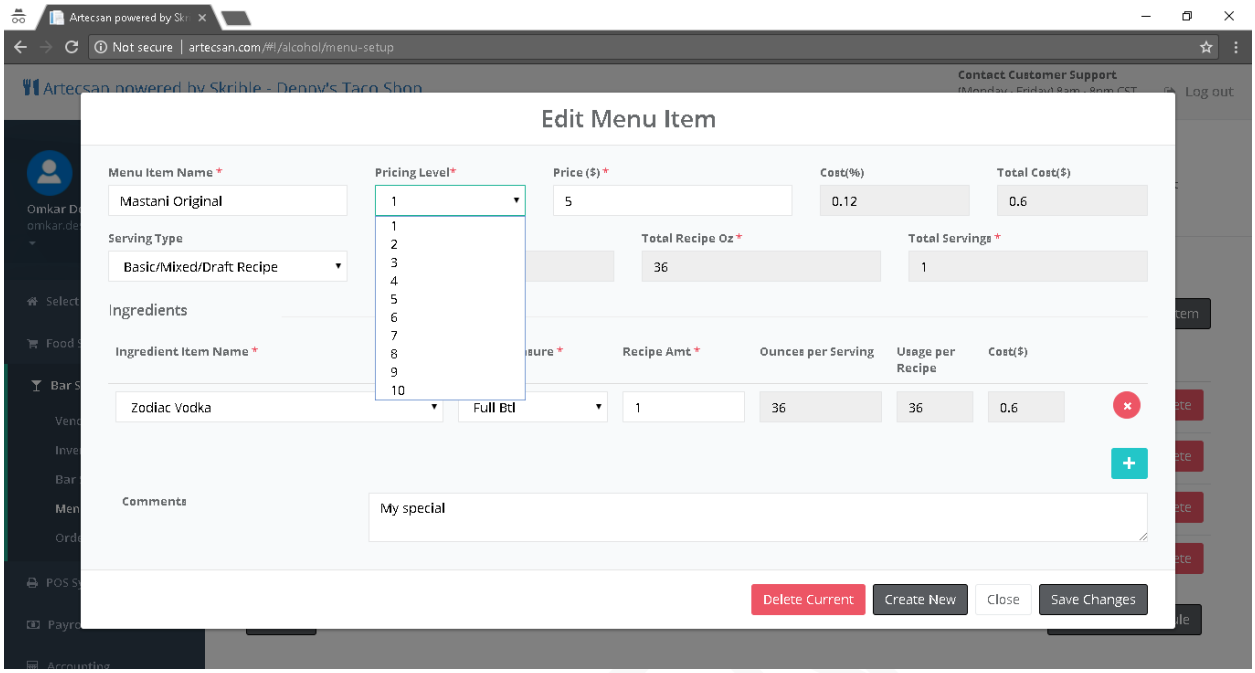
## Index

# 1. Pricing Level Mapping and Functionality:

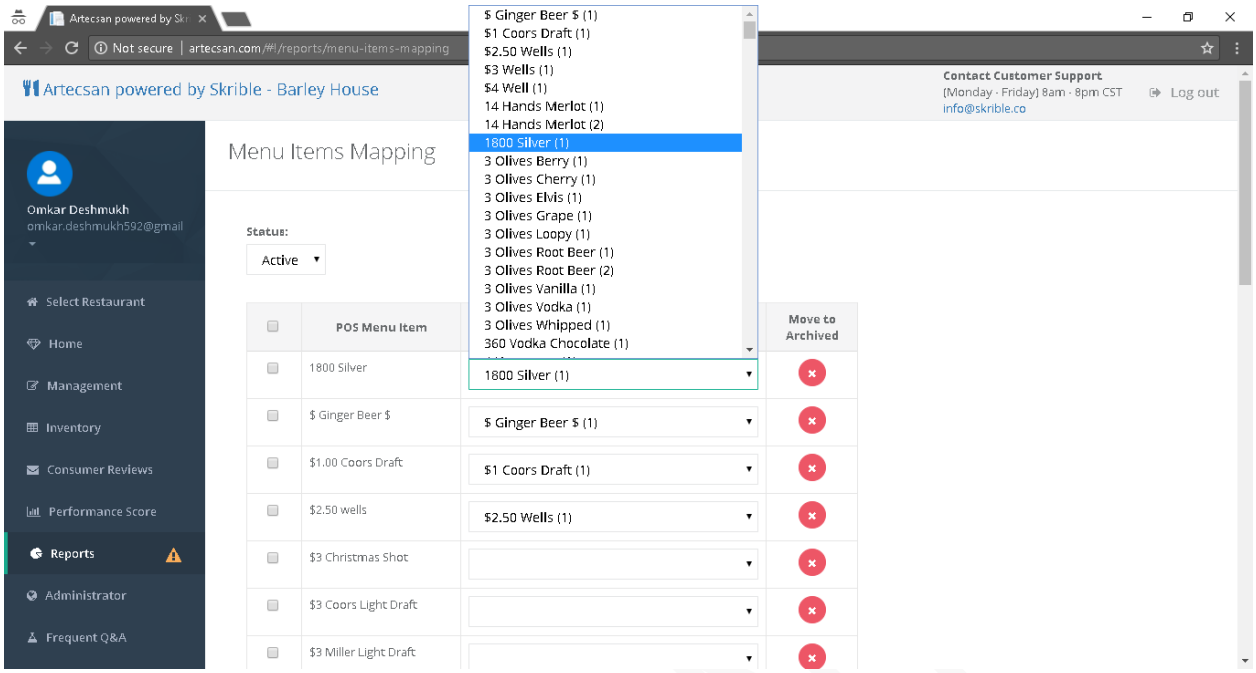



**Summary:**
Previously Artecsan was not built with an option or features that accommodated or used the pricing levels of a menu item. Each menu item has a pricing level associated with it ranging from 1 to 10. Thus the same menu item with different pricing level is considered to be unique; has a unique menu id, different recipe, different price, etc but the same menu item name like its other counterparts.

**What's already done:**
Added a pricing level field to the Edit Menu Item page.
Even if the menu items of different pricing levels have the same name, each one is considered as a unique item (having a unique menu id).
Consequently, the **as_restaurants_menu** table was given a new column 'pricing_level' to accommodate this change.
The uniqueness is now maintained based on menu item name + pricing level (a unique menu_id for each).

**Requirements:**
We need to update the mapping and data to chalk the data flow from the POS to various parts of the system and find out where a pricing level comes into play. If we have missed anything please discuss it with us as this section is very important.

Currently the Menu Items Mapping page shows all the POS menu items but not the pricing levels. In order to properly map the items we need to add the pricing levels for POS Menu items (but only for items that are sold or automatically mapped). We do not need to see unused or inactive pricing levels.



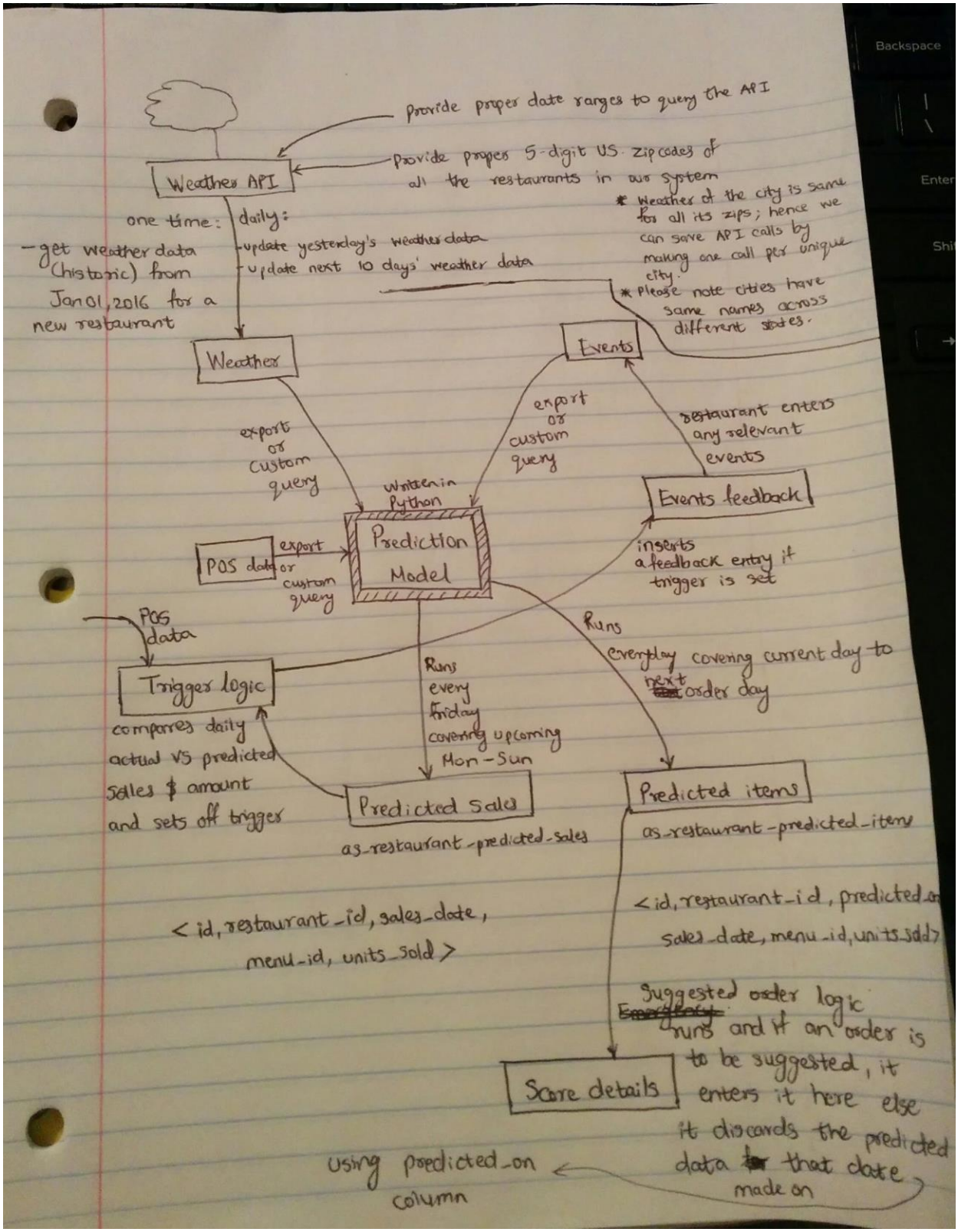

The Inventory Usage Report should also reflect the addition of pricing levels to the menu items. This means that the system should pull the menu items from the POS (with multiple pricing levels), then use the mapping to determine its ingredients, prices, etc (same as before). And then update the inventory usage report accordingly.

<u>Automated menu item mapping:</u>

Menu items automatically map whenever the user Creates, Saves or Updates a menu item. The system would look for identical name and pricing level.

---

## 2. Prediction Model:



**Summary:**
The prediction model will serve as the primary source of forecasting sales and suggested ordering which will drive other components of the system.
The model takes the POS data, weather data and events data as an input and produces prediction of units to be sold for every menu item for a particular date(s).

**What's already done:**
Prediction Model:
We have already implemented the model in Python.
The prediction model currently accepts sales data, weather data and event data by reading the exported CSV files present in the same folder as the source code. The queries for the exports are given in the 'more details' section.
The model processes the data, makes the prediction and outputs the results into a CSV file.
A few parameters need to be changed every time the model is to be run for different restaurants and for different date ranges.
The results of the prediction model were manually inserted into the DB to make it available to other parts of the system. The required tables namely, **as_restaurants_predicted_sales** and **as_restaurants_predicted_menu_items** that will store the results of the prediction are already created.

Weather data and weather API:
Currently we were fetching the weather data manually from the weather API and storing that data into the DB.
We have already created the tables that will store the weather data;
**as_weather_observation_dates** and **as_weather_observation_day.**

Events:
This table holds events data and we have already created and tested this table; **as_events.**

**Requirements:**
Prediction Model:
What we want to be done is to deploy the model implemented by us on our system (Artecsan) and allow Omkar to access the environment to develop and test the model in the future.

- The sales prediction model must automatically run for all the restaurants on Friday covering the dates of the next Monday to Sunday, the results of which must be stored in **as_restaurants_predicted_sales.**
- The program must automatically run for all restaurants daily, results of which must be stored in **as_restaurants_predicted_menu_items**.
- On a Friday the model can run just once and the same results can be stored in both the tables.

Weather data and weather API:

We require that the weather tables be populated daily (automatically using cron job) with the next 10 days forecast (including the current day) and as the day passes by, store the actual observed historical weather data for the previous day.

We also require that when the restaurant first signs up for an account, the program pulls and stores the weather data from Jan 01, 2016 based on the restaurant's zip code. If the data already exist for the specific zip code, the program should ignore the pull request.

We wanted to store the weather data of a particular city for a given date and thus made the following tables:
**as_weather_observation_dates** and **as_weather_observation_day**.
as_weather_observation_dates is basically a parent table that stores date and location (city & state) and has a unique dcs_id.
as_weather_observation_day points to dcs_id in as_weather_observation_dates and stores for a time slice of a day the weather data.

**Please note all dates are to be represented in YYYY-MM-DD format.**

We divided the day into time slices:
06:00 - 10:00, 10:00 - 14:00, 14:00 - 18:00, 18:00 - 22:00, 22:00 - 06:00 (next day)

We are using https://www.apixu.com as our primary source of weather data and a premium key with Gold plan ($29.95 a month) which gives us the feature to get 10 days of weather forecast data.
The key is: **d9d7ad5df78e45eba77154309171008**

The API call URLs for historic and forecast are available here https://www.apixu.com/api-explorer.aspx

Please provide the required dates and zip code of the restaurant to get the weather data. Cities will have same weather data for all its zips, hence avoid calling the weather API redundantly for the zips of the cities of which we already have the data for. This will significantly reduce our API call count and processing time.
We also may have to check our DB which contains zip, cities data. American zip codes are 5 digits, I am not sure why some zips in our DB are 3 or 4 digits which don't seem to work with the API or refer to completely different cities in different countries. Please have a look and fix / repopulate the required tables. For your reference: https://www.unitedstateszipcodes.org

The weather API gives datewise forecast and divides the day into 24 hour slots. As per our above requirement of time slices, aggregation is needed.
With respect to the weather API time slots:

| | Weather API | Temperature (f) | Wind (mph) | Precipitation (in) | Cloud |
|---|---|---|---|---|---|

| | | | | | |
|---|---|---|---|---|---|
| 06:00 - 10:00 | 06:00 - 09:00 | min | max | max | max |
| 10:00 - 14:00 | 10:00 - 13:00 | avg | max | max | max |
| 14:00 - 18:00 | 14:00 - 17:00 | max | max | max | max |
| 18:00 - 22:00 | 18:00 - 21:00 | avg | max | max | max |
| 22:00 - 06:00 (next day) | 22:00 - 05:00 (next day) | min | max | max | max |

Till now we were manually fetching the data and using a naive node js program to extract data to store in the tables. Please find the source code here for reference and the source data file here. Of course this won't be completely required for the cron job, I just gave you the source code for reference.
Insertion into as_weather_observation_dates is pretty straightforward. Insertion into as_weather_observation_day requires to properly store the dcs_id as the foreign key by matching the date, city and state in the parent table.

Events:
Events feedback form is a form in Artecsan which is automatically emailed to the restaurant whenever we see an unusually high spike in the restaurant sales. That email will contain a message indicating that our prediction model was off on a specific date and will contain a link directing the user to the events feedback page. The data from the Feedback form is fed to the "as_events table". Hence everyday, a trigger logic runs after the POS data for the previous day comes through. The logic compares the dollar value of actual sales and predicted sales (from **as_restaurants_predicted_sales** table) for yesterday using a simple SUM and GROUP BY query. The trigger logic is described in the 'more details' section. If the trigger is set off, a new entry is inserted into the **as_events_feedback_stack** with the required data.

**More details:**
Prediction Model:
Right now there seems to be one issue with **as_pos_sold_menu_items**, some records have menu_item_id but no menu_item_name.
Also items with similar names but different pricing levels have the same id (the id of the item with price_number = 1).
I believe once you fix and incorporate the pricing level design component, the queries will properly insert records without the above issue(s).

To fix the above issue, we created a copy of **as_pos_sold_menu_items** namely, **as_pos_sold_menu_items_new** and used the following:

```
UPDATE `as_pos_sold_menu_items_new` up
```

```
SET    up.`menu_item_name` = (SELECT m.menu_item_name
                              FROM   as_restaurants_menus m
                              WHERE  m.restaurant_id = up.restaurant_id
                                     AND m.id = up.menu_item_id)
WHERE  up.`menu_item_id` IS NOT NULL
       AND up.`menu_item_name` IS NULL;


UPDATE `as_pos_sold_menu_items_new` up
SET    up.`menu_item_id` = (SELECT m.id
                            FROM   as_restaurants_menus m
                            WHERE  m.restaurant_id = up.restaurant_id
                                   AND m.menu_item_name =
up.menu_item_name
                                   AND up.price_number = m.pricing_level)
WHERE  up.`menu_item_name` IS NOT NULL;
```

Assuming we have the weather table, event table and pos sold menu items tables all ready and error free, the Python program reads 3 csv files (after deploying this program online, you have the choice of either exporting the SQL query data as csv for the program to read or directly use dynamic queries to read data from the DB and instantiate pandas dataframe objects in the program).

The first csv is the **as_weather.csv** and it uses the following query to fetch data.

```
SELECT asdates.city AS City,
       asdates.state AS State,
       asdates.observation_date AS Observation_date,
       Dayname(asdates.observation_date) AS Day,
       Max(asday.temperature_f)          AS Temperature_f,
       Max(asday.wind_mph)               AS Wind_mph,
       Max(asday.precipitation_in)       AS Precipitation_in,
       Max(asday.cloud)                  AS Cloud
FROM   `as_weather_observation_dates` AS asdates
       INNER JOIN `as_weather_observation_day` AS asday
               ON asdates.dcs_id = asday.dcs_id
GROUP  BY asdates.dcs_id
ORDER  BY asdates.observation_date;
```

The second csv is the **as_events.csv** and it uses the following query to fetch data.

```
SELECT eve.date          AS Event_date,
       Dayname(eve.date) AS Day,
       eve.name          AS Event_name,
       eve.type          AS Event_type,
       eve.location_type AS Event_location_type,
```

```
       eve.location      AS Event_location
FROM   `as_events` eve
ORDER  BY event_date;
```

The third csv is the **restaurant_name+restaurant_id_master.csv** and it uses the following query to fetch data. Please note the dynamic parameter of this query will be the restaurant id found before the GROUP BY. Also pos_sold_menu_items_new will change to pos_sold_menu_items after the pricing levels are fixed.

```
SELECT `as_restaurants`.`restaurant_name` AS Restaurant_Name,
       c.city_name,
       c.state,
       Cast(start_date AS date)           AS Start_date,
       Dayname(start_date)                AS Day,
       report_group                       AS Category,
       name                               AS Menu_name,
       menu_item_id                       AS Menu_Item_ID,
       price_number                       AS Pricing_Level,
       price                              AS Price,
       units_sold,
       sales,
       Max(tod.temperature_f)             AS Temperature_f,
       Max(tod.wind_mph)                  AS Wind_mph,
       Max(tod.precipitation_in)          AS Precipitation_in,
       Max(tod.cloud)                     AS Cloud
FROM   as_rb_city c
       INNER JOIN `as_restaurants`
               ON as_restaurants.city_geoname_id = c.geoname_id
       INNER JOIN as_pos_sold_menu_items_new
               ON `as_restaurants`.id =
as_pos_sold_menu_items_new.restaurant_id
       INNER JOIN as_weather_observation_dates dcs
               ON start_date = dcs.observation_date
       INNER JOIN as_weather_observation_day tod
               ON dcs.dcs_id = tod.dcs_id
WHERE  c.city_name = dcs.city
       AND c.state = dcs.state
       AND units_sold > 0
       AND menu_item_id IS NOT NULL
       AND report_group IN ( 'Beer', 'Bottle Beer', 'Cocktail', 'Draft
Beer',
                             'Happy Bar', 'Liquor', 'Wine' )
       AND restaurant_id = 326
GROUP  BY start_date,
          menu_item_id,
          pricing_level
ORDER  BY start_date DESC;
```

If the program can read the above three files or data from the given queries it can run properly.

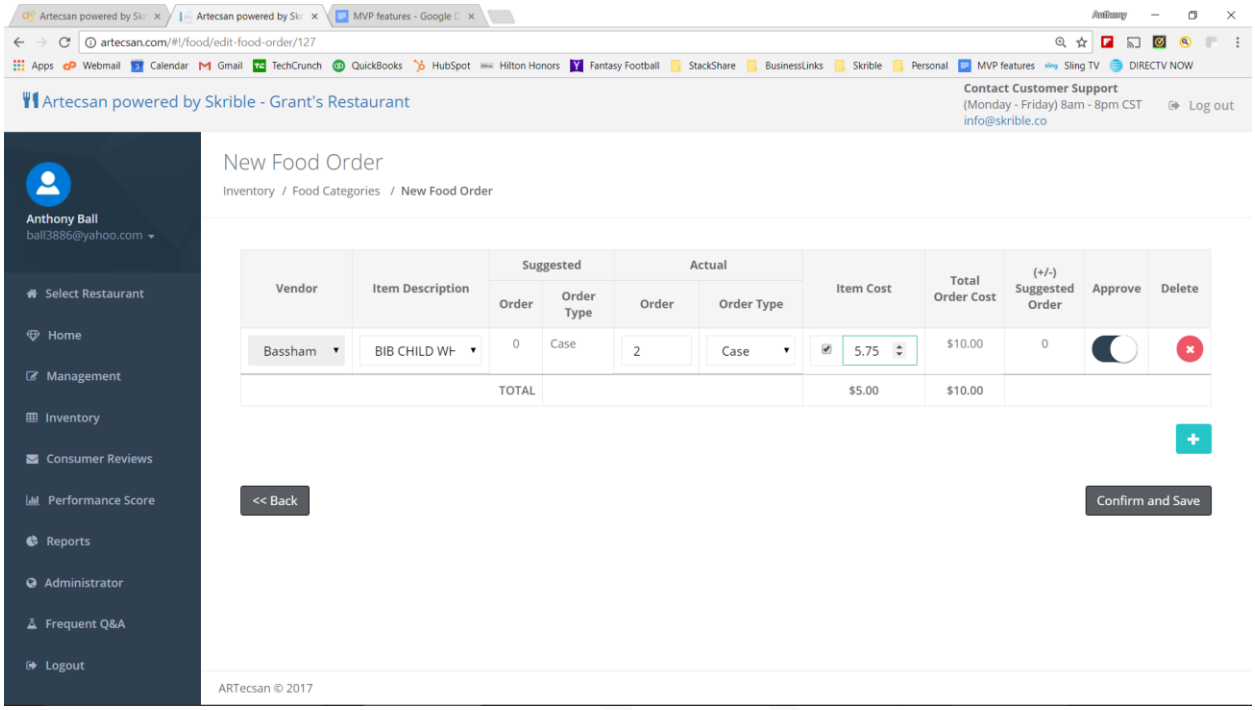
The next steps are transforming the data, training the model and predicting. The program outputs the prediction in the form of a csv named **restaurant_name+restaurant_id_results.csv**.

Trigger Logic:
if actual sales is less than $2000 and actual sales is greater than forecasted sales by 500%
if actual sales ranges between $2000 - $4000 and actual sales is greater than forecasted sales by 300%
if actual sales ranges between $4000 - $10,000 and actual sales is greater than forecasted sales by 175%
if actual sales ranges between $10,000 - $50,000 and actual sales is greater than forecasted sales by 150%
if actual sales is greater than $50,000 and actual sales is greater than forecasted sales 130%

If any one of the above is true, an email is automatically sent and a new entry in **as_events_feedback_stack** is inserted.

---

## 3. Suggested Ordering:




We want the prediction from the model to be converted to a suggested order for the restaurant and populate the Suggested Order column of the New Alcohol Order page automatically daily (if a suggested order is required).

Unapproved suggested orders items will be replaced if not used prior to the next day. Predictive ordering runs daily so it should take care of replacing the previous days order to include zeros however an suggested order of less than 1 item should not show up in the suggested order section.

Once an Order is received and confirmed that order should no longer be shown here.

The following example query converts the menu items quantities into total inventory items quantities. We require that once the suggested order logic runs and detects a need for a suggested order, the data from **as_restaurants_predicted_menu_items** will be converted using a query like the one below into its constituent inventory items in ounces and inserted into **as_restaurants_alcohol_performance_score_details.**

```
SELECT ps.restaurant_id,
       ps.sales_date,
       Sum(ps.sales_predicted)                                    AS
       total_sales_predicted,
       v.item_name,
       v.content_weight,
       Sum(ps.sales_predicted * mi.oz_per_serving / v.content_weight) AS
       units_consumed
```

```
FROM    `as_restaurants_predicted_menu_items` ps
        INNER JOIN as_restaurants_menus m
                ON ps.menu_id = m.id
                   AND ps.restaurant_id = m.restaurant_id
        INNER JOIN as_restaurants_menu_items mi
                ON m.id = mi.menu_id
        INNER JOIN as_vendor_sku v
                ON mi.vendor_sku_id = v.id
                   AND m.restaurant_id = v.restaurant_id
GROUP   BY m.restaurant_id,
           v.id;
```

Thus the UI will display the suggested order from **as_restaurants_alcohol_performance_score_details** table every day.
There will be days with normal suggested order as on Order days, some days with emergency order on non order days, some order days with no required suggested order and so on.

Thus **as_restaurants_alcohol_performance_score_details** table will serve the primary source of suggested orders.

The suggested order logic is given below:

**Step 1**: System runs the prediction model as per the schedule in the cron job list everyday and predicts menu item sales from current day to next order day. The next 'Order Day' is determined by the customer in the setup section.
**Step 2:** The projected menu item sales are deduced down to their individual unique inventory items in ounces. Example: if we project 300 total units of Greygoose will be sold from 'Current Day to next 'Order Day' and the servings budget for each unit of Greygoose is 2 ounces, this means that projected units would be 300x2=600 ounces
**Step 3:** System reviews db to determine how much of each inventory item is currently one hand in ounces.
**Step 4:** System subtracts current on-hand from projected inventory ounces. If the results are > 0, then we proceed with the next steps else we stop here and no suggested order is required for the current day.
**Step 5:** System then subtracts pending unreceived inventory items ounces from results of step 4
**Step 6:** System multiples 20% by the results in step 5 and add the results to step 5
**Step 7:** System converts the results from step 6 into the closest quantity of minimum order delivery unit. For eg: if Greygoose's minimum order delivery unit is 35 ounces, and the results from step 6 are 75 ounces. We round the results to 70 ounces so as to stick to the delivery unit. A traditional round function should be used here.
**Step 8:** The results if > 0 must be stored in **as_restaurants_alcohol_performance_score_details** table, under the suggested orders column.

**Example:**
Today is Tuesday but next Order day is Friday. System calculates forecasted usage of Corona Beers from Tuesday - Friday. Tue 10 bottles, Wed 20 bottles, Thur 70 bottles, Fri 100 bottles = total forecasted Corona 200 bottles or = 2400oz. Each bottle of Corona contains 12oz of beer (12oz*200btls=2400oz).
Next we noticed that the restaurant already has 25 bottles of Corona on-hand (300oz) so we subtract 300oz on-hand from 2400oz predicted sales = 2100 ounces needed (2100oz/12oz=175 bottles of Corona).
Next we review unreceived orders and we see that there are 1440oz (5 cases) of Corona that has been ordered but not received. So we subtract 2100oz needed - 1440oz ordered but not received = which gives us 660oz ounces needed or 55 bottles.
Next we calculate the par and take the 660oz (55 btls) and multiply it times 20% par = we get a result of 132oz (11 btls). We then add the 660oz needed plus the 132oz par = 792oz (66 btls) total.

Note: All deliveries must be narrowed down to the minimum order allowed. Most order minimum are 1 bottle however other items such as bottled beers have a case as a minimum order. Cases of beer normally contains 24 bottles. Therefore our minimum order of 66 bottles would be rounded to the nearest full case count which would be 3 cases or 72 bottles. The customer output would be in the form of minimum order (3 cases).

Excel Table Example

| | Forecasted Usage Oz 11-23-2017 | Total Forecasted Usage in Oz | Beginning On-Hand Oz | Unreceived Orders in Oz | Inventory Needs in Oz | Suggested Order Oz 11-23-2017 | Minimum Order | Minimum Order Qty | Unit Qty in Ounces | Output in Units or Cases |
|---|---|---|---|---|---|---|---|---|---|---|
| Greygoose | 100 | 625 | 95 | 24 | 506 | 607 | Units | 1 | 36 | 17 |
| Budweiser | 888 | 2268 | 142 | 0 | 2126 | 2551 | Case | 24 | 12 | 9 |
| Miller LT | 732 | 3336 | 0 | 0 | 3336 | 4003 | Case | 24 | 12 | 14 |
| Patron | 127 | 214 | 85 | 156 | 0 | 0 | Units | 1 | 26 | 0 |
| Deep Ellum Vodka | 96 | 965 | 0 | 978 | 0 | 0 | Units | 1 | 36 | 0 |
| Zodiac Vodka | 36 | 147 | 168 | 0 | 0 | 0 | Units | 1 | 36 | 0 |
| Makers Mark | 195 | 333 | 25 | 0 | 308 | 370 | Units | 1 | 36 | 10 |

## The Formula Flow

| | | 1 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 4 | 4 | 5 | 5 | 5 | 5 | 5 | 6 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 20-Nov | 21-Nov | 22-Nov | 23-Nov | 24-Nov | 25-Nov | 26-Nov | 27-Nov | 28-Nov | 29-Nov | 30-Nov | 1-Dec | 2-Dec | 3-Dec | 4-Dec | 5-Dec | 6-Dec | 7-Dec | 8-Dec |
| | Mon | Tue | Wed | Thu | Fri | Sat | Sun | Mon | Tue | Wed | Thu | Fri | Sat | Sun | Mon | Tue | Wed | Thu | Fri |
| Forecasted Usage | | 50 | 75 | 100 | 200 | 200 | 25 | 50 | 50 | 175 | 100 | 200 | 250 | 100 | 50 | 100 | 75 | 100 | 250 |
| Actual Usage | | 95 | 60 | 90 | 220 | 300 | 20 | 45 | 52 | 196 | 117 | 230 | 310 | 90 | 86 | 101 | 82 | 123 | 198 |
| Actual Delivery | | 0 | 150 | 24 | 607 | 0 | 0 | 11 | 0 | 316 | 0 | 804 | 0 | 0 | 0 | 0 | 209 | 0 | 538 |
| On-Hand | 100 | **5** | 95 | **29** | 416 | 116 | 96 | 62 | **10** | 130 | **13** | 587 | 277 | 187 | 101 | **0** | 127 | **4** | 344 |
| Inventory Needs | | **125** | **20** | **506** | **0** | **0** | **9** | **0** | **263** | **0** | **670** | **0** | **0** | **0** | **0** | **174** | **0** | **448** | **0** |
| Suggested Order 7am | **0** | **150** | **24** | **607** | **0** | **0** | **11** | **0** | **316** | **0** | **804** | **0** | **0** | **0** | **0** | **209** | **0** | **538** | **0** |

Example: Nov 23 the Total Forecasted Usage is calculated by looking at the range between current date to next order date which in this case is Nov 23rd - 28th which totals 625oz. The ending inventory count from the previous night will be the next days new beginning amount. In this case the beginning on-hand inventory amount is 95oz. The restaurant is due to receive a delivery of 24oz (ordered the day prior but not received in Artecsan yet). Formula 625oz - 95oz - 24oz = 506oz. Take the 506oz and multiply by 20% Par = 101oz. Now add the 506oz + 101oz = 607oz suggested order.

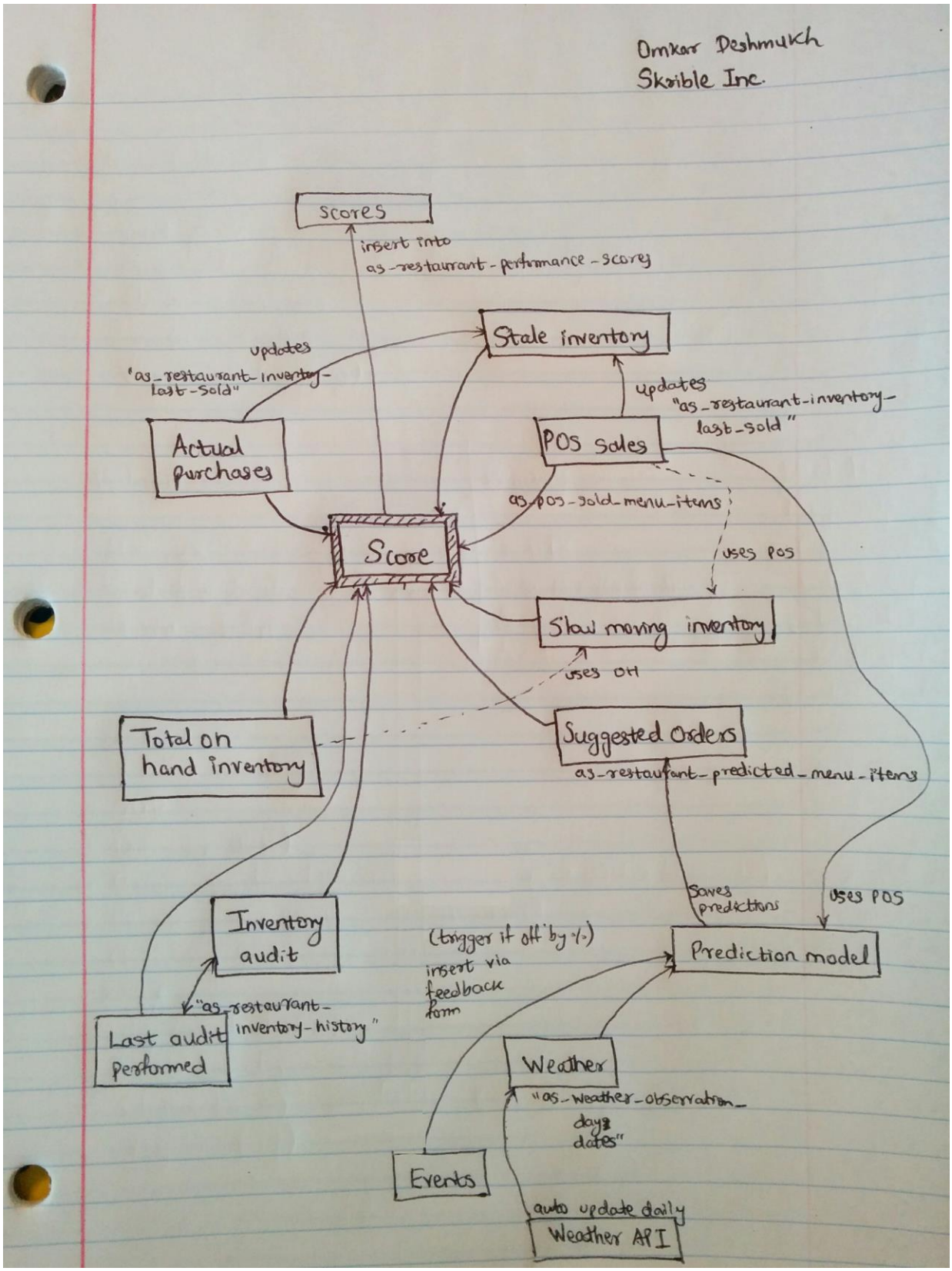
Let's assume this example is talking about a liter bottle of Brand Vodka. A full liter has an estimated 36oz so we divided 607/36 = 17 bottles of Brand Vodka. The system would then check the minimum order quantity and make the suggested order. In this case it would be 17 bottles

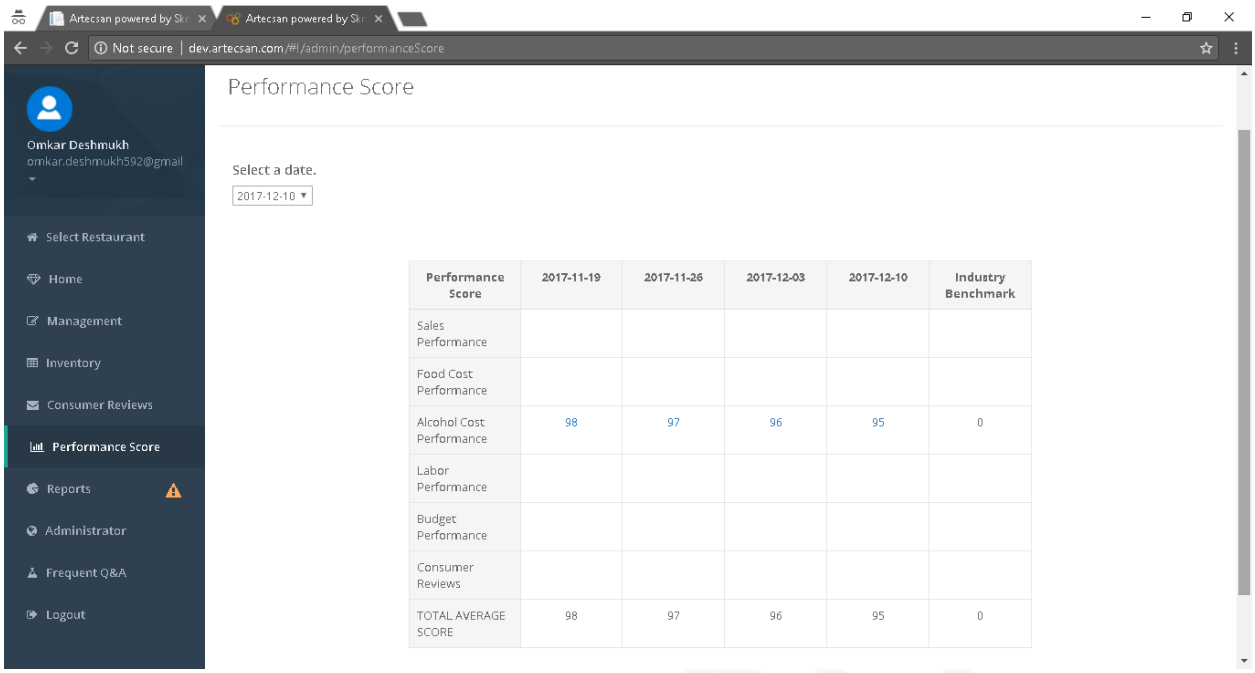
**Combining Days:** For the current day to order day formula you may have a better way of tying days together however I can share my method. In order to identify the days which needed to be tied together I created a formula that numbered the days. You will see in the top line of the image the following numbers 1,2,2,3,3,3,3,3,4,4,5,5,5,5,5,6,6,7. The formula basically starts a new number series **every delivery day**. The formula to link the days together is as follows: "on the current day I told the computer to find all the numbers that match tomorrows number and add their forecasted usage to todays forecasted usage. Example: Nov 23 looks for Nov 24 number which is 3 and add forecasted sales for all days with the # and add today's forecast to that total. The result is 625oz

In order to create the numbered days I told the computer to number the first day as 1 and all subsequent days will match the prior day unless its a delivery day. If its a delivery day the computer should add the value 1.
Formula is: if(D3="Delivery",D2+1,D2).....If day is a delivery day then create a new number for today by adding 1 plus previous days number. If it's not a delivery day then copy yesterday's number.

**Designated Delivery Days:** Keep in mind that even though the formula will run daily, the delivery days will be broken up into categories during the setup process (example: liquor can have delivery days of Mon & Wed, draft beer can have delivery days of Tu & Th, bottled beer can have delivery days of Mon & Th etc...). Those categories will designate the Order and Delivery days which is the basis for our calculation.

## 4. Performance Score:






**Summary:**
The performance score module basically summarizes the data from various parts of the system and assigns a score for the week that just passed.
It runs on a Monday, calculates and saves the score and sends out an email to the concerned users at 6AM with the performance score for the previous week (M-S).

**What's already done:**
The score calculation function is almost completed (except for a few things) which fetches the required data from the original tables. But we would want the score function to fetch data from **as_restaurants_alcohol_performance_score_details** table primarily and a few other tables.

The front end fetches data from the score table. Each score when clicked opens a page which displays detailed data from the **as_restaurants_alcohol_performance_score_details** table (yet to be created) for the concerned week.

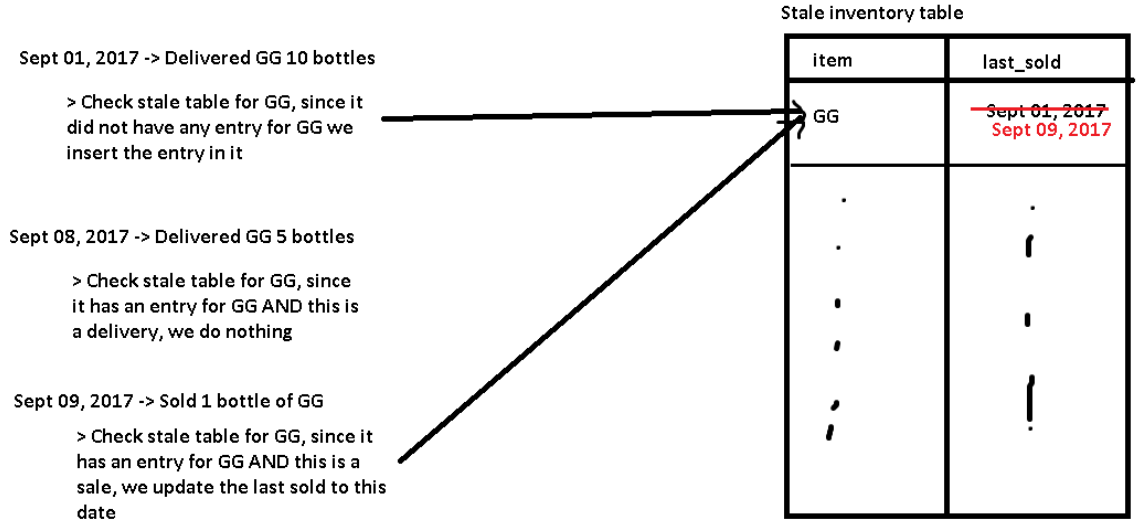
**Requirements:**
**as_restaurants_alcohol_performance_score_details** table is required to be updated daily or when after certain events occur.
For eg: It should store for each day, for each inventory item of the restaurant whose Total On Hand > 0, the Actual Purchases, Suggested order, POS sales, Total On Hand in ounces.

We would like the score calculation function to run dynamically for all the restaurants on a Monday morning and send emails at 6AM.

We would also like you to design queries which fetch the On Hand inventory value in $ and also the + and - cost variance. Cost variance = (What we say they should have on hand) minus (what they actually have on hand after they did an inventory audit).

Cost variance = Over/short that weekly period.
Over = + cost variance
Short = - cost variance

The function should also fetch the stale inventory value in $. Stale inventory are all the inventory items which have not been sold for the last 7 days. The as_restaurants_inventory_last_sold table stores the data which says what inventory item was last sold when.
This last_sold column in this table must be updated every time an inventory item is either sold or was delivered. The logic for the same is explained below.




The function should also fetch the $ value of slow moving inventory items. Slow moving inventory items are those items where the ounces sold for that item that week was < 8.25% of the On Hand of the Monday morning. For each item add 0.25 and return that sum as score_object['p20'].

**More details:**
Now it is obvious that the week that the score takes in as the input M-S will have overlapping Order period from previous and the next week as shown below:

| 20-Nov | 21-Nov | 22-Nov | 23-Nov | 24-Nov | 25-Nov | 26-Nov | 27-Nov | 28-Nov | 29-Nov | 30-Nov | 01-Dec |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Wed | Thu | Fri | Sat | Sun | Mon | Tue | Wed | Thu | Fri | Sat | Sun |

Yellows are the Order days
Reds are the days that the score function will use to calculate the score.

So assuming on Nov, 20 the prediction model will make a prediction and cover the days from Nov 20 to Nov 27 and eventually suggests an order. We see that the suggested order although covers

Nov 20 to Nov 27, it will have a date of Nov 20. Also the score function only reads data covering the dates Nov 25 to Dec 01.
To accommodate the score functions requirements, we have to spread the suggested order across the days that it covers. This is done using a logic that we developed and does not need any data manipulation in the DB.

| | 20-Nov | 21-Nov | 22-Nov | 23-Nov | 24-Nov | 25-Nov | 26-Nov | 27-Nov | 28-Nov | 29-Nov |
|---|---|---|---|---|---|---|---|---|---|---|
| | Wed | Thu | Fri | Sat | Sun | Mon | Tue | Wed | Thu | Fri |
| Forecasted Cost | 50 | 25 | 100 | 150 | 90 | 10 | 50 | 75 | 90 | 130 |
| Actual Sales | 65 | 20 | 216 | 175 | 93 | 15 | 48 | 72 | 98 | 125 |
| % of Forecast | 0.09 | 0.04 | 0.25 | 0.26 | 0.15 | 0.02 | 0.08 | 0.12 | | |
| Daily Orders | 69 | 27 | 189 | 194 | 109 | 15 | 59 | 88 | | |
| | | | | | | | | | | |
| Suggested Orders | 550 | | Order on 11/20 Delivered on 11/21. Start with zero on-hand | | | | | | | |
| Actual Sales | 704 | | Customer actually ordered 200 more for the same order period | | | | | | | |
| Total | 1254 | | | | | | | | | |
| | | | | | | | | | | |
| Actual Purchases | 750 | | | | | | | | | |

This formula is designed to us proportion to calculate how much of a transaction belongs to the current week vs the next week. We start by calculating the total forecasted cost of all items suggested from 'Order Day' to 'Order Day' which is highlighted in yellow. Example on Wed 11/20 thru Wed 11/27 we suggested the restaurant purchase a total $550 in total inventory items. During the same period the restaurant actually sold $704 in total inventory items. During the same period the restaurant actually purchased $750 in total inventory items. We combined the Forecasted Cost to Actual Cost for each day and divided it by the combined weekly total. Example: Fri 11/22 we forecasted the restaurant cost to be $100 but the actual cost was $216. The total of the two was $316 or 25% of the total $1254 (Total is calculated from Order Day to Order Day for both Suggested Order and Actual Sales). We then look at the Actual purchases for the same period and applied the daily percentages. So on Friday Nov 22 the value of the inventory order was $189.

Thus the above logic will calculate for every order day of the restaurant (Nov 20 and Nov 27) and calculate the daily orders for all the days between the order days.

1. The logic take Nov 20 as the input and calculate daily orders for the dates: Nov 20 to Nov 27
2. The logic take Nov 27 as the input and calculate daily orders for the dates: Nov 27 to Dec 04

We see we have two daily orders for Nov 27. We disregard the value made by the step 2.

## 5. Order of Cron jobs:

Following list gives the order of Cron jobs:

1. Pull the POS data from the restaurant POS systems (already exists 4am daily)
2. Update the On Hand values and save the same into **as_score_details** (Daily 5am)
3. Calculate the score using the data from **as_score_details**, store the score in **as_restaurants_performance_scores** (after #2 on Monday). And send an email containing the scores to the concerned users (Monday Morning 6am)
4. Pull weather data from the weather API (either after #2 daily OR after #3 on Monday)
5. Run the sales prediction model and save the results into the appropriate table (Daily after #4). If it is a Friday send an email containing the next weeks predicted sales in $ (discuss level of detail of email). (Friday 9am)
6. Run the suggested order logic. (Daily after #5). If the suggested order is less than 1 or if no suggested order is required, delete the predictions done in this step else save the suggested order into **as_score_details.** And if Artecsan makes a suggested order, email the order to the concerned users with the email having a link to New Orders page with it already pre-populated with the suggested order items.(Daily 7am).
7. Run the trigger logic to compare actual vs predicted sales for yesterday.(After #6). If the trigger is set off then insert an entry into **as_events_feedback_stack** and email the concerned users (Daily 8am)

---

## 6. Auto Email Notifications:

- Weekly Performance Score is emailed on Monday mornings 6am
- Predicted sales is emailed on Fridays for the upcoming week 9am
- If we detect that the restaurant is going to run short of any item(s), we send the owner a list of emergency suggested order items (for non-order days). Daily 7am
- Emails on regular order days with the list of items to order governed by the Order Schedule module. Order days at 7am
- An email when the prediction model under predicts by a certain % for the suggested order. 8am

| | Owner | Administrator | GM | Bar Manager | Kitchen Manager |
|---|---|---|---|---|---|
| Sales Forecast | x | x | x | | |
| Performance Score | x | x | x | | |
| Regular Inventory Orders | | | x | x | x |
| Emergency Orders | | | x | x | x |
| Events Feedback | | | x | | |

- On suggested orders, once the restaurant has confirmed the order please give the restaurant an option to either save, cancel or “Save & Send” the order directly to the vendor.

---