE WANT TO CONSIDER THESE GROUPINGS AS WELL). This implies, we will have to initially segregate this list based on the zones which will help us to check repeatability for a particular zone.
   This is done by constructing a map of zone_vs_(list of events associated with that zone at that time on a particular day.)
   Zone 1 → event_list1
   Zone 2 → event_list2
   Zone_3 → event_list3
   ….
   Now what do we do? Recommend the user? **NO**
   Now comes the main part for checking if the user actually **repeats** this action. If yes, then we can **THINK** about recommending it.
   So how do we check for **repeatability** given we know the total number of weeks (weeks for which we have been collecting data) and some percentage of repeatability (maybe 70% - 80%) considering that we may want to recommend something that the user does 7 to 8 times from 10?
   This is done by passing the above map to **remove_outliers_by_checking_repeatability()** method.

   Every **value** of the above map contains all the previous events related to a certain zone on a particular day in a particular time window. So once we can go through the list associated with every zone and just check that the total events in a particular list are within the threshold range of total number of weeks to check for repeatability? **NO. Not that simple.**
   What we need to essentially consider is the number of events with **unique dates** associated with event zone.
   What the algorithm does is: It takes the list associated with every zone, constructs a map/ dict of **unique_date VS events** for that zone.
   For every event list which is associated with a zone in the above map,

   Date1 → events .. (if there are multiple events for the same zone in the same time stamp window)
   Date2 → events ..
   Date3 → events ..
   …..

   If the number of keys (unique dates) in the map are in the threshold range with number of weeks elapsed, the event has occurred almost every week. **Right ? Yeah!**

   Yet, can we be certain about the event though? NO. This is because we are sure that the event occurs every week. But is the user usually setting the level of the zone to a particular level or does the user switch it on in the 1st week and switch it off in the 2nd? That's what we want to know as well. Also what if there are multiple events in

the same window for a date. Those events can be random since one can be playing with the Pico randomly generating events in a 5 min window. **WE DON'T WANT THIS. HOW WOULD WE REMOVE THEM?**

To remove them all the outliers and random events, we will go through events associated with every date in the above map. These are events which occurred on a unique date, unique time window and for a particular zone. Now, of there is more than one event, we take the average of all the levels and compare it to the individual levels. If they differ by more than threshold that we want to consider (can be easily changed by tweaking the ValueMaintainer), we decide that the events associated with the particular zone we are considering is random.

Eg: zone1 → event1, event2, event3 [2 week data]
Zone2→ event4, event5

Then we construct
Date1→ event1, event2 (assume event1, event2 are in the same window)
Date2→ event3

1st step now will be compare average(event1, event2) to event1 and event2. If they aren't in range zone1→ event1, event2, event3 from the map is an outlier. This means there are events that occur with zone1 repeatedly but the levels of these events is not consistent.

If x = average(event1, event2) is within the threshold range of event 1 and event2, we calculate average(x, event3) and compare them with x and event3. This makes sure that the event occurs with some consistent level before we can think about recommending it to the user.

If both these tests pass, zone1 → level(computed by kernel regression by taking distances into account) is one of the recommendations. If the tests pass for zone2 as well, zone1 and zone2 form a group and are recommended.

Woah. Very confusing! In plain English, after making sure that an event associated with a particular zone is repeated every week in the same time window, we just made sure that the levels associated with that event are within the threshold range. ☺

4. Now what we have is the same original map of zone vs events, but the zones which were outliers are removed from the map. If there are any zones remaining, we will add this map to recommendations and compute the suggested level for every zone by using weighted average Kernel Regression. This is because we want our suggestions to be mostly resembling the event that occurred almost at the same time and look less like those who were a part of window, but away☺. **add_recommendations** will make a recommendation object and store this in a list maintained by recommendations. There is also a trust factor associated with every zone which takes into consideration the number of previous events which have been taken into consideration for making decision about this recommendation. Along with this, the trust factor also takes into account the time stamps of the events which went

into making this decision for advanced filtering and better "Time accurate" recommendation. This is because assume the following scenario:
1st Monday 9 am → 100 (level)
2nd Monday 9 am → 100
When we are generating recommendations, at time 8.55 am on Monday, 2 events will be considered. The same thing will happen for generating a recommendation for 9 am on Monday. The only way to differentiate and suggest the user to switch on the lights at 9 am will be to take into consideration the time stamp of previous events that have been taken into consideration. This is how we associate an intuitive trust factor with every recommendation.

The algorithm quickly computes this list by going through all the days and logical time stamps. Now comes what do we actually want to recommend to the user.

Recommendations: This class maintains a list of recommendations generated by the core-algorithm. Do we need to really recommend everything to the user? NO. While the recommendation objects are generated by the user, they are filtered ON THE GO to ensure efficiency.

This is done in the following way:

While a new recommendation is generated, we will go through the list of previously finalized recommendations to find if there is a recommendation with "**similar zone vs level**" and "**similar time stamp**" (This will be efficient since the list will never be large as we won't add redundant recommendations to it).

If such a recommendation is found, we will check if the newly generated recommendation is on the **same day** or not. If yes, we will see if the **trust factor** of this newly generated recommendation is more. If yes, we will replace the old recommendation with the new one on the fly. If the trust factor is less, the newly created recommendation is redundant.

If the newly created recommendation is **not on the same day** as the similar recommendation found, we will append the day of this recommendation to a "**day_list**" of the similar recommendation which has previously been stored. In this manner, the recommendations will get merged (and we can detect clusters in day usage patterns as well)

Visualizations:

The application is designed to plot graphs for better understanding of what is happening.

Event plots:

The event graphs are plotted for a particular day(Monday or Tuesday …) and a particular zone. Eg. A graph will be plotted for all 4 Mondays, zone_1 and all 4 Mondays, zone_2 if we are considering a data for 1 month (4 weeks) and the data consists of 2 zones. All the 4 Mondays will be overlapped on the graph so that we can easily spot the places where the event is repeated.

Recommendation plots:

The recommendation plots are plotted for all the 7 days and available zones. They are only plotted if there is a recommendation which is generated. For example, if there is a recommendation which is generated

for zone_1 and zone_2 for Monday and Tuesday, there will be 2 plots one for Monday with zone_1 and zone_2 overlapped and one for Tuesday with zone_1 and zone_2 overlapped.

For any doubts, corrections, optimizations feel free to ping me on Skype: **chshah.lutron** OR **chshah@lutron.com**