# Presented by SG
# IMMC 2020 – Flash Sale

**Summary**:

Flash sales are times when the price of products are greatly reduced, for a limited time only. A prime example of a flash sale is the Black Friday sales in the United States, where shoppers rush to purchase gifts for friends and family. However, in the mad dash that ensues, there is inevitably damage to the products on sale. These damages are borne by the company, which is detrimental to the profits earned. Hence, it is important for these companies to minimise damages incurred during the flash sale. In order to do so, owners must know the optimal layout that minimises damages. By identifying the major layout factors that cause damage to products, we can come up with a suitable floor plan to minimise the damage done to the items.

Our first task was to identify the factors that lead to damage of the items. By relating it to measurable quantities such as the density of people, we can estimate the level of damages in the entire shop. Events that lead to damage of the items include collision between shoppers, collapse of shelves, collision of shoppers with shelves and so on. We identify the level of damage to be primarily dependent on the density of shoppers and the price of the damaged goods, and calculate a damage value from these two factors.

Our second task was to identify the factors contributing to the popularity of products. This was necessary as popularity of products would affect the density of shoppers around certain areas. In this section, we used economics to study the quantifiable factors such as size and discount amount in order to calculate a popularity index for all items. This popularity index was then used to find the density of shoppers around each department.

Our third task was to develop a mathematical model to create the best layout plan of the store to minimize damage. This was further separated into three steps. Firstly, we determined the physical factors that affect density of shoppers, which include the number of shelves and relative placements of the departments. Secondly, based on the floor plan provided, we ran a simulation to calculate the total damage value for different arrangements of departments, to find the optimal locations for each department. Finally, we generated our own floor plan based on the data gathered from the simulations to minimise damage done to the products. We then tested our own floor plan and made adjustments to optimise the floor plan.

Finally, we performed sensitivity analysis on various independent variables in our model to see how our model performs with different types of parameters. The results were that our model was robust, performed as expected and did not contradict with logic. Thus, we conclude that our model is indeed well developed for our task.

## Table of Contents

## <u>**Section 1: Introduction**</u>

### 3.1. General assumptions

We will be considering general assumptions in the event which determine how we create a model to determine the level of damage for different department layouts.

1) We assume that the size of one shopper is between 40-60 arbitrary units (a.u.) and the size of the largest product (the refrigerator) is 100 a.u.. Thus, each cell can only hold a maximum of 300 arbitrary units ( $2 \times$ size of shopper (50 a.u.) + $2 \times$ size of refrigerator) or there will definitely be a collision.
2) We assume that each shopper will take different products until they carry a maximum of three products or such that their total size (person and products) is less than 300 a.u. since it will be difficult for a normal shopper to walk around with too many or very heavy products.
3) We assume that each cell of the shelf has the same carrying capacity and can carry up to 3 different units of products or 150 a.u. worth of items. This is because the store is likely to space out their items to prevent overcrowding during a flash sale.

## <u>**Section 2: Types of damage to products**</u>

Damage to products can be caused by 4 types of events, but each type of event is dependent on a few common factors.

| Table 1 | Variables associated with damage level |
|---|---|
| $D_T$ | Total damage in all cells |
| $D_{cell\_shoppers}$ | Level of damage in each cell due to collision of shoppers |
| $D_{cell\_shelf}$ | Level of damage in each cell due to collision of shoppers with shelf |
| $D_{cell\_drop}$ | Level of damage in each cell due to shoppers dropping items |
| $S$ | Stability of one shelf |
| $Z_p$ | Size of products |
| $Z_S$ | Size of a shopper |
| $\rho_{cell}$ | Density of cell |
| $\rho_{adjusted}$ | Adjusted density of each cell |
| $d_p$ | Distance between two adjacent products on the same shelf |
| $h_s$ | Height of each layer of the shelf |
| $n_p$ | Number of products each customer is holding |

| | |
|---|---|
| $pr_{cell}$ | Price of the cell |
| $pr_{total}$ | Total price of all goods in the store |
| $pr_{adjusted}$ | Adjusted price of each cell |
| $pr_{shelf}$ | Total price of all goods on a shelf |
| $P_{collision}$ | Probability of collision |

### 2.1. Damage to entire shelves of products from collisions of shoppers with shelves

If shoppers collide with the shelves of products due to the large crowd and narrow aisle, the shelves may topple, damaging entire shelves of products. The level of damage can be modelled to be based on the stability of the shelves and the density of the shoppers around the shelves.

Assuming that the height of each shelf is the same, the stability of shelf is

$$S = kA, k \in \mathbb{R}$$

When the shelf topples, all the products on the shelf are damaged. The price of shelf toppling is hence

$$pr_{shelf} = \sum_{all\ products\ on\ the\ shelf} pr_{product}$$

The density of each cell is

$$\rho_{cell} = \sum_{all\ shoppers\ on\ cell} (Z_p + Z_s)$$

The price of each cell is the sum of the prices of all the products shoppers are carrying is

$$pr_{cell} = \sum_{all\ products\ carried\ by\ shoppers} pr_{product}$$

We assumed that the probability of collision with a shelf is linearly related to the size of the shopper based on the kinetic theory of gases, which states that the pressure on the container walls is proportional to the size of the molecules. When the size of the shopper and his products is 150 a.u., the shopper will definitely collide with the shelf beside and $\boldsymbol{P_{collision}} = 1$.

$$P_{collision} = \frac{Z_p + Z_s}{150}$$

The damage in each cell due to collision of shoppers with shelves is therefore

$$D_{cell\_shelf} = \frac{1}{S} \times P_{collision} \times (pr_{shelf} + pr_{cell})$$

### 2.2. Damage to adjacent products when grabbing products from shelves

Shoppers may accidentally knock adjacent products off the shelves when securing products they wish to purchase off the shelf due to a lack of consideration for their surroundings.

The level of damage is related to ease of taking products from shelves. The closer the proximity between products, the more likely the shopper is to cause damage to adjacent products. At the same time, when the vertical distance between shelves is larger, it is easier for shoppers to grab their desired products quickly, thus they are less likely to damage adjacent products.

In the model, given that there are only a maximum of 3 units of objects per shelf, they should be sufficiently spaced such that the products are undamaged from the knocking of adjacent products. Thus, we did not consider this source of damage in our model.

### 2.3. Damage due to collision between shoppers carrying products

When shoppers rush around the store, they may collide with each other, causing the products they have on hand to fall, resulting in damages. The level of damage depends on the density of the shoppers, and the total price of their products. The greater the density of shoppers the greater probability of collision.

Based on the kinetic model of matter, the collision frequency is proportional to the square of the number of particles in a system. Thus, we model the probability of collision between shoppers to be proportional to the square of the density of each cell.

$$P_{collision} = \frac{[\min(\rho_{cell}, 300)]^2}{300^2}$$

where 300 a.u. is the sum of the size of 2 shoppers and 2 fridges.
Therefore, the damage in each cell due to collision between shoppers is

$$D_{cell\_shoppers} = P_{collision} \times pr_{cell}$$

### 2.4. Damage due to dropping products due to crowd and rush

When shoppers are holding many products, they may drop their products due to the amount they are carrying. Therefore, the equation for damage due to dropping products is the same as damage due to collision between shoppers carrying products.

$$D_{cell\_drop} = \frac{[\min(\rho_{cell}, 300)]^2}{300^2} \times pr_{cell}$$

### 2.5. Total damage

The total damage incurred by the shop is therefore

$$D_T = \sum_{all\ cells} D_{cell}$$

## <u>Section 3: Factors affecting popularity of products</u>

We calculate a value of popularity for each product, depending on the inherent popularity and the popularity due to the discount.

| Table 2 | Variables associated with popularity rating of products |
|---|---|
| $R_{consumer}$ | Customer rating of the product |
| $R_{brand}$ | Brand rating of the product |
| $R_{net}$ | Net rating of the product |
| $PED$ | Price elasticity of demand |
| $PED_r$ | Relative price elasticity of demand |
| $pr_{product}$ | Price of the product |
| $Y$ | Median daily income of the shopper |
| $Per$ | Percentage of households with the specific type of the product |
| $U$ | Utility gained by shoppers from buying the product |

| | |
|---|---|
| $U_0$ | Utility lost by the shopper from buying the product at initial price |
| $U_1$ | Utility gained by shoppers from the discount |
| $U_{final}$ | Final utility lost by shoppers from buying the product at the discounted price |
| $U_{net}$ | Net utility gained by shoppers from buying the product at the discounted price as compared to the initial price |
| $L_{aversion}$ | Loss aversion factor of inherent popularity of the product |
| $S_{inherent}$ | Inherent saliency of the specific product |
| $S_{surroundings}$ | Saliency of surrounding products to the specific product |
| $d_p$ | Distance between the shelves |
| $Pop_i$ | Inherent popularity of the product |
| $Pop_d$ | Increase in popularity due to the discount |
| $Pop_{relative}$ | Relative popularity of product |

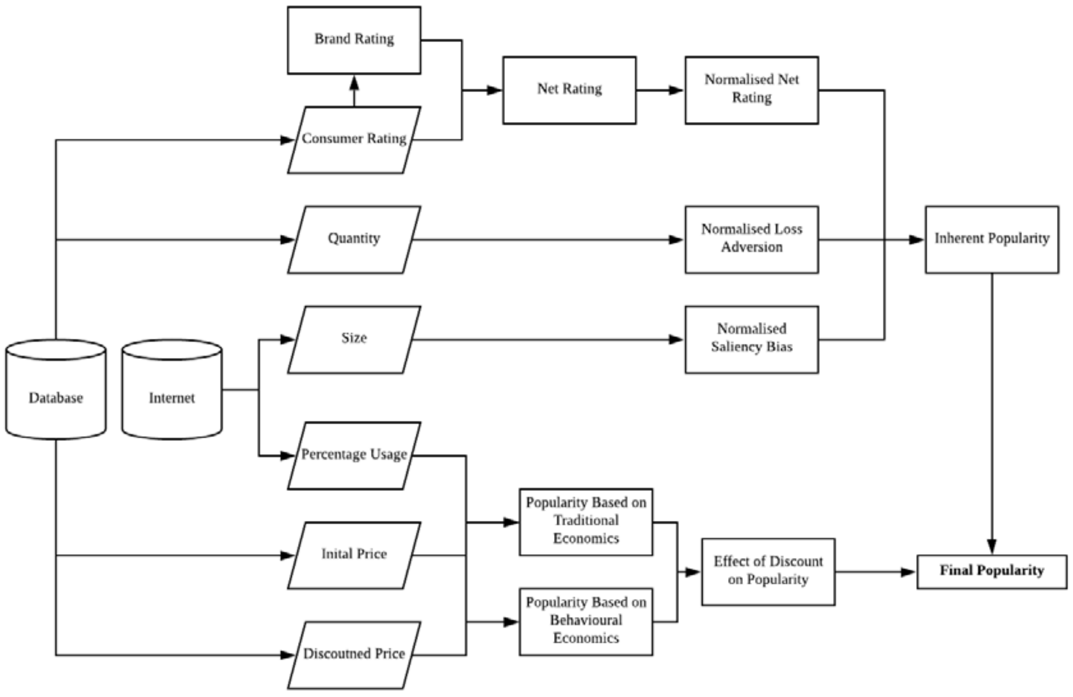


Figure 1: Flowchart of popularity calculations

## 3.1. Net rating of product

The net rating of the product depends on the customer rating and brand rating of the product.

### 3.1.1. Customer rating

The higher the customer rating, the more popular the item will be. Shoppers see it as more desirable, making it more popular.

### 3.1.2. Brand rating

When choosing products, consumers do not only consider the customer rating of individual products, but also the brand of the products. Taking the average of customer ratings for different products from the same brand gives us a brand rating.

$$R_{brand} = \frac{\sum_{i=1}^{n} R_{customer} of\ the\ nth\ item\ from\ the\ specific\ brand}{n}$$

The net rating is a weighted average of brand and customer ratings. Brand rating is given a lower percentage of 20% as 20% of consumers purchased an item solely based on their opinion of a brand.

$$R_{net} = 80\% \times R_{customer} + 20\% \times R_{brand}$$

## 3.2. Saliency bias

Saliency bias refers to the fact that individuals are more likely to focus on products or information that are more prominent. The more obvious the item is, the more likely shoppers are to take notice of the item and purchase the item.

Inherently, the effect of saliency bias increases with increasing size of the product, in this model, they are assumed to be proportional.

$$Sa_{inherent} = sigmoid(Z_p)$$

Additionally, the product location may affect the shoppers' decisions as well. The effect of saliency bias on shelves of products can be modelled to be based on the proximity and quantity of adjacent shelves as well as the saliency of products in the adjacent shelves. A greater saliency of adjacent shelves is assumed to lower the saliency of the shelf in question as a shopper's attention are deviated away to the nearby shelves.

Thus the net effect of saliency bias on a shelf *i* next to a shelf *j* is:

$$saliency(i) = \ Sa_{inherent}(i) - Sa_{surroundings}(i)$$

$$saliency(i) = Sa_{inherent}(i) - \sum_{shelf\ j}^{all\ shelves} dist(i,j) \times saliency(j)$$

When computing *saliency(i)*, we initialise all shelves with saliency equal to their inherent saliency and iteratively update the values of the saliency of the shelves.

## 3.3. Loss aversion factor

Loss aversion is a principle which suggests that "losses loom larger than corresponding gains". When the quantity of a particular item available is smaller, shoppers feel more urgent about buying the products since they try to avoid the prospective losses resulting from failing to buy the products. Therefore, products which are already in small quantities on shelves will be more popular and more likely to be purchased. Hence, the loss aversion factor of popularity of the item is inversely related to the quantity available.

$$L_{aversion} = -e^{quantity\ available \times b}, where\ b\ is\ the\ loss\ aversion\ constant$$

## 3.4. Impact of discount

The discount affects the popularity of the item being sold. To analyse the relationship between the discount amount and the popularity of the item, we utilise traditional and behavioural economics in terms of calculating the price elasticity of demand of a good and evaluating the expected utility of the product given that humans are not rational decision-makers.

### 3.4.1. Price Elasticity of Demand

Price Elasticity of Demand (PED) measures the responsiveness of quantity demanded of a good to a change in its price. It thus measures change in desirability of the good due to the discount.

$$PED = \frac{\%\Delta Q}{\%\Delta pr} = \frac{dQ}{dpr_{product}} \times \frac{pr_{product}}{Q}$$

In comparing the relative impact of discount, we will consider the relative PED. PED has 4 determinants: proportion of income, degree of necessity, availability of substitutes and time period. As availability of substitutes is generally high for the electronic appliances available, and the time period is equal to the time of the flash sale, we did not consider these factors.

$$Proportion\ of\ income = \frac{pr_{product}}{Y}$$

To measure the degree of necessity, we take the percentage of shoppers who use the item as an indicator of its necessity. Relative PED is taken to be the product of these two factors.

$$PED_r = \frac{pr_{product}}{Y} \times Per = \frac{dQ}{dP} \times \frac{pr_{product}}{Q}$$

As price changes from $P_0$ to $P_1$, quantity changes from $Q_0$ to $Q_1$.

$$\int_{P_0}^{P_1} \left(\frac{1}{Y} \times Per\right) dpr_{product} = [lnQ]\ from\ Q_0\ to\ Q_1$$

$$(pr_1 - pr_0) \times \frac{1}{Y} \times Per = \ln(\frac{Q_1}{Q_0})$$

As price elasticity of a good increases, any fall in price will result in an increase in quantity demanded, hence showing an increase in popularity.

### 3.4.2. Prospect Theory

Prospect theory states that the loss in value from a certain loss is greater than the gain in value from a gain of the same amount. This is depicted by having a steeper utility curve at negative utility. People tend to think of discounted objects as losing the initial cost of the item and then gaining back the discount. Thus, to quantify the effect of the discount, we see shopper's change in utility before and after the discount.

In this case, the "loss" is the amount spent on the item without discount, and the gain is the amount of the discount.

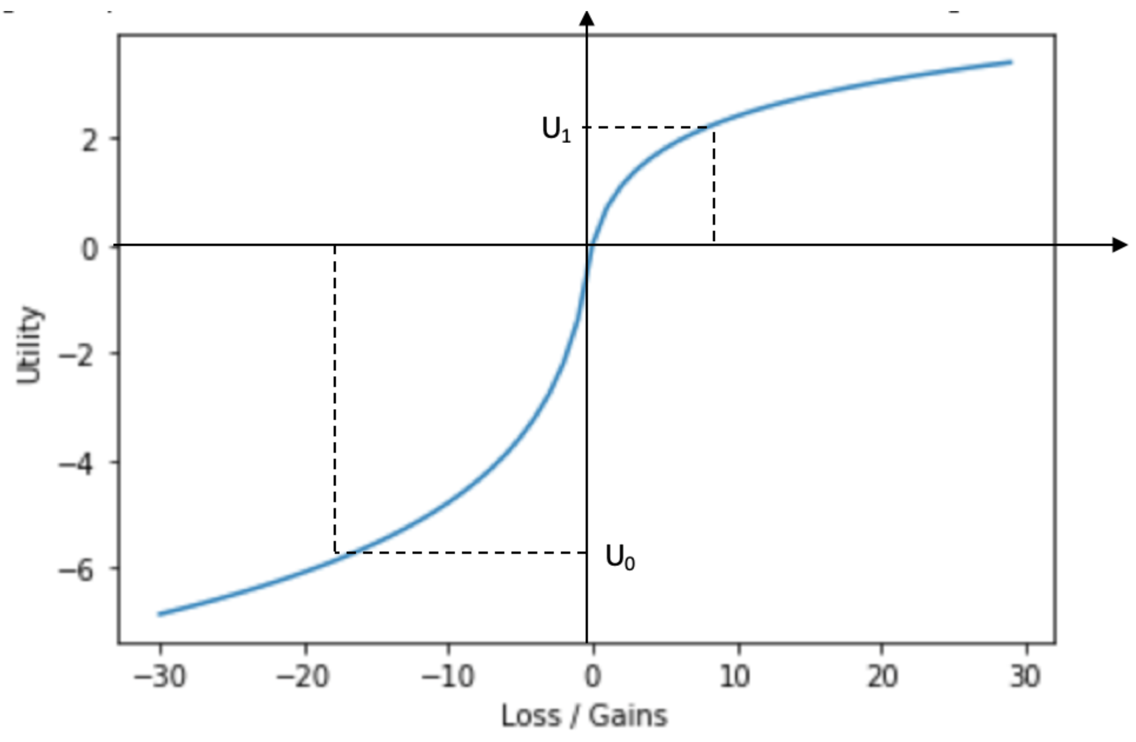


Figure 2: Utility function

U is a function from [0,∞) to [0,∞) which is strictly concave, strictly increasing, twice differentiable, and such that U(0)=0 (Fig. 2). We choose the function

$$U(x) = \log(x+1)$$

As it meets the required conditions while being easy to implement. To quantify the effects of the discount, we measure the change in the utility as a result of the discount.

$$U_{final} = U_0 + U_1$$

$$U_{change} = U_{final} - U_{initial} = (U_0 + U_1) - U_0 = U_1$$

## 3.5. Relative popularities of products

We first normalised the value of each factor such that they were within the range of 0 to 1, then used a weighted average system to calculate relative popularity and impact of discount of each item (Table 3).

| | Factor | Weight |
|---|---|---|
| Inherent popularity, $Pop_i$ | Net rating, $R_{net}$ | 0.5 |
| | Saliency bias, *saliency(i)* | 0.06 |
| | Loss aversion, $L_{aversion}$ | 0.44 |
| Factors relating to discount, $Pop_d$ | Relative Price Elasticity of Demand, $PED_r$ | 0.5 |
| | Prospect theory, $U_{change}$ | 0.5 |

Table 3: Weights of each factor of popularity

The net rating of a product has the highest weight as with the products being much cheaper, shoppers have a tendency to buy high quality products, especially since they would want to make their money worth and spend more on higher quality products which would otherwise be much more expensive.

Loss aversion has the second highest weight as flash sales do not occur very regularly. When shoppers are satisfied with the quality of the products and want to buy the item, they tend to be concerned about the quantity available so as to not lose out. Saliency bias has the lowest weight as during flash sales, many shoppers usually already have what they want to buy in mind and the location of the products and whether they are obvious is less likely to affect its popularity.

$$Pop_i = 0.5 \times R_{net} + 0.06 \times saliency(i) + 0.44 \times L_{aversion}$$

The factors relating to discount was not considered and not assigned a weight as it is an additional factor which adds on to the popularity. We took the mean of the factors relating to the discount.

$$Pop_d = \frac{(U_{change} + PED_r)}{2}$$

Therefore, relative popularity is

$$Pop_{relative} = sigmoid(Pop_i + Pop_d)$$

$$Where\ sigmoid(x) = \frac{1}{1 + e^{-x}}$$

**Section 4: Description of store layout factors**

| Table 4 | Variables associated with store layout |
|---|---|
| $n_{type}$ | Number of products for each type |
| $n_{dept}$ | Number of products in each department |
| $n_{shoppers}$ | Number of shoppers in the store |
| $Pop_{dept}$ | Popularity of the department |

## 4.1. Number of shelves

The greater the number of shelves, the higher the density of shoppers, and the more likely it is for collision to occur. Thus, the greater space available for movement will decrease the damage, assuming that the number of customers remains the same.

That being said, with a greater number of shelves, the density of products on shelves is lower. Thus, shoppers will not have to crowd around shelves as much, potentially resulting in less damages.

## 4.2. Location of the most popular departments

Assuming that the shoppers are rational, they will rush to grab the most popular products first. Thus, shoppers tend to gather at the shelves of the most popular products, increasing risks of damage at those locations. Moreover, as shoppers run to the products by the order of popularity, the location of the popular products will also affect the paths of the shoppers in the shop, thus affecting the most possible locations of the collisions happening. However, we did not consider the location of individual products as it is logical for products to be grouped in departments, thus we use the popularity index of each department instead (Table 5).

To differentiate the different departments, we calculated a popularity index that ranks the popularity of the department as a whole by taking the mean popularity of all items in the department.

$$Pop_{dept} = \frac{\sum_{all\ types\ of\ items\ in\ the\ department}(Pop_{relative} of\ each\ type \times n_{type})}{n_{dept}}$$

| Department | $Pop_{dept}$ | Popularity ranking |
|---|---|---|
| Appliances | 0.82373 | 1 |
| Audio | 0.76331 | 6 |
| Cameras | 0.77866 | 5 |
| Cell phones | 0.75089 | 7 |
| Computers and tablets | 0.79837 | 4 |
| TV and Home theatre | 0.81369 | 2 |
| Video Gaming | 0.81305 | 3 |

Table 5: Popularity of each department

### 4.3. Type of Shelves

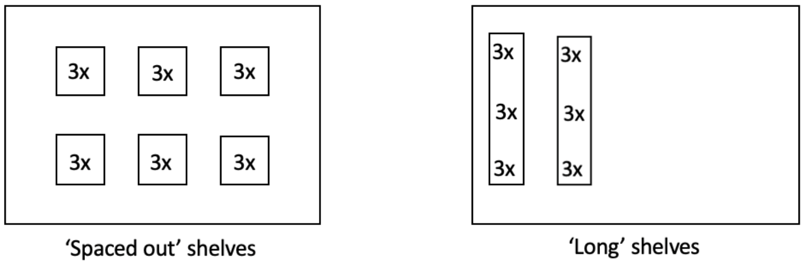


Figure 3: Types of shelves

The first type of shelf arrangement are 'spaced out' shelves where cells are in small clusters. This arrangement creates multiple paths between the shelves for the shoppers to move, reducing the probability of their paths overlapping. Therefore, such an arrangement reduces the density of shoppers around the shelves, reducing probability of collision.

The second type of shelf arrangement are 'long' shelves where cells are in a long strip. In such a situation, shoppers have limited paths to move to the counters after taking their desired products. The probability of paths overlapping in the narrow path between the adjacent shelves increases. However, the area of each path increases, reducing probability of collision (Fig. 3).

### 4.4. The location of cashier counters

Since the counters are the final destination of all the shoppers, the location of the counters will significantly affect the paths of the shoppers. Areas near cashier counters are also places where there will be a high density of shoppers, and based on factors previously identified, this would increase the likelihood of collision.

## Section 5: Model and Results analysis

### 5.1. Assumptions

1) Despite shoppers having different tastes and preferences, our model assumes that they buy from the top 5 most popular products based on the calculated relative popularity of each product.
2) Assume there is a continuous flow of shoppers into the store as during a flash sale, there are a lot of shoppers trying to secure the discounts. This is realistic as there is typically a queue outside a store during a flash sale.
3) Assume all shoppers are travelling through the store at the same speed. This is a reasonable assumption as the crowd size limits how fast each individual can travel.
4) The model assumes all products in the shop are sold out after the sale. This is realistic as a common feature of flash sales is their limited quantity of products available.

## 5.2. Description of model

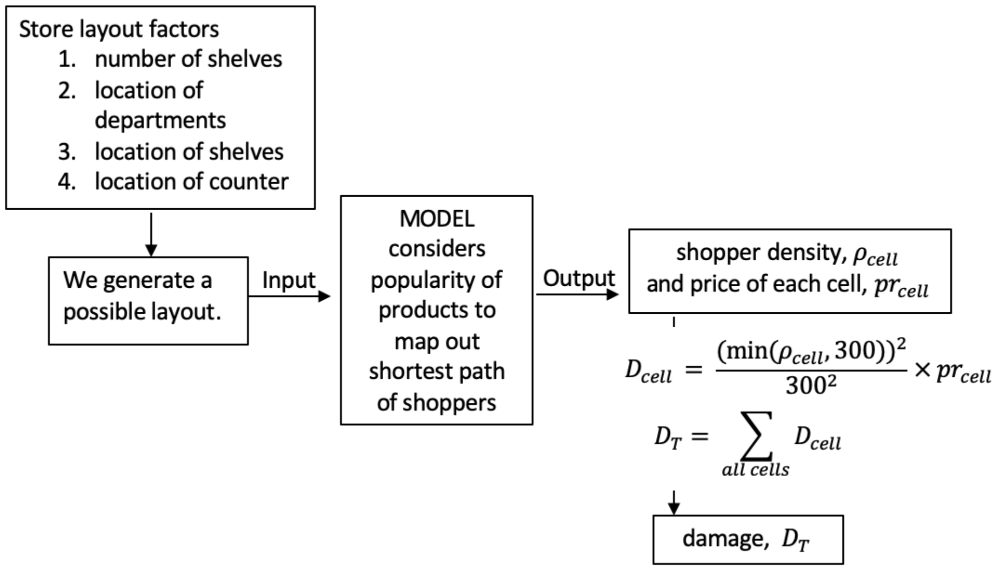

Store layout factors
1. number of shelves
2. location of departments
3. location of shelves
4. location of counter
We generate a possible layout.
Input
MODEL considers popularity of products to map out shortest path of shoppers
Output
shopper density, $\rho_{cell}$ and price of each cell, $pr_{cell}$
$D_{cell} = \frac{(\min(\rho_{cell}, 300))^2}{300^2} \times pr_{cell}$
$D_T = \sum_{all\ cells} D_{cell}$
damage, $D_T$

Figure 4: Description of model

We used the above mentioned 1m by 1m grid system for this model. The shelves are marked out on the grid (in scale) and we logically divided the shelves into departments such that the number of shelves in each department is similar.

Each department area is indicated by a number, and products are placed into shelves according to their department as shown in Figure 5.

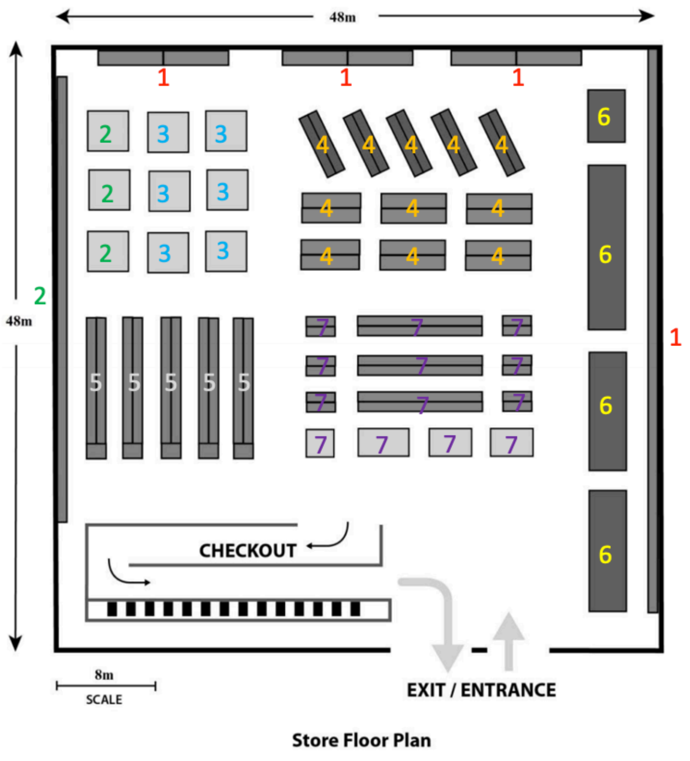


Figure 5: Division of departments

The products each shopper goes to is one of the top 5 most popular products found by the relative popularity, $Pop_{relative}$ of each product. Each shopper moves from shelf to shelf taking different products until they carry a maximum of three products or 300 arbitrary units.

We used the A* search algorithm to generate the paths of our shoppers. It is commonly used for path-finding purposes to efficiently find the shortest path by assigning heuristic values to the cells. The lower the heuristic value, the closer the action brings the shopper to their target destination. Hence, we applied this to individual shoppers in our model, such that shoppers always take the shortest path from the entrance of the shop to the shelf of their desired products. The algorithm is generally representative of a rational shopper that takes the shortest path to get their products quickly.

To determine the density of shoppers in each cell, we superimposed the path travelled by all the shoppers together onto the same grid, such that each cell on the grid shows the sum of the sizes of all the shoppers and the products they have on that cell.

Similarly, we calculate the price density of cells by superimposing the path travelled by all shoppers together where the price density in each cell is the sum of the prices of all the products of all the shoppers that traversed the cell.

Using the values and equations above, the damage done can then be calculated (Fig.4).

### 5.2.1. First basic model

We ran the simulation on a 8*8 grid with and obtained a reasonable result where damage was concentrated at certain areas. In this model, the entrance was at 7,7, the exit was at 7,6 and the counter was at 7,0. We ran the simulation with only 3 shoppers shopping for 2 types of products, placed at the shelves in 1,2 and 1,5 respectively (Fig.6). Note that coordinates are labelled as (y,x) where 0,0 is the upper left-hand corner.

This assured us that our model worked and we hence proceeded to increase the number of the shelves and the shoppers based on the realistic situation.

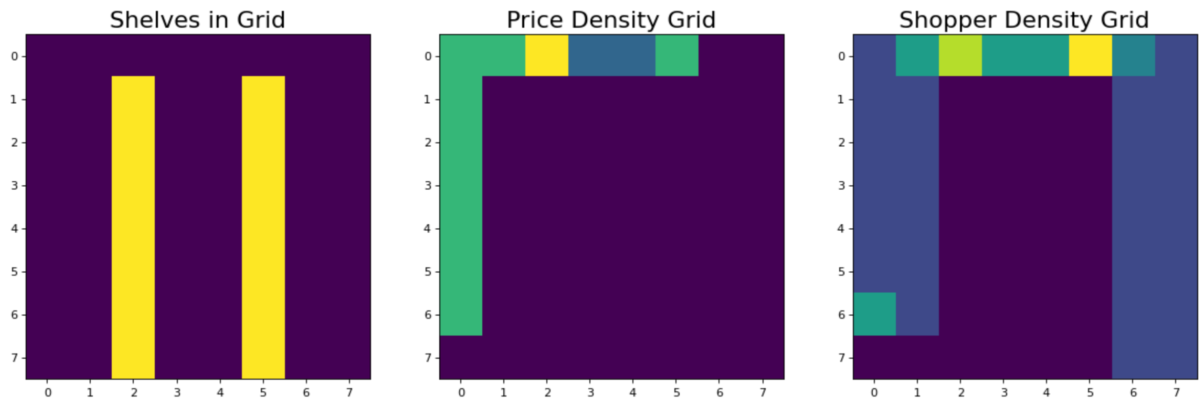


Figure 6: First basic model

This showed that our path finding algorithm was working because the shoppers visited the required shelves and did not waste time walking in the region between the 2 rows of shelves.

### 5.2.2. Fine Tuning

**Allowing shoppers to avoid crowds**

In the previous model, shoppers would move without regarding the current density of other shoppers. However, this is not realistic as shoppers tend to avoid places with high density of people as that would let them move more quickly. Thus, we tweaked the model such that the heuristic value of the cells would increase with density of people. Thus, shoppers would avoid cells with higher densities of people.

**Introduction of adjusted price density of each cell**

At first, we defined the value of the products carried by each shopper to be constant and equal to the total price of the products being carried throughout the shopper's whole journey. For example, if one shopper carries products with a total price of $x, each cell the shopper walks past will be assigned a value of $x. Once the density of shoppers in any of these cells exceeds the maximum density of 300 a.u., a collision happens, the cell immediately loses all of the value assigned to it.

However, it is unrealistic for a product to be completely damaged after only one collision. For instance, the collision may only damage the product packaging, but the product can be resold at a lower price, though it may be lower than its initial value.

Another problem with this model was that when we added up the values of all the cells to calculate the damage, it would never equal to the total price of all the products being sold in the store. This is because the value $x of the same product is assigned to multiple cells that the shopper walks past and hence the value of each product was counted multiple times when we added up the value of all cells to obtain the total price.

To solve these problems, we refined our model by introducing the adjusted price of each cell, which is obtained by equally distributing the total value of products held by each shoppers over every cell the shopper has walked past. In the above case, assuming that the shopper walks past $n$ cells, the value of each cell will be assigned as $\$\frac{x}{n}$ instead.

This implies that in reality, one product could experience multiple collisions before it is completely damaged. At the same time, if the shopper takes a longer path, there would be a lower possibility of damage of the products resulting from each single collision since the damage is spread out along his path.

Thus, by replacing $pr_{cell}$ in the model with $pr_{adjusted}$, we ensured that the total price of all the cells is equal to the total initial value of all the products as well. $pr_{adjusted}$ can be obtained by taking the ratio of the price of one cell to the price of all cells, and rebasing it over the total value.

$$pr_{adjusted} = \frac{pr_{cell}}{\sum_{all\ cells} pr_{cell}} \times pr_{total}$$

$$pr_{total} = \$891{,}784.61$$

**<u>Introduction of adjusted shopper density of each cell</u>**

Similarly, by replacing $\rho_{cell}$ with $\rho_{adjusted}$, the total density of all the cells is equal to the total size of all the shoppers and products. $\rho_{adjusted}$ can be obtained by taking the ratio of the density of one cell to the sum of the density of all cells, and rebasing it over the total size of shoppers and products.

$$\rho_{adjusted} = \frac{\rho_{cell}}{\sum_{all\ cells} \rho_{cell}} \times Z_{total}$$

$$Z_{total} = \sum_{all\ products} Z_{products} + n_{shoppers} \times 50a.u.$$

### 5.3. Simulation and results analysis

By rearranging different departments around the store and finding the total damage in the store for that layout, we can find the best arrangement of departments in the floor plan that minimises damages.

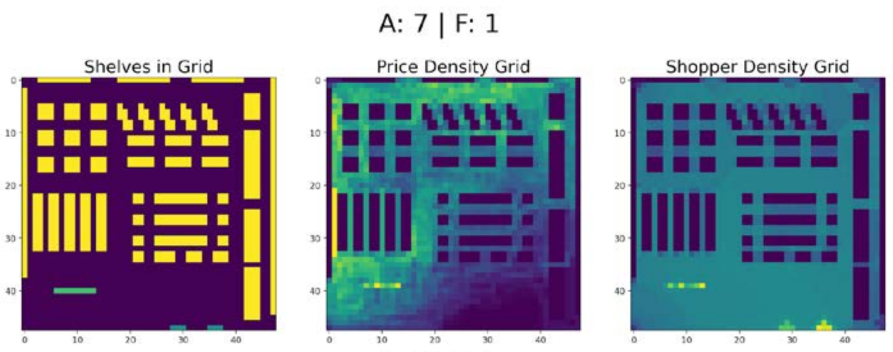
We only considered the rearrangement of the top two most popular departments (Appliances and TV & Home theatre) based on the department popularity ranking since it would take too long to simulate all permutations of all seven departments. This is also reasonable as the most popular departments will have the most significant impact on the movement of the shoppers.

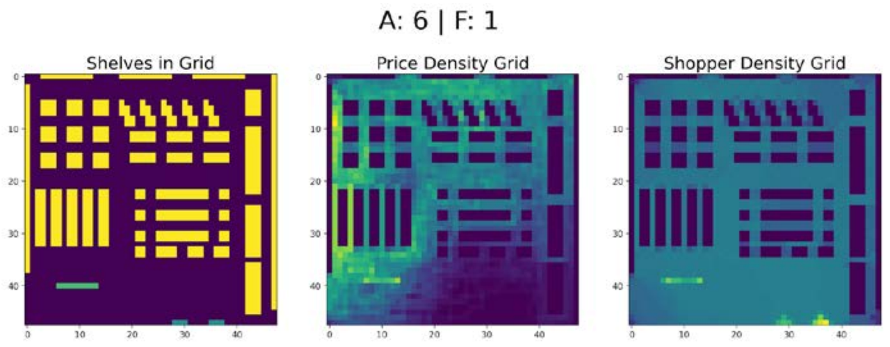
#### 5.3.1. Location of departments

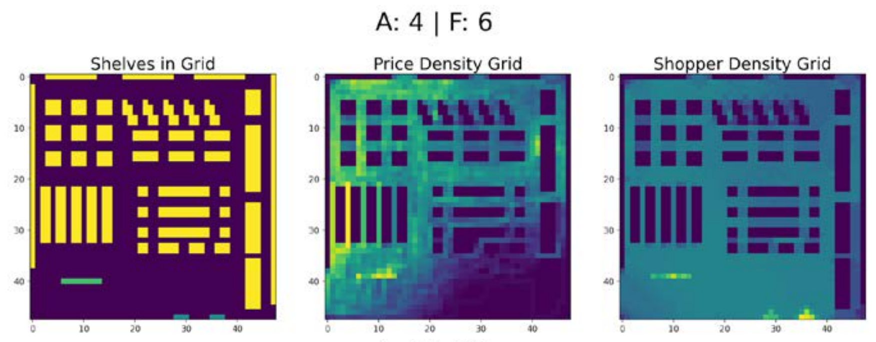
By rearranging the top 2 most popular departments, we generated 42 other arrangements. Results of the arrangements which causes the top 2 most damage and results of the arrangement which gives the 2 least damage are shown in Table 6:

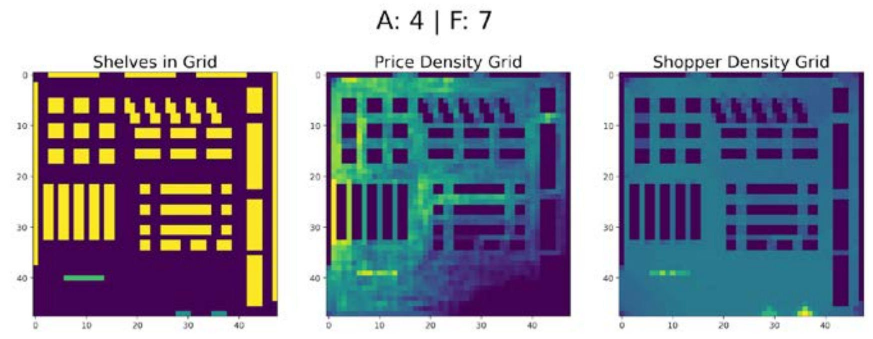
| Arrangement | Area of Appliances | Position of TV and Home Theater | Total Damage |
|---|---|---|---|
| 1 (Least damage) | 7 | 1 | $4394 |
| 2 (Second least damage) | 6 | 1 | $4432 |
| 3 (Second greatest damage) | 4 | 7 | $4752 |
| 4 ( Greatest damage) | 4 | 6 | $4772 |

Table 6: Best and worst department arrangements









Figures 7-10: Results from 2 best and 2 worst department layouts

In the shelf grid, shelves are in yellow, the counter is in green while the entrance and exit are the turquoise colours (Fig.7-10).

Based on the simulation, the level of damage is the least when the departments are arranged as seen in Figure 11.

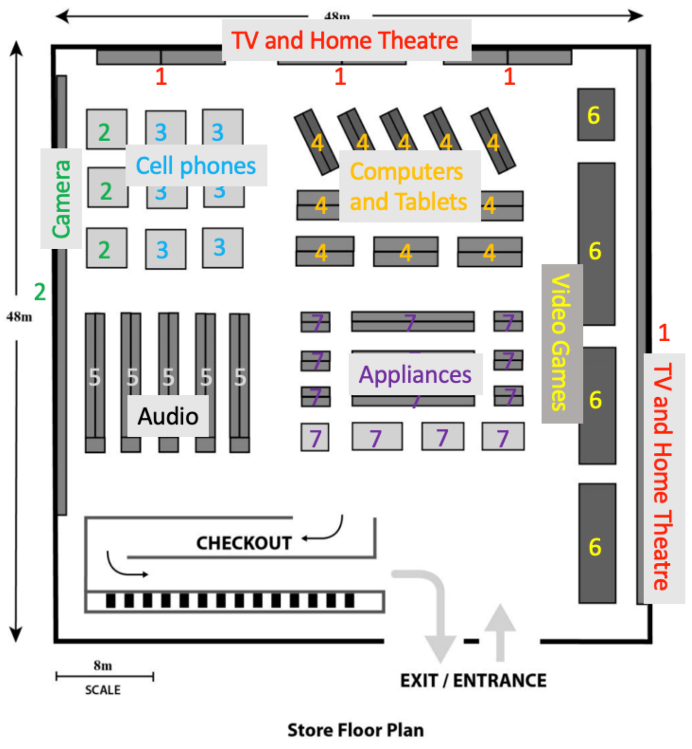


Figure 11: Optimal arrangement of departments

In better layouts, activity zones are spread out (in the lower left hand corner and in the upper right hand corner) while for the relatively worse layouts, activity zones are all on the right. We concluded that we should spread out popular products to reduce congestion around them.

Despite "Appliances" and "TV & Home Theatre" being the most popular, only "TV & Home Theatre" seemed to be creating a cluster regardless of position. Thus, it would be necessary to put the "TV & Home Theatre" far from other popular product clusters.

In all cases, zones 5 and 2 had high price density, this is likely due to the shape of shelves rather than the arrangement of departments as this high price density cluster is present in all simulations. Thus, we learnt that we should reduce the number of long and narrow aisles to allow shoppers to spread out more, lowering price and shopper density.

However, we also realized that we cannot have completely 'spaced out' shelves, where all the shelves were split into merely small blocks. This is because while there are more paths, the paths would be narrow with no large open spaces between shelves, increasing the shopper density and probability of collision.

### 5.3.2. Location of specific items

The most popular items within each department should be placed on shelves with the most space around them. In the arrangement of departments which resulted in the least damage, the most popular departments have a lot of empty spaces around the shelves. Thus, we generalised to say that the most popular items of each department should also be placed spaced out as much as possible within the department.

## Section 6: Generation of new floor plan

By considering the number of shelves, the arrangement of departments, the type and layout of shelves and the position of cashier counters, we proposed a layout which we believe would incur the lowest cost. Each factor is explained in greater detail below.

### 6.1. Minimising number of empty shelves

As empty shelves take up walking space but do not provide utility, we should lower the number of empty shelves.

As certain departments require more cells (shelves) than other departments as they have a larger quantity of items, the number of shelves for each department should be different to correspond to the number of items in each department instead of allocating a similar number of cells to all departments (Table 7).

| Department | Minimum shelves in terms of cells |
|---|---|
| Appliances | 119 |
| Audio | 18 |
| Cameras | 59 |
| Cell phones | 19 |
| Computers and tablets | 159 |
| TV and Home theatre | 134 |
| Video Gaming | 62 |

Table 7: Minimum number of shelves

### 6.2. Optimal arrangement of departments

For our proposed layout, as we have already obtained the optimal layout of departments, we would be focusing on the optimal layout of shelves within each department. i.e. the relative location of the departments remains the same as mentioned above.

### 6.3. Optimal arrangement of shelves

Initially, to increase the number of paths between shelves, we used many “spaced out” shelves and our layout is shown in Figure 12. After running the simulation, we realised that our net loss was higher than the current store floor plan at $4769. Furthermore, we note from the shopper density grid that many of the small aisles that we placed were not being used by shoppers. This confirms that we cannot have completely ‘spaced out’ shelves as explained above.

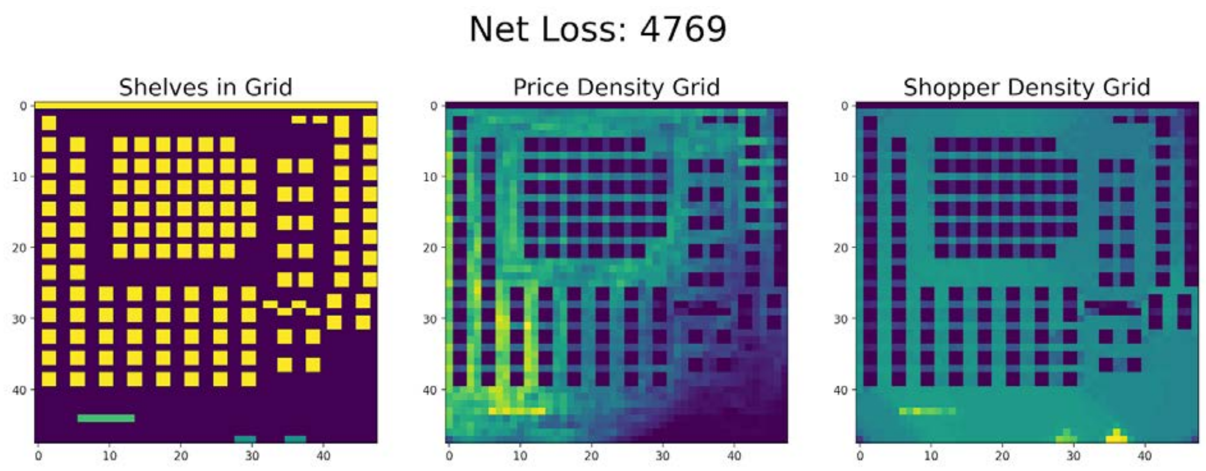


Figure 12: 1st proposed layout

By combining some "spaced out" shelves together, we were able to create large open spaces between shelves while still ensuring that there were still multiple paths to the shelves (Fig. 13). After running the simulation, we realised that the damage suffered decreased significantly to \$4226.

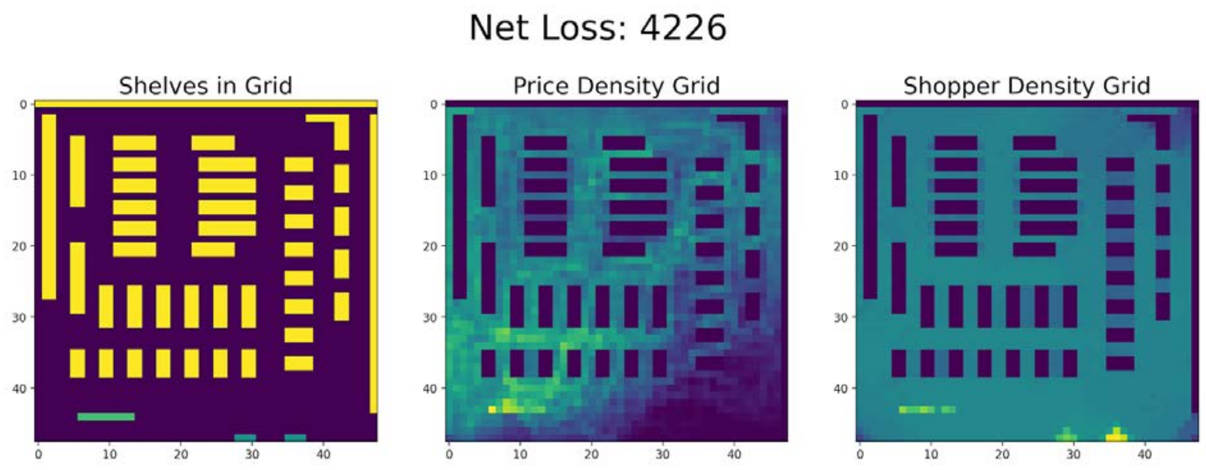


Figure 13: 2nd proposed layout

### 6.4. Optimal position of cashier counters

As the position of cashier counters determines the path shoppers take after obtaining their desired products, the cashier position should be placed such that shoppers have many routes which they can take to reach it. This decreases the density of shoppers along each path, reducing the probability of collision. From our simulation, the position of the counter should be moved 7m to the right to incur the least cost (Fig. 14).

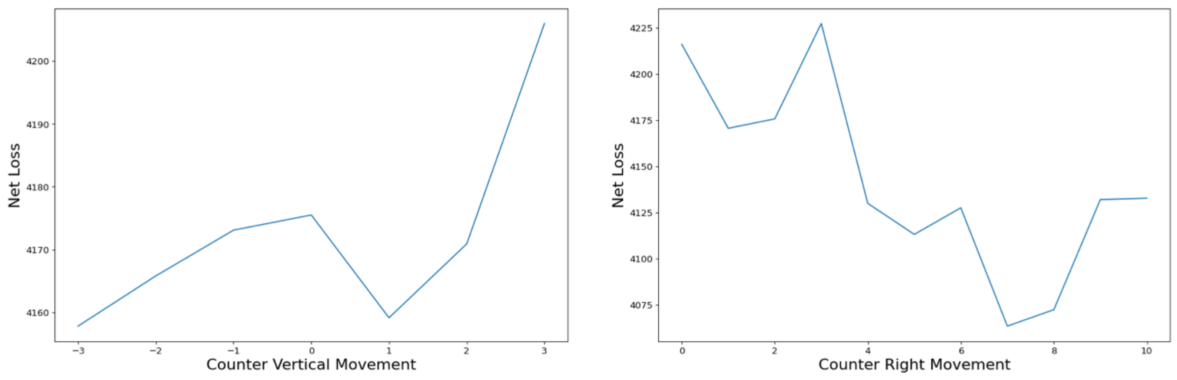


Figure 14 &15: Graph of damage against vertical and horizontal positions of cashier counter

As there is minimal difference in the damage incurred when counters are moved up and down (with the exception of the counter being especially close to the shelf above it where damage incurred increased significantly), the vertical position of the cashier counter can remain the same (Fig. 15). The spread and number of counters does not significantly decrease the damage incurred (refer to Annex), thus we do not change the spread and number of counters.

Therefore, our final proposed layout as shown in Figure 16 and 17 only incurs a damage of $4129.

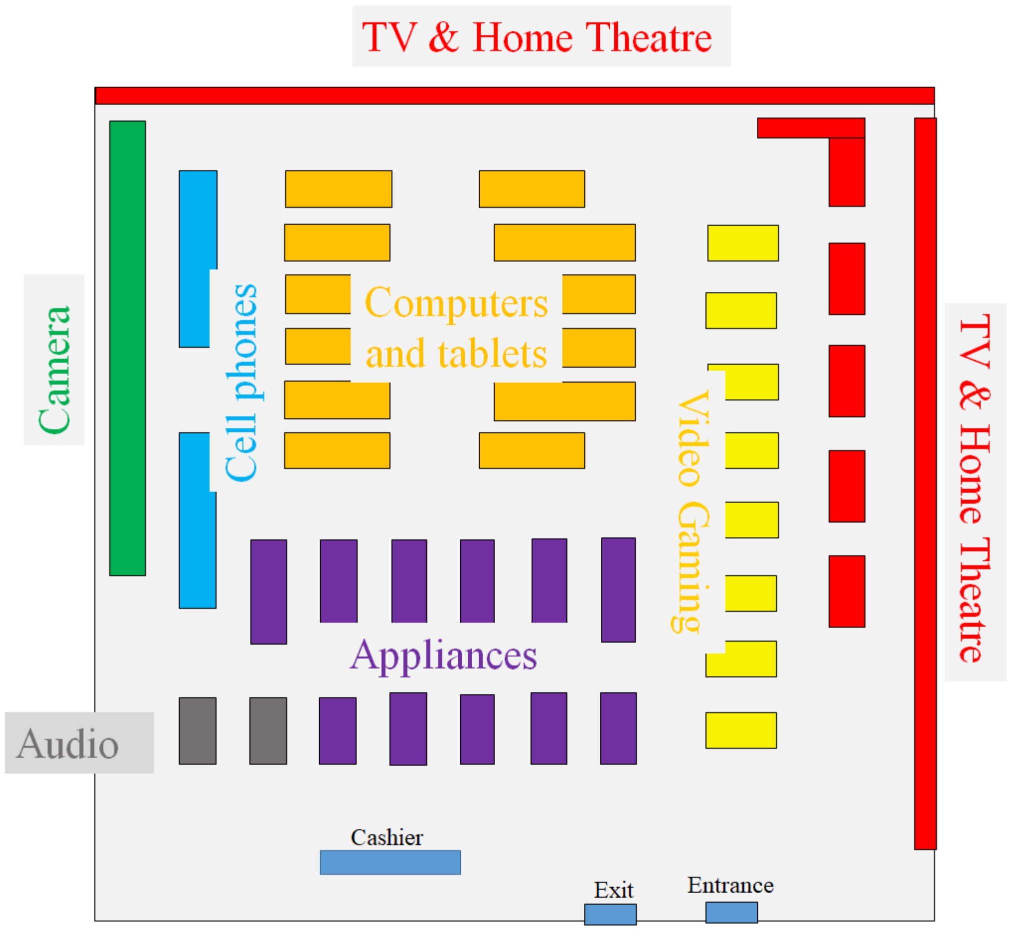


Figure 16: Final proposed layout

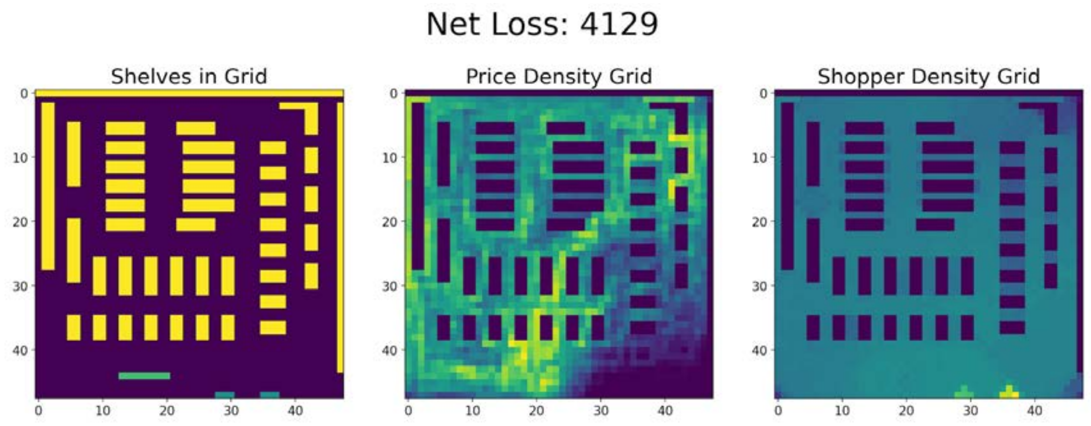


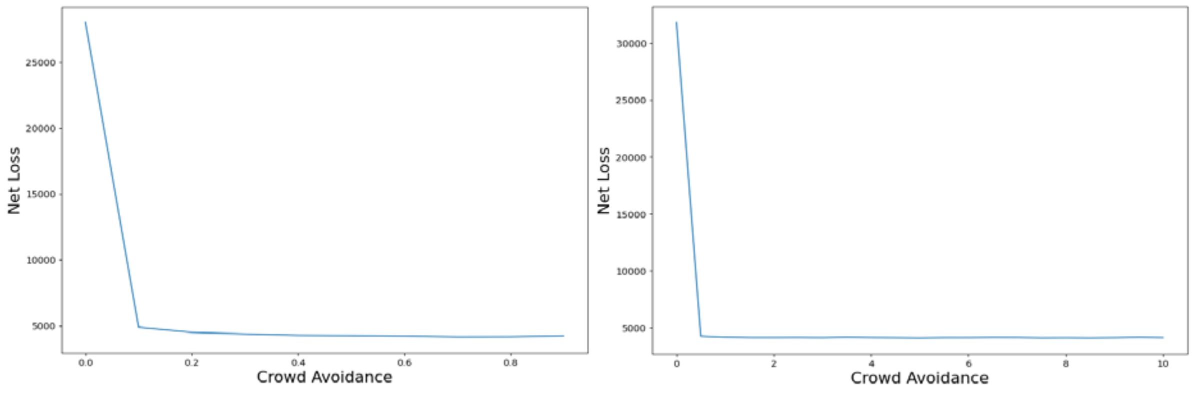
Figure 17: Damage results from the final proposed layout

## Section 7: Sensitivity Analysis

In this section, we varied some key parameters to test the response of the model to different scenarios.

### 7.1. Crowd avoidance of shoppers

This variable is varied within the domain of [0,10], with the baseline value being 0.5.



Figures 18&19: Graphs of damage against crowd avoidance

From Figures 18 and 19, it can be seen that with any positive value of crowd avoidance, loss incurred decreases to about 4000-5000 and remains in that range despite further increases in crowd avoidance. The high loss incurred when crowd avoidance is 0 is due to shoppers ignoring other shoppers that may be in their way (Table A4.1). Shoppers thus walk into each other and collide more frequently. However, once there is at least a small amount of crow avoidance, shoppers quickly spread themselves out and shopper density becomes nearly homogenous. This results in a lower loss than if there was no crowd avoidance. However, due to the degree of homogeneity of the shopper density after there is a small amount of crowd avoidance, there is no benefit from further increases in crowd avoidance.

### 7.2. Maximum shelf capacity – number of products

The maximum shelf capacity in units of products was varied within domain [3, 500], with baseline value of 150.

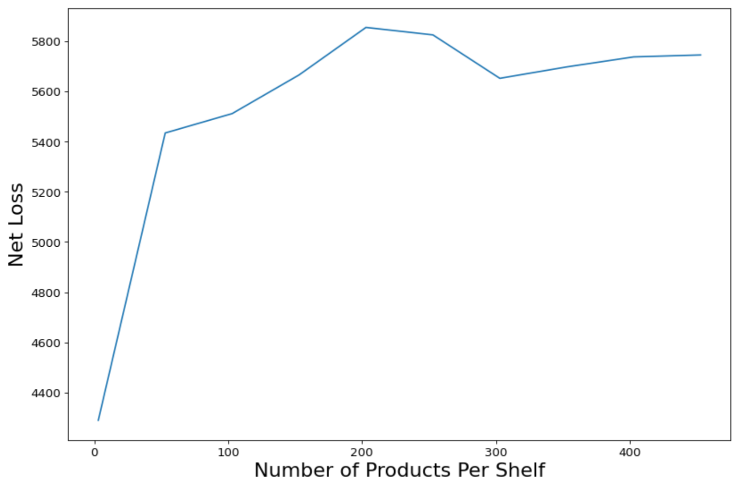


Figure 20: Graph of damage against number of products per shelf

With increasing max. quantity of products per shelf, shoppers crowd around shelves more (Table A4.2), this results in a greater rate of collisions between shoppers and between shoppers and shelves, leading to higher damages (Fig. 20). However, past a certain point, the limiting factor for products being placed into shelves is the shelf capacity (size). Thus subsequent increases in the maximum number of products per shelf have no impact on damage incurred.

## 7.3. Maximum shelf capacity – size on shelf

Maximum shelf capacity in terms of total size of items was varied in the domain [150,500], with baseline value of 150.

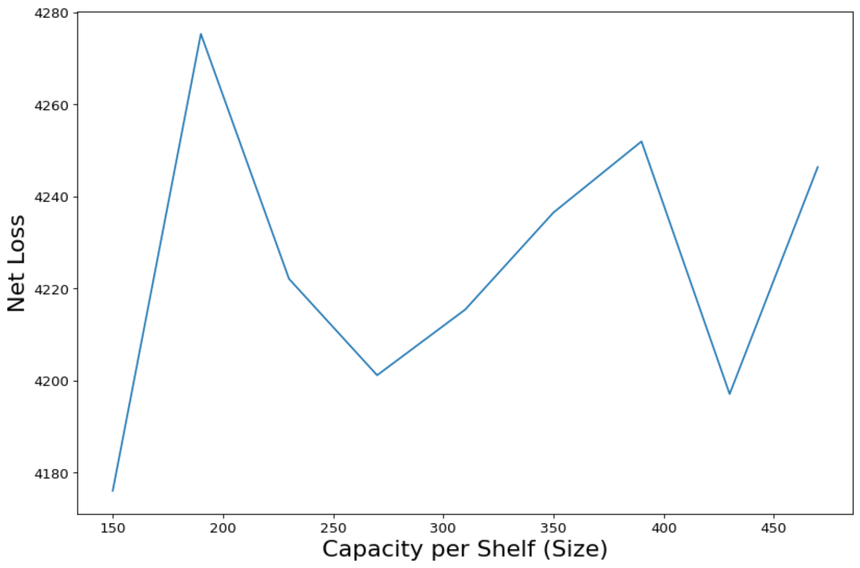


Figure 21: Graph of damage against capacity of shelves

Total loss seems to be relatively independent with respect to the shelf capacity (Table A4.3 and Fig. 21). This is because the shelves are largely limited by the number of products per shelf. Thus, increasing the capacity of shelves without increasing the maximum number of products per shelf does not reduce loss.

## 7.4. Shopper carrying capacity

The maximum carrying capacity of a shopper and their products was varied in domain [160,440], with baseline value of 300.

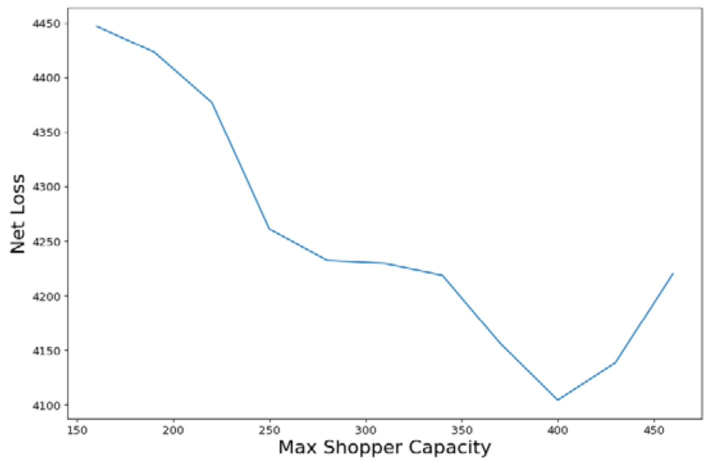


Figure 22: Graph of damage against maximum shopper carrying capacity

As shoppers' carrying capacity increases, the number of products each shopper buys rises, thus the number of shoppers which will enter the shop falls (Fig. 22), thus we see less collisions and less damage to products (Table A4.4). However, past a certain point (400), shoppers walk around the store for a longer time, resulting in higher damages.

## 7.5. Shopper size

The size of shoppers was varied in the domain [10, 190] with baseline value of 50.

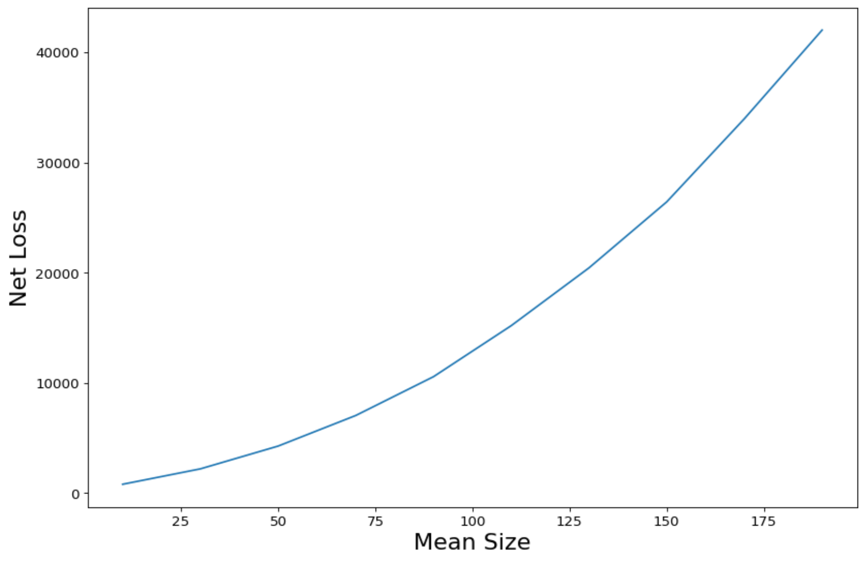


Figure 23: Graph of damage against the mean size of shoppers.

With higher size, probability of collisions increases at a rate proportional to the square of the mean size (Fig. 23). This is consistent with our equation to calculate the probability of collision between shoppers (Section 2.3).

## <u>Conclusion</u>

By considering different store layout factors as well as the popularity of items sold in the store, we have come up with a mathematical model which not only allows us to determine the optimal location of products and departments based on a given floor plan, but it also allowed us to come up with an improved floor plan. We also conducted sensitivity analysis and further ascertained that it is a model which is logical and robust.

## **Letter**

To the owner of IMMC Electronic and Appliance Store:

Dear Sir/Madam,

After modelling the expected density of shoppers in the store and the popularity of the items you sell, we were able to predict the behaviour of shoppers during a flash sale. In the process, we considered 4 layout factors: the number of shelves, the arrangement of departments, the type and layout of shelves and the positioning of the cashier counter. Our recommended floor plan to incur the least damage is attached below.

Firstly, we recommend minimising the number of empty shelves used and to use shelves which are spaced out and not too long. Both of which allow large open spaces to be present between shelves while still ensuring that shoppers have multiple routes of movement around the shelves. By doing so, the density of shoppers along each route is reduced, lessening the likelihood of collisions occurring.

Secondly, we recommend that your departments be placed in the layout shown below, where the most popular departments are spread out. For placement of specific sale products on the shelves, we recommend spacing out the most popular products. This lessens the crowd density around the most popular products and most popular departments, reducing the probability of collision. With fewer collisions, the products are less likely to be damaged, hence minimising the loss incurred.

Lastly, using our mathematical model, we determined a better location for the cashier counters. The cashier position in our proposed layout is such that shoppers have many routes which they can take to reach it. Similarly, this decreases the density of shoppers along each possible path, reducing the probability of collision and minimising loss.

Besides the layout, some other recommendations we have are to:

1. Secure the shelves to the floor to eliminate the chances of shelves toppling and causing damage and injuries to products and shoppers respectively.
2. For items that cost more than $100, we recommend showing discounts in absolute values, whereas for items that cost less than that, a percentage discount would be more attractive. This helps to maximise the quantity of items bought and make the most earnings.

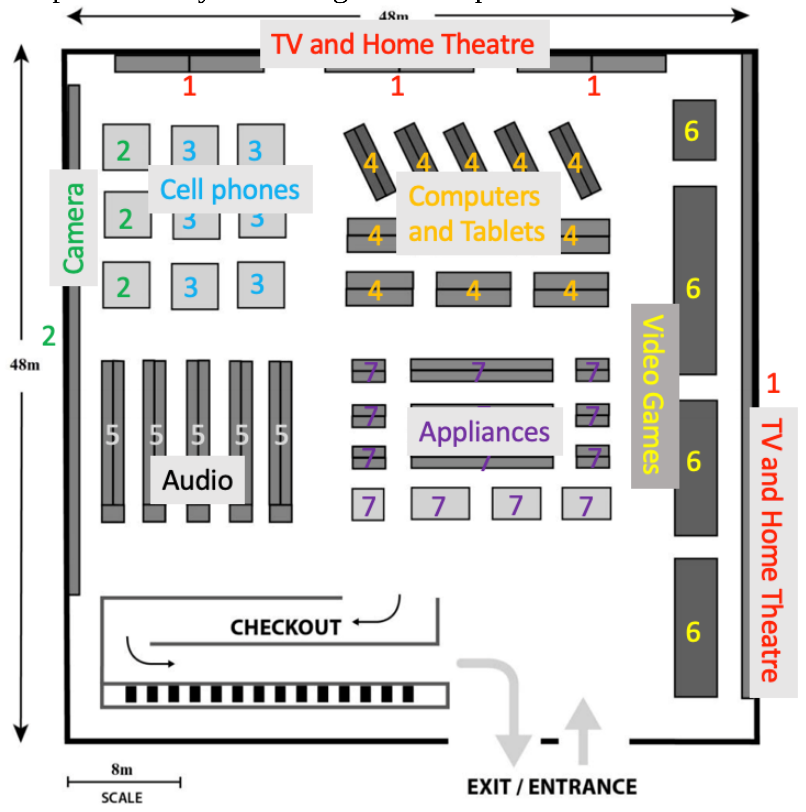
We hope that our suggestions will be helpful and we wish you all the best for your opening and flash sales.

Best regards,

Bethany, Brandon, Rachel, Xin Yao

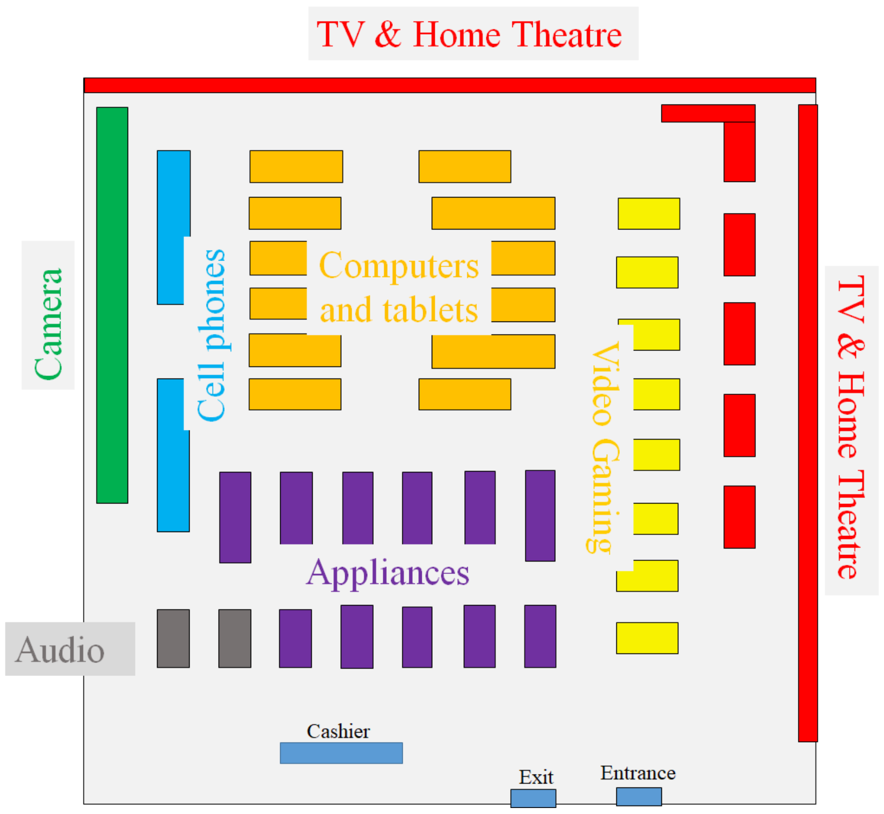
**Floor Plan Diagrams**

Requirement 2c: Department layout of original floor plan



Store Floor Plan

Requirement 2d: Proposed layout



**Annex**

Table A1: Extensions of Raw Data

| Name | Brand Rating | Percentage Usage | Size ($m^3$) | index_size |
|---|---|---|---|---|
| 40" 1080p Smart LED HDTV, 5 Series | 4.623809524 | 0.95 | 0.029812826 | 2.012689585 |
| 2-in-1 11.6" Touch-Screen Chromebook, Intel Celeron, 4GB RAM, 32GB | 4.5 | 0.78 | 0.00160599 | 0.108421768 |
| 2-in-1 12.2" Touch-Screen Chromebook, Intel Celeron, 4GB RAM, 32G | 4.623809524 | 0.78 | 0.00160599 | 0.108421768 |
| 2-in-1 14" Touch-Screen Chromebook, Intel Core i3, 4GB RAM, 128GB | 4.58 | 0.78 | 0.00160599 | 0.108421768 |
| 2-in-1 14" Touch-Screen Chromebook, Intel Core i3, 8GB RAM, 64GB eMMC Fla | 4.631578947 | 0.78 | 0.00160599 | 0.108421768 |
| 2-in-1 11.6" Touch-Screen Chromebook, 4GB RAM, 32GB eMMC Flash Mem | 4.4 | 0.78 | 0.00160599 | 0.108421768 |
| 2-in-1 13.3" 8GB RAM 256GB Flash Memory | 4.623809524 | 0.78 | 0.00160599 | 0.108421768 |
| 2-in-1 11.6" 4GB RAM 32GB Flash Memory | 4.58 | 0.78 | 0.00160599 | 0.108421768 |
| 2-in-1 14" Touch-Screen Laptop, Intel Core i5, 8GB RAM, 256GB S | 4.631578947 | 0.78 | 0.00160599 | 0.108421768 |
| 2-in-1 11.6" Touch-Screen Laptop, Intel Pentium, 4GB RAM, 128GB | 4.631578947 | 0.78 | 0.00160599 | 0.108421768 |
| 2-in-1 15.6" 4K Ultra HD Touch-Screen Laptop, Intel Core i7, 16GB | 4.631578947 | 0.78 | 0.00160599 | 0.108421768 |
| 43" 4K UHD HDR Smart LED TV, 6 Series | 4.623809524 | 0.95 | 0.029812826 | 2.012689585 |

| | | | | |
|---|---|---|---|---|
| 50" 4K UHD HDR Smart LED TV, 7 Series | 4.623809524 | 0.95 | 0.047288851 | 3.192511117 |
| 50" 4K UHD HDR Smart LED TV, NU6900 Series | 4.623809524 | 0.95 | 0.047288851 | 3.192511117 |
| 55" 4K UHD HDR Smart LED TV, NU6900 Series | 4.623809524 | 0.95 | 0.047288851 | 3.192511117 |
| 55" 4K UHD HDR Smart LED TV, X800G Series | 4.769230769 | 0.95 | 0.047288851 | 3.192511117 |
| 50" 4K UHD HDR Smart LED Roku TV | 4.6 | 0.95 | 0.047288851 | 3.192511117 |
| 55" 4K UHD HDR Smart LED Roku TV, 4 Series | 4.6 | 0.95 | 0.047288851 | 3.192511117 |
| 55" 4K UHD HDR Smart LED TV, UK6090PUA Series | 4.533333333 | 0.95 | 0.047288851 | 3.192511117 |
| 65" 4K UHD HDR Smart LED TV, NU6900 Series | 4.623809524 | 0.95 | 0.073433795 | 4.957578881 |
| 65" 4K UHD HDR Smart LED TV, 7 Series | 4.623809524 | 0.95 | 0.073433795 | 4.957578881 |
| 65" 4K UHD HDR Smart LED TV, X800G Series | 4.769230769 | 0.95 | 0.073433795 | 4.957578881 |
| 65" 4K UHD HDR Smart LED TV, X900F Series | 4.769230769 | 0.95 | 0.073433795 | 4.957578881 |
| 65" 4K UHD HDR Smart LED Roku TV, 4 Series | 4.6 | 0.95 | 0.073433795 | 4.957578881 |
| 65" 4K UHD HDR Smart LED TV, H6500F Series | 4.3 | 0.95 | 0.073433795 | 4.957578881 |
| 75" 4K UHD HDR Smart LED TV, NU6900 Series | 4.623809524 | 0.95 | 0.098117873 | 6.624022282 |
| 70" 4K UHD HDR Smart LED TV, 6 Series | 4.623809524 | 0.95 | 0.098117873 | 6.624022282 |
| 75" 4K UHD HDR LED Smart TV, X800G Series | 4.769230769 | 0.95 | 0.098117873 | 6.624022282 |
| 85" 4K UHD HDR Smart LED TV, X900F Series | 4.769230769 | 0.95 | 0.153331477 | 10.35154032 |
| 55" 4K UHD HDR Smart OLED TV, A8G Series | 4.769230769 | 0.95 | 0.047288851 | 3.192511117 |

| | | | | |
|---|---|---|---|---|
| 65" 4K UHD HDR Smart OLED TV, A8G Series | 4.769230769 | 0.95 | 0.073433795 | 4.957578881 |
| 65" 4K UHD HDR Smart QLED TV, Q70 Series | 4.623809524 | 0.95 | 0.073433795 | 4.957578881 |
| 65" 4K UHD HDR Smart QLED TV, Q60 Series | 4.623809524 | 0.95 | 0.073433795 | 4.957578881 |
| 65" 4K UHD HDR Smart QLED TV, Q80 Series | 4.623809524 | 0.95 | 0.073433795 | 4.957578881 |
| 75" 4K UHD HDR Smart QLED TV, Q70 Series | 4.623809524 | 0.95 | 0.098117873 | 6.624022282 |
| 75" 4K UHD HDR Smart QLED TV, Q60 Series | 4.623809524 | 0.95 | 0.098117873 | 6.624022282 |
| 32" 720p LED HDTV | 4.533333333 | 0.95 | 0.029812826 | 2.012689585 |
| 32" 720p Smart LED HDTV Roku TV, 3 Series | 4.6 | 0.95 | 0.029812826 | 2.012689585 |
| 32" LED 720p Smart TV, H5500 Series | 4.3 | 0.95 | 0.029812826 | 2.012689585 |
| 23.8" Touch-Screen All-in-One, Intel Core i5, 12GB RAM, 256GB SSD | 4.631578947 | 0.89 | 0.013165918 | 0.888842474 |
| 27" Touch-Screen All-in-One, Intel Core i7, 12GB RAM, 256GB SSD | 4.631578947 | 0.89 | 0.013165918 | 0.888842474 |
| 23.8" Touch-Screen All-in-One, AMD Ryzen 3-Series, 8GB Memory, 256GB | 4.4 | 0.89 | 0.013165918 | 0.888842474 |
| Wireless All-in-One Printer | 4.6 | 0.82 | 0.016121347 | 1.088366056 |
| Wireless Color All-in-One Printer | 4.6 | 0.82 | 0.016121347 | 1.088366056 |
| Wireless All-in-One Printer | 4.6 | 0.82 | 0.016121347 | 1.088366056 |
| Wireless All-in-One Printer | 4.631578947 | 0.82 | 0.016121347 | 1.088366056 |
| Wireless All-in-One Instant Ink Ready Printer | 4.631578947 | 0.82 | 0.016121347 | 1.088366056 |

| | | | | |
|---|---|---|---|---|
| Color Wireless All-in-One Printer | 4.631578947 | 0.82 | 0.016121347 | 1.088366056 |
| Streaming 4K Ultra HD Audio Wi-Fi Built-In Blu-Ray Player | 4.623809524 | 0.44 | 0.01179878 | 0.796545852 |
| Streaming 4K Ultra HD Hi-Res Audio Wi-Fi Built-In Blu-Ray Player | 4.769230769 | 0.44 | 0.01179878 | 0.796545852 |
| Streaming 4K Ultra HD Hi-Res Audio Wi-Fi Built-In Blu-Ray Player | 4.769230769 | 0.44 | 0.01179878 | 0.796545852 |
| 4K Ultra HD Blu-Ray Player | 4.533333333 | 0.44 | 0.01179878 | 0.796545852 |
| Streaming Audio Wi-Fi Built-In Blu-Ray Player | 4.533333333 | 0.44 | 0.01179878 | 0.796545852 |
| Streaming Audio Blu-Ray Player | 4.533333333 | 0.44 | 0.01179878 | 0.796545852 |
| DSLR Camera, Body Only, Black | 4.866666667 | 0.62 | 0.000764555 | 0.051615758 |
| DSLR Camera, Body Only, Black | 4.6 | 0.62 | 0.000764555 | 0.051615758 |
| DSLR Two Lens Kit with AF-P DX NIKKOR 18-55mmf/3.5-5.6G VR & | 4.866666667 | 0.62 | 0.000764555 | 0.051615758 |
| DSLR Two Lens Kit with 18-55mm and 70-300mm Lenses, Black | 4.866666667 | 0.62 | 0.000764555 | 0.051615758 |
| DSLR Camera with 18-55mm IS STM Lens, Black | 4.6 | 0.62 | 0.000764555 | 0.051615758 |
| DSLR Two Lens Kit with EF-S 18-55mm IS II and EF 75-300m | 4.6 | 0.62 | 0.000764555 | 0.051615758 |
| Mirrorless Camera Two Lens Kit with 16-50mm and 55-210mm Le | 4.769230769 | 0.62 | 0.000764555 | 0.051615758 |
| Full-Frame Mirrorless Camera with 28-70mm Lens, Black | 4.769230769 | 0.62 | 0.000764555 | 0.051615758 |

| | | | | |
|---|---|---|---|---|
| Mirrorless Camera with FE 28-70mm F3.5-5.6 OSS Lens | 4.769230769 | 0.62 | 0.000764555 | 0.051615758 |
| Mirrorless Camera with Lens | 4.6 | 0.62 | 0.000764555 | 0.051615758 |
| 11.6" Chromebook, Intel Atom x5, 2GB Ram, 16GB eMMC Flash Memory | 4.623809524 | 0.78 | 0.00160599 | 0.108421768 |
| 11.6" Chromebook, Intel Atom x5, 4GB Memory, 32GB eMMC Flash Memo | 4.623809524 | 0.78 | 0.00160599 | 0.108421768 |
| 1TB Fortnite Neo Versa Console Bundle - Jet Black | 4.769230769 | 0.5 | 0.01538341 | 1.03854729 |
| 32GB Console - Gray Joy-Con + 2 more items | 4.6 | 0.5 | 0.01538341 | 1.03854729 |
| 1TB Star Wars Jedi: Fallen Order Deluxe Edition Console Bundle | 4.733333333 | 0.5 | 0.01538341 | 1.03854729 |
| 1TB NBA 2K20 Bundle - Black | 4.733333333 | 0.5 | 0.01538341 | 1.03854729 |
| Desktop, Intel Core i7, 8GB RAM, 256GB SSD | 4.631578947 | 0.89 | 0.013165918 | 0.888842474 |
| Intel Core i7 9700, 16GB RAM, NVIDIA GeForce GTX 1660 Ti, | 4.631578947 | 0.89 | 0.013165918 | 0.888842474 |
| 24" Tall Tub Built-In Dishwasher, Monochromatic Stainless Steel | 4.475 | 0.49 | 0.33069095 | 22.32523142 |
| 24" Front Control Tall Tub Built-In Dishwasher, Stainless Steel | 4.3 | 0.49 | 0.33069095 | 22.32523142 |
| 7.0cu ft 13-Cycle Electric Dryer, White | 4.475 | 0.68 | 0.5136853 | 34.67933792 |
| 7.2cu ft 3-Cycle Electric Dryer, White | 4.45 | 0.68 | 0.5136853 | 34.67933792 |
| 7.3cu ft 8-Cycle Electric Dryer, White | 4.533333333 | 0.68 | 0.5136853 | 34.67933792 |

| | | | | |
|---|---|---|---|---|
| 7.4cu ft 10-Cycle Smart Wi-Fi Enabled Electric Dryer, White | 4.533333333 | 0.68 | 0.5136853 | 34.67933792 |
| Gamer Supreme Liquid Cool Gaming Desktop, AMD Ryzen 7 3700X | 4.9 | 0.89 | 0.046047157 | 3.108683303 |
| Gamer Master Gaming Desktop, AMD Ryzen 5 3600, 8GB Memory | 4.9 | 0.89 | 0.046047157 | 3.108683303 |
| Gamer Master Gaming Desktop, AMD Ryzen 3 2300X, 8GB Memory | 4.9 | 0.89 | 0.046047157 | 3.108683303 |
| Gaming Desktop, Intel Core i5-9400F, 8GB RAM, NVIDIA GeForce G | 4.7 | 0.89 | 0.046047157 | 3.108683303 |
| Gaming Desktop, Intel Core i7-9700K, 16GB RAM, NVIDIA GeForce | 4.7 | 0.89 | 0.046047157 | 3.108683303 |
| 15.6" Gaming Laptop, Intel Core i5, 8GB RAM, NVIDIA GeForce GTX 1650, 51 | 4.5 | 0.78 | 0.002115212 | 0.142799775 |
| 17.3" Gaming Laptop, Intel Core i7, 16GB RAM, NVIDIA GeForce GTX 1660 T | 4.5 | 0.78 | 0.002115212 | 0.142799775 |
| 15.6" Gaming Laptop, AMD Ryzen 5, 8GB Ram, NVIDIA GeForce GTX 1050, 25 | 4.631578947 | 0.78 | 0.002115212 | 0.142799775 |
| 15.6" Gaming Laptop, Intel Core i7, 32GB RAM, NVIDIA GeForce RTX 2060, 5 | 4.6 | 0.78 | 0.002115212 | 0.142799775 |
| Wireless Wearable Speaker - Black | 4.7 | 0.16 | 0.000121762 | 0.008220287 |
| Wireless Noise Cancelling Earbud Headphones - Graphite | 4 | 0.16 | 0.000121762 | 0.008220287 |
| Wireless Bluetooth Headset - Black | 4.533333333 | 0.16 | 0.000121762 | 0.008220287 |

| | | | | |
|---|---|---|---|---|
| 28" LED 4K UHD Monitor, UE590 Series | 4.623809524 | 0.5 | 0.068763781 | 4.642302183 |
| 24" LED FHD Monitor, Black | 4.5 | 0.5 | 0.068763781 | 4.642302183 |
| 27" LED QHD G-Sync Monitor, Black | 4.58 | 0.5 | 0.068763781 | 4.642302183 |
| 20.7" LED FHD Monitor | 4.631578947 | 0.5 | 0.068763781 | 4.642302183 |
| 27" IPS LED FHD FreeSync Monitor, 27f | 4.631578947 | 0.5 | 0.068763781 | 4.642302183 |
| 31.5" IPS LED FHD Monitor | 4.631578947 | 0.5 | 0.068763781 | 4.642302183 |
| 32" LED QHD Monitor | 4.631578947 | 0.5 | 0.068763781 | 4.642302183 |
| 1.6cu ft Over-the-Range Microwave, Black on Stainless | 4.6 | 0.93 | 0.135599677 | 9.154451217 |
| 1.6cu ft Over-the-Range Microwave, Stainless Steel | 4.45 | 0.93 | 0.135599677 | 9.154451217 |
| 15" 16GB RAM 256GB Solid State Drive | 4.623809524 | 0.78 | 0.00160599 | 0.108421768 |
| 15.6" Touch-Screen Laptop, Intel Core i5, 8GB Ram, 256GB SSD | 4.58 | 0.78 | 0.00160599 | 0.108421768 |
| 15.6" Touch-Screen Laptop, Intel Core i3, 8GB Ram, 128GB SSD | 4.58 | 0.78 | 0.00160599 | 0.108421768 |
| 14" Laptop, AMD A9 Series, 4GB Ram, AMD Radeon R5, 128GB SSD, WIndows | 4.631578947 | 0.78 | 0.00160599 | 0.108421768 |
| 17.3" Laptop, Intel Core i5, 8GB Memory, 256GB SSD, Jet Black, Maglia Pattern | 4.631578947 | 0.78 | 0.00160599 | 0.108421768 |
| 2-in-1 15.6" Touch-Screen Laptop, Intel Core i7, 12GB RAM, 512GB S | 4.631578947 | 0.78 | 0.00160599 | 0.108421768 |

| | | | | |
|---|---|---|---|---|
| 11.4" Laptop, AMD A6 Series, 4GB Ram, AMD Radeon R4, 65GB e | 4.4 | 0.78 | 0.00160599 | 0.108421768 |
| 13.5" 8GB RAM 256GB Solid State Drive | 4.733333333 | 0.78 | 0.00160599 | 0.108421768 |
| 30" Built-In Single Electric Wall Oven, Stainless Steel | 4.475 | 0.9 | 0.53880666 | 36.37530261 |
| 5.1cu ft Freestanding Gas Range, Stainless Steel | 4.475 | 0.9 | 0.53880666 | 36.37530261 |
| 5.3cu ft Slide-In Electric Range, Stainless Steel | 4.45 | 0.9 | 0.53880666 | 36.37530261 |
| 5.0cu ft Freestanding Gas Range, Stainless Steel | 4.45 | 0.9 | 0.53880666 | 36.37530261 |
| 6.3cu ft Slide-In Electric Range with ProBake Convection, Stainless Steel | 4.533333333 | 0.9 | 0.53880666 | 36.37530261 |
| 30" Combination Double Electric Convection Wall Oven with Built-In Microwave | 4.533333333 | 0.9 | 0.53880666 | 36.37530261 |
| 24.7cu ft French Door Refrigerator, Black Stainless Steel | 4.475 | 0.99 | 1.4812431 | 100 |
| 26.8cu ft French Door Refrigerator, Stainless Steel | 4.475 | 0.99 | 1.4812431 | 100 |
| 25.1cu ft Side-by-Side Refrigerator, Fingerprint Resistant, Stainless Steel | 4.45 | 0.99 | 1.4812431 | 100 |
| 27.8cu ft 4 Door French Door Refrigerator, PrintProof, InstaView Door-in-Door, Stainless | 4.533333333 | 0.99 | 1.4812431 | 100 |
| 26.2cu ft French Door Smart Wi-Fi Enabled Refrigerator, PrintProof, Black Stainless | 4.533333333 | 0.99 | 1.4812431 | 100 |
| App-Controlled Robot Vacuum | 4.35 | 0.43 | 0.012742581 | 0.860262638 |

| | | | | |
|---|---|---|---|---|
| App-Controlled Self-Charging Robot Vacuum | 4.5 | 0.43 | 0.012742581 | 0.860262638 |
| App-Controlled Self-Charging Robot Vacuum | 4.4 | 0.43 | 0.012742581 | 0.860262638 |
| App-Controlled Robot Vacuum | 4.4 | 0.43 | 0.012742581 | 0.860262638 |
| Bagless Cordless Pet Handheld/Stick Vacuum | 4.35 | 0.43 | 0.1635429 | 11.04092232 |
| 10.1" Tablet, 32GB | 3.7 | 0.52 | 0.000417673 | 0.028197498 |
| 12.3" Tablet, 64GB | 4.1 | 0.52 | 0.000417673 | 0.028197498 |
| Ball Animal 2 Bagless Upright Vacuum | 4.7 | 0.43 | 0.1635429 | 11.04092232 |
| Ball Animal + Allergy Bagless Upright Vacuum | 4.7 | 0.43 | 0.1635429 | 11.04092232 |
| 4.3cu ft 12-Cycle Top-Loading Washer, White | 4.475 | 0.98 | 0.53310397 | 35.99030909 |
| 3.8cu ft 12-Cycle Top-Loading Washer, White | 4.475 | 0.98 | 0.53310397 | 35.99030909 |
| 4.2cu ft 11-Cycle Top-Loading Washer, White on White | 4.45 | 0.98 | 0.53310397 | 35.99030909 |
| 4.1cu ft 11-Cycle HE Top-Loading Washer, White | 4.4 | 0.98 | 0.53310397 | 35.99030909 |
| 3.8cu ft 12-Cycle Top-Loading Washer, White | 4.5 | 0.98 | 0.53310397 | 35.99030909 |
| Wireless Earbud Headphones | 4.5 | 0.16 | 0.000121762 | 0.008220287 |
| Sport Wireless Earbud Headphones | 4.1 | 0.16 | 0.000121762 | 0.008220287 |

Table A2: Analysis of Raw Data

| Department | Number of Units of Products Per Department | Mean Popularity |
|---|---|---|
| Appliances | 257 | 0.8389350550900967 |
| Audio | 50 | 0.7909613754768059 |

| | | |
|---|---|---|
| Cameras | 165 | 0.8028821374113138 |
| Cell Phones Number | 55 | 0.7798840389861017 |
| Computers & Tablets | 444 | 0.814878007267299 |
| TV & Home Theater | 383 | 0.8321277481983057 |
| Video Gaming | 181 | 0.8275033824709389 |

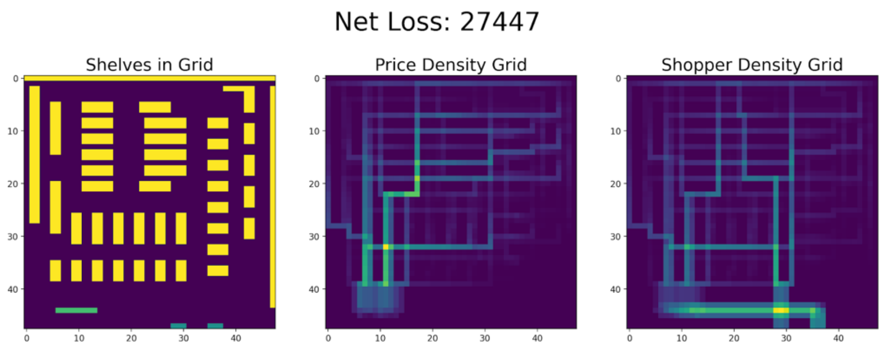
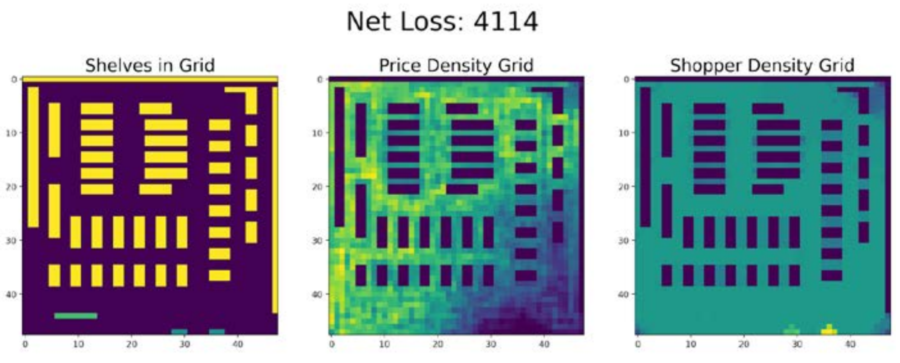
**Total Number of Units of Products: 1535**
**Global Average Popularity: 0.8213761744384138**

Table A3: Simulation of Figure 1 Layout Results

| **Position of Appliances** | **Position of TV & Home Theatre** | **Net Loss** |
|---|---|---|
| 7 | 1 | 4394.030666666669 |
| 6 | 1 | 4432.70676666667 |
| 5 | 1 | 4443.277566666663 |
| 4 | 1 | 4478.672877777776 |
| 7 | 6 | 4562.070699999998 |
| 6 | 2 | 4564.639266666681 |
| 5 | 6 | 4569.631855555563 |
| 6 | 4 | 4591.45647777777 |
| 5 | 7 | 4598.69213333332 |
| 7 | 2 | 4610.325922222224 |
| 6 | 7 | 4612.091100000018 |
| 7 | 3 | 4620.267588888878 |
| 4 | 2 | 4626.421377777772, |
| 7 | 4 | 4627.433866666661 |
| 5 | 2 | 4663.19474444445 |
| 4 | 3 | 4667.191777777775 |

| | | |
|---|---|---|
| 7 | 5 | 4673.98428888888 |
| 5 | 4 | 4699.86371111112 |
| 6 | 3 | 4711.9135666666625 |
| 6 | 5 | 4734.085344444442 |
| 5 | 3 | 4736.36588888889 |
| 4 | 5 | 4740.461699999997 |
| 4 | 7 | 4746.716955555563 |
| 4 | 6 | 4771.911155555549 |

Table A4.1: Sensitivity Analysis for Crowd Avoidance

| **Value** | **Diagram** | **Total Loss** |
|---|---|---|
| 0 |  | 27447 |
| 5.5 |  | 4114 |

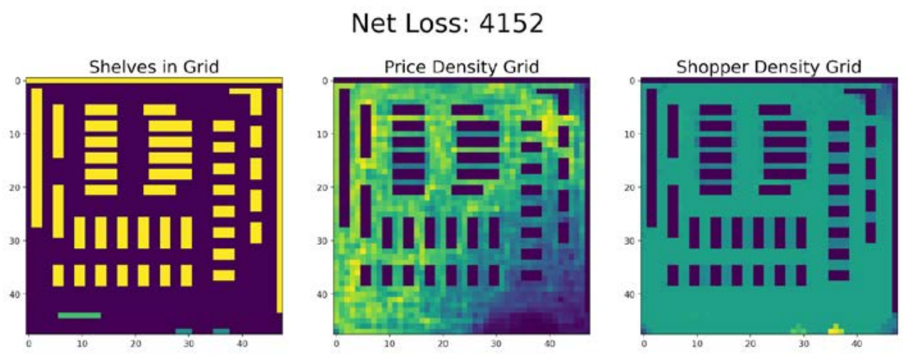
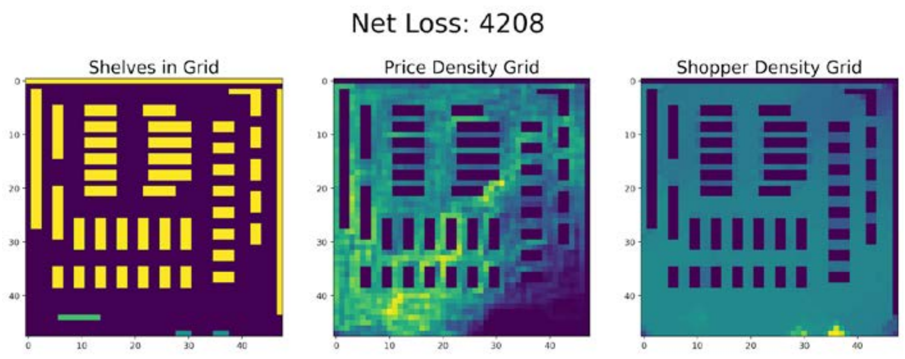
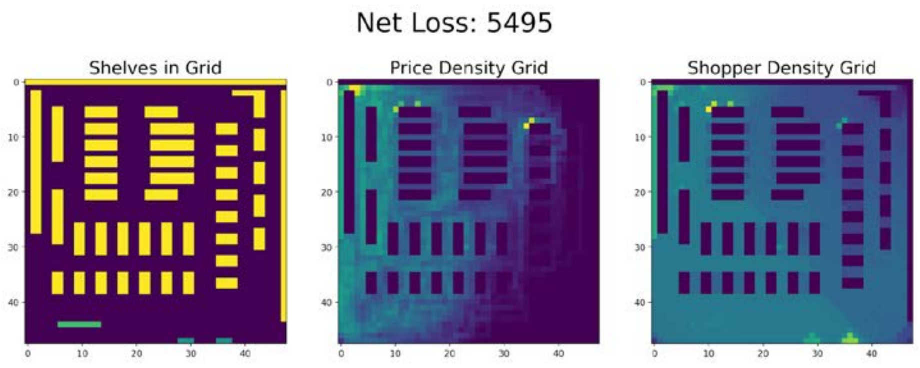
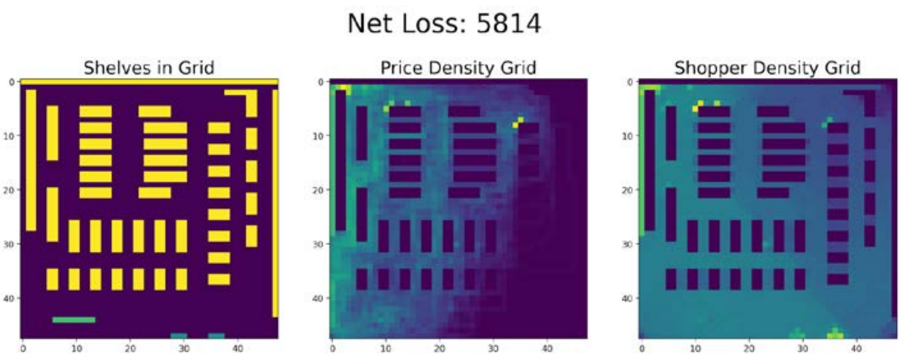
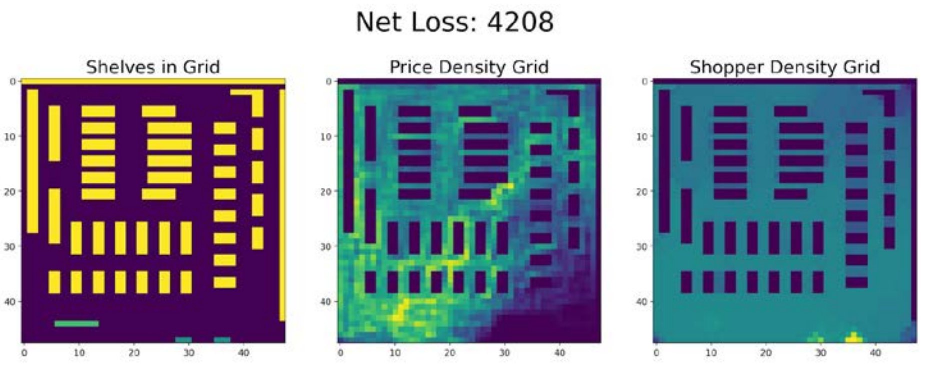
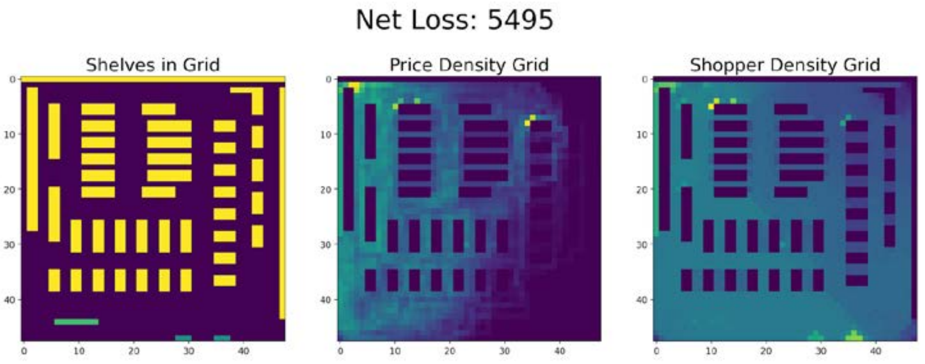
| | | |
|---|---|---|
| 10 |  | 4152 |

Table A4.2: Sensitivity Analysis for Maximum Shelf Capacity (Number of Products)

| **Value** | **Diagram** | **Total Loss** |
|---|---|---|
| 3 |  | 4208 |
| 250 |  | 5495 |

| | | |
|---|---|---|
| 500 |  | 5814 |

**<u>Table A4.3: Sensitivity Analysis for Maximum Shelf Capacity (Number of Products)</u>**

| Value | Diagram | Total Loss |
|---|---|---|
| 3 |  | 4208 |
| 250 |  | 5495 |

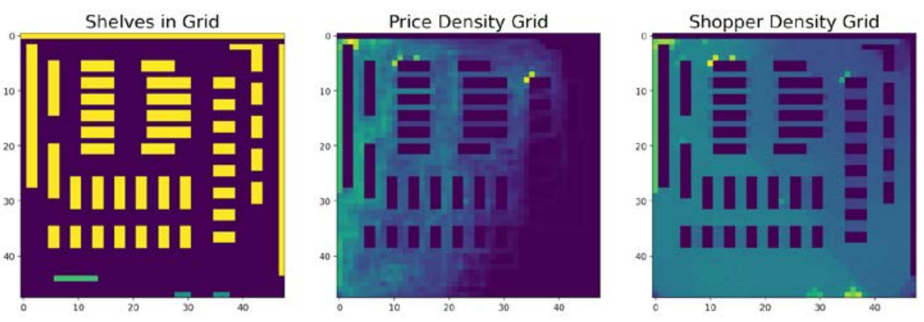
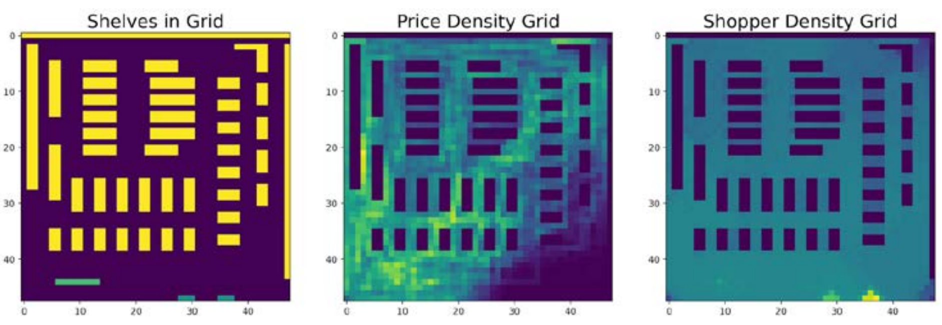
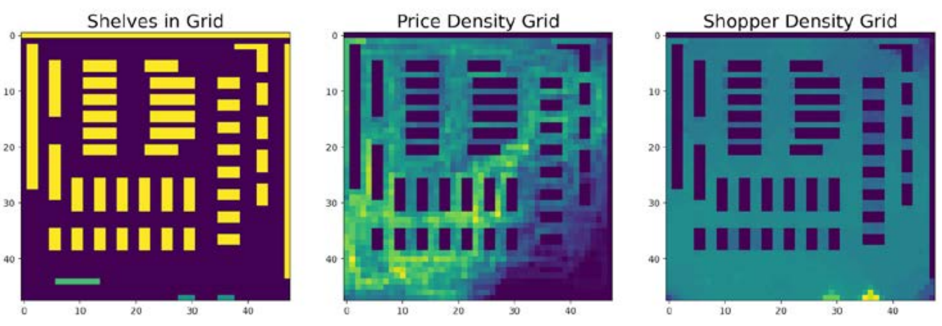
| | | |
|---|---|---|
| 500 | Net Loss: 5814  | 5814 |

Table A4.4: Sensitivity Analysis for Maximum Shelf Capacity (Size)

| Value | Diagram | Total Loss |
|---|---|---|
| 150 | Net Loss: 4233  | 4233 |
| 300 | Net Loss: 4288  | 4288 |

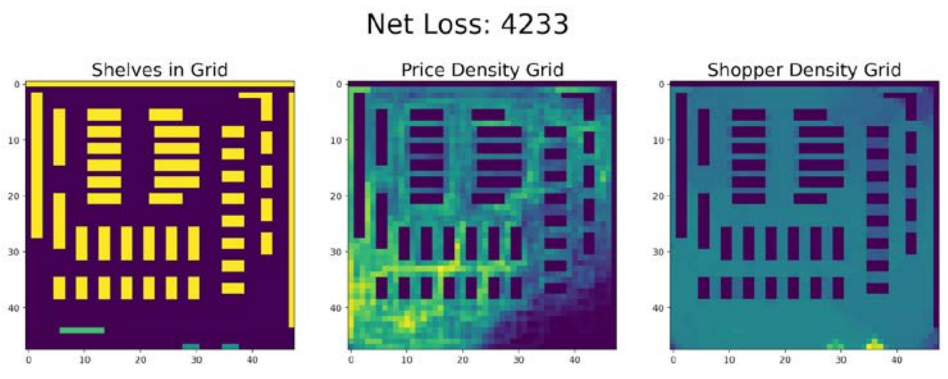
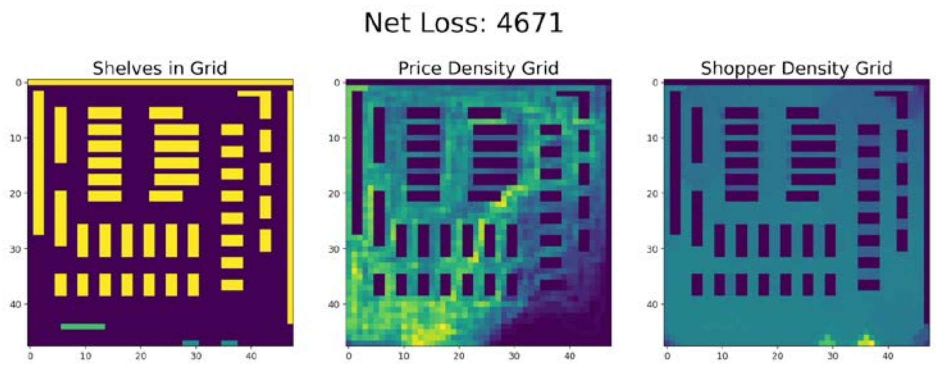
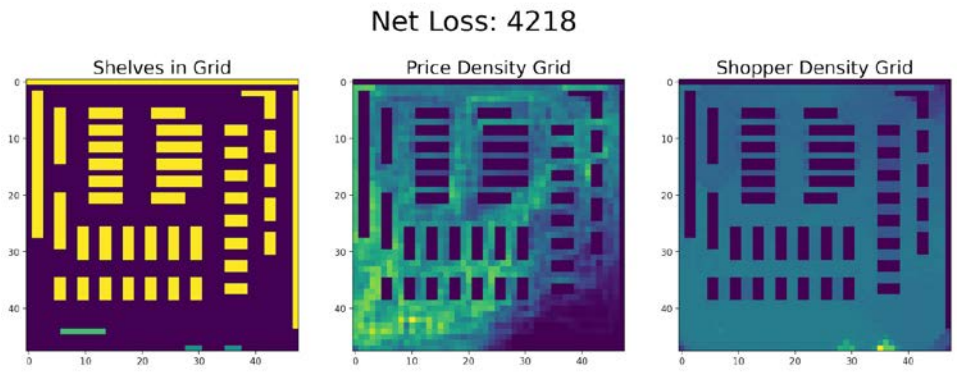
| | | |
|---|---|---|
| 500 |  | 4233 |

Table A4.5: Sensitivity Analysis for Maximum Shopper Capacity (Size)

| **Value** | **Diagram** | **Total Loss** |
|---|---|---|
| 160 |  | 4671 |
| 300 |  | 4218 |

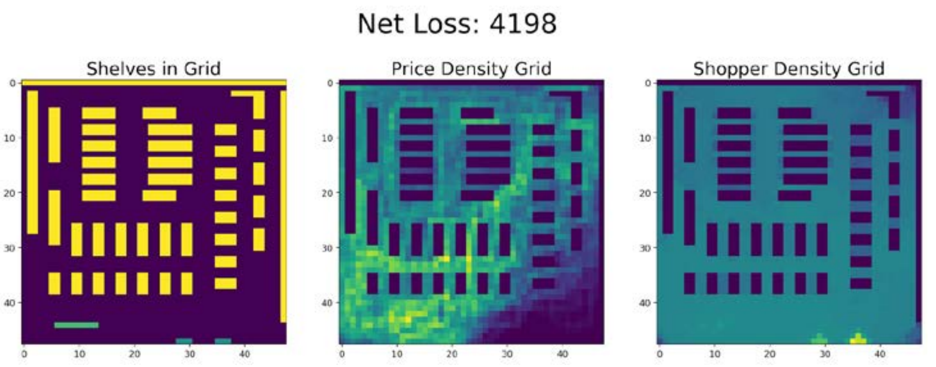
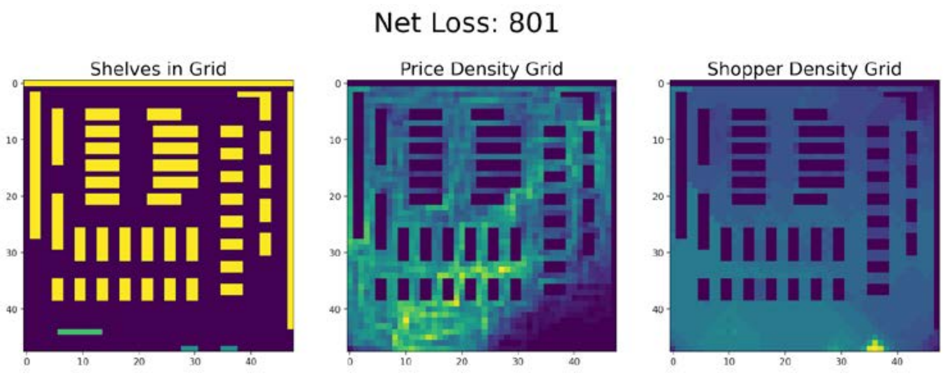
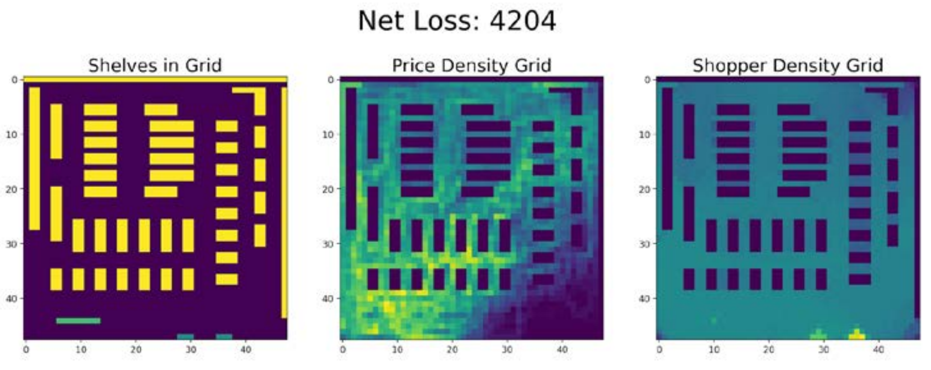
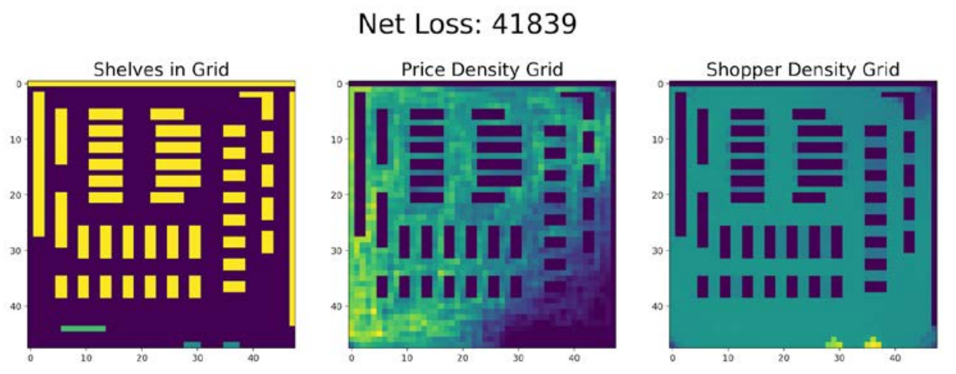
| | | |
|---|---|---|
| 440 |  | 4198 |

Table A4.5: Sensitivity Analysis for Mean Shopper Size

| **Value** | **Diagram** | **Total Loss** |
|---|---|---|
| 10 |  | 790 |
| 50 |  | 4204 |

| | | |
|---|---|---|
| 190 | Net Loss: 41839<br>Shelves in Grid<br>Price Density Grid<br>Shopper Density Grid | 41839 |

**Citations**

1. "Absolute Discount." Changingminds, http://changingminds.org/disciplines/marketing/pricing/absolute_discount.htm. Accessed March 20, 2020.
2. Rodner, Derek. "Percent Off vs. Dollar Discounts: The Psychology of Promotions." Agilenceinc, https://blog.agilenceinc.com/percent-off-vs.-dollar-discounts-the-psychology-of-promotions. Accessed March 20, 2020.
3. Simpson, Craig. "Do Percentages Sell Better than Dollar Amounts?." Entrepreneur, July 26, 2016, https://www.entrepreneur.com/article/278134. Accessed March 20, 2020.
4. Author, Become. "The behavioural economics of discounting, and why Kogan would profit from discount deception." Theconversation, May 29, 2019, http://theconversation.com/the-behavioural-economics-of-discounting-and-why-kogan-would-profit-from-discount-deception-117895. Accessed March 20, 2020.
5. "Semanticscholar." Semanticscholar, https://pdfs.semanticscholar.org/88a5/540ea635b9bd7b4ef295fde4157a51c3d54a.pdf. Accessed March 20, 2020.
6. "Kinetic theory of gases | physics | Britannica." Britannica, https://www.britannica.com/science/kinetic-theory-of-gases. Accessed March 20, 2020.
7. "Loss Aversion Theory." The Economics of Design | Interaction Design Foundation, June 03, 2016, https://www.interaction-design.org/literature/article/loss-aversion-theory-the-economics-of-design. Accessed March 20, 2020.
8. "5.1 Price Elasticity of Demand and Price Elasticity of Supply – Principles of Economics." Opentextbc.ca, https://opentextbc.ca/principlesofeconomics/chapter/5-1-price-elasticity-of-demand-and-price-elasticity-of-supply/. Accessed March 20, 2020.
9. "• Statista." The Statistics Portal for Market Data, Market Research and Market Studies, https://www.statista.com/. Accessed March 20, 2020.
10. "Prospect Theory." Economics Help, https://www.economicshelp.org/blog/glossary/prospect-theory/. Accessed March 20, 2020.
11. "Semanticscholar." Semanticscholar, https://pdfs.semanticscholar.org/88a5/540ea635b9bd7b4ef295fde4157a51c3d54a.pdf. Accessed March 20, 2020.
12. "Normalized Data / Normalization." Statistics How To, https://www.statisticshowto.datasciencecentral.com/normalized/ . Accessed March 20, 2020.
13. "A* Search Algorithm." GeeksforGeeks, https://www.geeksforgeeks.org/a-search-algorithm/ . Accessed March 20, 2020.
14. "The Effect of Income on Appliances in U.S. Households-- 2001 RECS." Eia, https://www.eia.gov/consumption/residential/data/2001/appliances/appliances.php. Accessed March 20, 2020.
15. "• UK households: ownership of dishwashers 1994-2018 | Statista." Statista, https://www.statista.com/statistics/289151/household-dishwashing-in-the-uk/. Accessed March 20, 2020.
16. "• UK households: ownership of microwaves 1994-2018 | Statista." Statista, https://www.statista.com/statistics/289155/household-microwave-penetration-in-the-uk/. Accessed March 20, 2020.

17. "•  Washing machine ownership 1970-2018 | Statista." Statista, https://www.statista.com/statistics/289017/washing-machine-ownership-in-the-uk/. Accessed March 20, 2020.
18. "Why Refrigerators Were So Slow to Catch On in China." The Atlantic, May 04, 2016, https://www.theatlantic.com/technology/archive/2016/05/why-refrigerators-were-so-slow-to-catch-on-in-china/481029/. Accessed March 20, 2020.
19. "Why You Need More Than One Vacuum at Home." Consumer Reports, May 05, 2017, https://www.consumerreports.org/vacuum-cleaners/why-you-need-more-than-one-vacuum-at-home/. Accessed March 20, 2020.
20. "Census." Census, https://www.census.gov/content/dam/Census/library/publications/2017/acs/acs-37.pdf. Accessed March 20, 2020.
21. "•  U.S. households with PC/computer at home 2016 | Statista." Statista, https://www.statista.com/statistics/214641/household-adoption-rate-of-computer-in-the-us-since-1997/. Accessed March 20, 2020.
22. "•  U.S. households: digital camera ownership Q4 2010-Q4 2011 | Statista." Statista, https://www.statista.com/statistics/223710/household-digital-camera-ownership-in-the-united-states/. Accessed March 20, 2020.
23. "•  Tablet ownership among U.S. adults 2010-2019 | Statista." Statista, https://www.statista.com/statistics/756045/tablet-owners-among-us-adults/. Accessed March 20, 2020.
24. "23% of U.S. broadband households own wireless earbuds, and 16% own wireless headphones." Parksassociates, https://www.parksassociates.com/blog/article/pr-04112017. Accessed March 20, 2020.
25. "•  Ownership of microwave ovens in Ireland 2003-2017  | Statista." Statista, https://www.statista.com/statistics/656906/utility-ownership-home-development-microwave-ovens-roi/. Accessed March 20, 2020.
26. "•  How many people have access to a computer 2018 | Statista." Statista, https://www.statista.com/statistics/748551/worldwide-households-with-computer/. Accessed March 20, 2020.
27. "Blu-Ray Struggles in the Streaming Age | Fortune." Fortune, https://fortune.com/2016/01/08/blu-ray-struggles-in-the-streaming-age/. Accessed March 20, 2020.
28. "How Many People Own Video Game Consoles?." Marketing Charts, https://www.marketingcharts.com/cross-media-and-traditional/videogames-traditional-and-cross-channel-82362. Accessed March 20, 2020.
29. "•  Electronic devices owned in UK households 2014 | Statista." Statista, https://www.statista.com/statistics/386357/electronic-products-and-devices-owned-in-households-in-the-united-kingdom/. Accessed March 20, 2020.
30. Rudolph, Stacey. "To What Extent Does Branding Affect Consumers' Purchasing Decisions? [Infographics]." Business 2 Community, https://www.business2community.com/infographics/extent-branding-affect-consumers-purchasing-decisions-infographics-01325769  . Accessed March 20, 2020.
31. "Television ownership in private domestic households - CLOSER." https://www.closer.ac.uk/data/television-ownership-in-domestic-households/. Accessed March 20, 2020.

# IMMC Layout Evaluation Code

March 20, 2020

## 1 IMMC 2020: Testing of Layouts

#### 1.0.1 Importing Modules and Data

[1]:
```
import pandas as pd
import copy
import re
import random
import math, statistics
import numpy as np
import progressbar
import gc
import matplotlib.pyplot as plt
from pprint import pprint

median_income = 9733/365
loss_aversion_coefficient = 2
max_shelf_capacity = 150
max_shopper_capacity = 300
units_per_object = 1
max_pdt_per_shelf = 3
pdts_per_shopper = 3
crowd_avoidance = 0.5
mean_shopper_size=50 #mean size of shopper

pdt_csv_data = pd.read_csv("StoreData_IMMC_CSV.csv")
#print(pdt_csv_data.head())
```

[2]:
```
def sigmoid(x):
    return 1 / (1 + math.exp(-x))
```

## 1.1 Determining the Popularity of Product

### 1.1.1 Impact of Discount on Popularity

[3]:
```
# Traditional Econs Approach
def q1_over_q0(p0, p1, percentage_usage):
    x = (p0 - p1)*(percentage_usage/median_income)
    return (math.exp(x))

# Behavioural Econs Approach
def prospect_utility(x):
    if x > 0:
        return(math.log(x+1))
    else:
        return(-loss_aversion_coefficient * math.log(-x + 1))

def increase_utility(p0, p1):
    return(prospect_utility(p0 - p1))

# Helper Variables
max_increase_utility = increase_utility(3329.99, 2199.99)
min_increase_utility = 0
max_q1_over_q0 = q1_over_q0(3329.99, 2199.99, 1)
min_q1_over_qo = 1

# Combined Effect [Between 0 and 1]
def popularity_due_to_discount(p0, p1, percentage_usage):
    traditional_econs_adjusted = (q1_over_q0(p0, p1,␣
↪percentage_usage)-min_q1_over_qo)/max_q1_over_q0
    behavioural_econs_adjusted = (increase_utility(p0,␣
↪p1)-min_increase_utility)/max_increase_utility

    total_adjusted = statistics.mean([traditional_econs_adjusted,␣
↪behavioural_econs_adjusted])
    return(total_adjusted)
```

### 1.1.2 Effect of Loss Adversion on Popularity

[4]:
```
# [Between 0 and 1]
def popularity_due_to_loss_aversion(qty):
    return(math.exp(-qty/loss_aversion_coefficient))
```

### 1.1.3 Effects of Saliency Bias on Popularity (TO DO)

```
[5]: def popularity_due_to_saliency_bias(size, qty):
         return sigmoid(size*qty)
```

### 1.1.4 Effects of Ratings on Popularity

```
[6]: # [Between 0 and 1]
     def popularity_due_to_rating(pdt_rating, brand_rating):
         raw_brand = (0.8*pdt_rating + 0.2*brand_rating)
         return(raw_brand/5)
```

## 1.2 Creating Product Class and List

```
[7]: class product:
         # Popularity Coefficients
         pop_loss_adversion_coefficient = 0.44
         pop_saliency_coefficient = 0.6
         pop_rating_coefficient = 0.5


         # Raw Data
         def __init__(self, index, name, department, product_category, product_type,␣
     ↪brand, initial_price, discounted_price, qty, customer_rating,␣
     ↪brand_rating=5, percentage_usage=0.5, size=20):
             self.name = name
             self.index = index
             self.department = department
             self.product_category = product_category
             self.product_type = product_type
             self.brand = brand
             self.initial_price = initial_price
             self.discounted_price = discounted_price
             self.qty = qty
             self.customer_rating = customer_rating
             self.brand_rating = brand_rating
             self.percentage_usage = percentage_usage
             self.size = size

             self.popularity = 0

         # Processed Data

         def set_popularity(self):
             discount_factor = popularity_due_to_discount(self.initial_price, self.
     ↪discounted_price, self.percentage_usage)
```

```
        loss_adversion_factor = popularity_due_to_loss_aversion(self.qty)
        saliency_factor = popularity_due_to_saliency_bias(self.size, self.qty)
        rating_factor = popularity_due_to_rating(self.customer_rating, self.
↪brand_rating)

        initial_popularity = self.
↪pop_loss_adversion_coefficient*loss_adversion_factor + self.
↪pop_saliency_coefficient*saliency_factor + self.
↪pop_rating_coefficient*rating_factor
        self.popularity = sigmoid(initial_popularity + discount_factor)
```

[8]:
```
# Populating the Product List
pdt_list = []
pdt_counter = 0
for index, row in pdt_csv_data.iterrows():

    total_qty = row["qty"]
    unit_size = row["index_size"]
    while True:

        cur_qty = min(min(int(max_shelf_capacity/unit_size), total_qty),␣
↪units_per_object)
        # print("index", index, "department", row["department"], "qty",␣
↪cur_qty, "net_size", cur_qty*unit_size)
        total_qty -= cur_qty

        cur_pdt = product(index, row["name"], row["department"],␣
↪row["product_category"], row["product_type"], row["brand"],␣
↪row["initial_price"], row["discounted_price"], cur_qty,␣
↪row["customer_rating"], size=unit_size, brand_rating=row["brand_rating"],␣
↪percentage_usage=row["percentage_usage"])
        cur_pdt.set_popularity()

        pdt_counter += 1
        pdt_list.append(cur_pdt)

        if total_qty <=0:
            break

print("pdt_list len", len(pdt_list))
print("Number of products", pdt_counter)
```

```
pdt_list len 1535
Number of products 1535
```

```
[9]: department_popularities = {
         "Appliances": 0,
         "Audio": 0,
         "Cameras": 0,
         "Cell Phones": 0,
         "Computers&Tablets": 0,
         "TV&Home Theater": 0,
         "Video Gaming": 0
     }
     department_qty = {
         "Appliances": 0,
         "Audio": 0,
         "Cameras": 0,
         "Cell Phones": 0,
         "Computers&Tablets": 0,
         "TV&Home Theater": 0,
         "Video Gaming": 0
     }


     total_objects = 0
     for department, total_pop in department_popularities.items():
         number_of_objects = 0
         for pdt in pdt_list:
             if pdt.department == department:
                 number_of_objects += 1
         total_objects += number_of_objects
         print("Department:", department, "Number of Objects", number_of_objects)

     print("Total Number of Objects", len(pdt_list))

     print("--------")

     total_pop = 0
     for pdt in pdt_list:
         department_popularities[pdt.department]+= pdt.popularity*pdt.qty
         department_qty[pdt.department]+= pdt.qty
         total_pop += pdt.popularity*pdt.qty

     for department, pop in department_popularities.items():
         print("Department:", department, "Average Popularity", pop/
      ↪department_qty[department])
     print("Global Average Popularity:", total_pop/len(pdt_list))
```

```
Department: Appliances Number of Objects 257
Department: Audio Number of Objects 50
Department: Cameras Number of Objects 165
```

```
Department: Cell Phones Number of Objects 55
Department: Computers&Tablets Number of Objects 444
Department: TV&Home Theater Number of Objects 383
Department: Video Gaming Number of Objects 181
Total Number of Objects 1535
--------
Department: Appliances Average Popularity 0.8389350550900967
Department: Audio Average Popularity 0.7909613754768059
Department: Cameras Average Popularity 0.8028821374113138
Department: Cell Phones Average Popularity 0.7798840389861017
Department: Computers&Tablets Average Popularity 0.814878007267299
Department: TV&Home Theater Average Popularity 0.8321277481983057
Department: Video Gaming Average Popularity 0.8275033824709389
Global Average Popularity: 0.8213761744384271
```

[10]:
```
pdt_list[4].index
```

[10]: 0

### 1.3 Creating Shelf Class and Layout Object

[11]:
```
class shelf:
    # Class Variables

    def __init__(self, department):
        self.pdts = []
        self.pdt_set = set()
        self.department = department
        self.cur_capacity = 0

    def add_pdt(self, pdt):
        # Check Department
        # print("Adding Product")
        # print("  product_department", pdt.department)
        # print("  shelf_department", self.department)
        if pdt.department != self.department:
            return -1

        # If shelf can accomodate the product
        if self.cur_capacity + pdt.size*pdt.qty <= max_shelf_capacity:
            self.cur_capacity += pdt.size*pdt.qty
            self.pdts.append(pdt)
            self.pdt_set.add(pdt.index)
            return 0

        # If shelf is full
        return -1
```

```
[12]: class layout:

        def __init__(self, grid, counter, entrance, exit, shelf_list=[],␣
      ↪pdt_list=[]):
            self.counter = counter
            self.value_of_goods_bought = 0
            self.shopper_size = 0
            self.entrance = entrance
            self.exit = exit

            # Grid is a 2d matrix where shelves are 1 indexed
            self.grid = copy.deepcopy(grid)
            self.shelf_list = copy.deepcopy(shelf_list)

            # A* Grid is a grid where shelves are labelled as 1
            self.a_star_grid = copy.deepcopy(grid)
            for i in range(len(self.a_star_grid)):
                for j in range(len(self.a_star_grid)):
                    if self.a_star_grid[i][j] > 1:
                        self.a_star_grid[i][j] = 1

            # Shopper Density Grid is a grid to record the density of shoppers;␣
      ↪shelves have a density of 99
            self.shopper_density_grid = copy.deepcopy(self.a_star_grid)
            for i in range(len(self.shopper_density_grid)):
                for j in range(len(self.shopper_density_grid)):
                    if self.shopper_density_grid[i][j] == 1:
                        self.shopper_density_grid[i][j] = -1

            # Price Density Grid is a grid to record to value of products the␣
      ↪customers are carrying at particular locations
            self.price_density_grid = copy.deepcopy(grid)
            for i in range(len(self.price_density_grid)):
                for j in range(len(self.price_density_grid)):
                    if self.price_density_grid[i][j] > 0:
                        self.price_density_grid[i][j] = 0


            # pdt_list contains the products that exists somewhere within the layout
            self.pdt_list = copy.deepcopy(pdt_list)



        # A_Star Performs a simulation of a person walking within the layout from␣
      ↪init to goal
        # A_Star Returns a list of nodes visited on the path
```

```
    # Coordinates are written as [y,x] with [0,0] being the upper left hand␣
↪corner
    def a_star(self, init, goal):
        grid = copy.deepcopy(self.a_star_grid)
        cost = 1

        # the cost map which pushes the path closer to the goal
#         heuristic = [[0 for row in range(len(grid[0]))] for col in␣
↪range(len(grid))]
#         for i in range(len(grid)):
#             for j in range(len(grid[0])):
#                 heuristic[i][j] = abs(i - goal[0]) + abs(j - goal[1])
#                 if grid[i][j] == 1:
#                     heuristic[i][j] = 99  # added extra penalty in the␣
↪heuristic map

        heuristic = [[0 for row in range(len(grid[0]))] for col in␣
↪range(len(grid))]
        for i in range(len(grid)):
            for j in range(len(grid[0])):
                heuristic[i][j] = abs(i - goal[0]) + abs(j - goal[1])
                if grid[i][j] == 1:
                    heuristic[i][j] = 9999999  # added extra penalty in the␣
↪heuristic map
                else:
                    heuristic[i][j] = crowd_avoidance * self.
↪shopper_density_grid[i][j]



        # the actions we can take


        delta = [[-1, 0], [0, -1], [1, 0], [0, 1]]  # go up  # go left  # go␣
↪down  # go right
        # function to search the path
        def search(grid, init, goal, cost, heuristic):



            closed = [
                [0 for col in range(len(grid[0]))] for row in range(len(grid))
            ]  # the reference grid
            closed[init[0]][init[1]] = 1
            action = [
                [0 for col in range(len(grid[0]))] for row in range(len(grid))
            ]  # the action grid

            x = init[0]
            y = init[1]
            g = 0
            f = g + heuristic[init[0]][init[0]]
            cell = [[f, g, x, y]]

            found = False  # flag that is set when search is complete
            resign = False  # flag set if we can't find expand

            while not found and not resign:
                if len(cell) == 0:
                    return "FAIL"
                else:
                    cell.sort()  # to choose the least costliest action so as␣
↪to move closer to the goal
                    cell.reverse()
                    next = cell.pop()
                    x = next[2]
                    y = next[3]
                    g = next[1]

                    if x == goal[0] and y == goal[1]:
                        found = True
                    else:
                        for i in range(len(delta)):  # to try out different␣
↪valid actions
                            x2 = x + delta[i][0]
                            y2 = y + delta[i][1]
                            if x2 >= 0 and x2 < len(grid) and y2 >= 0 and y2 <␣
↪len(grid[0]):
                                if closed[x2][y2] == 0 and grid[x2][y2] == 0:
                                    g2 = g + cost
                                    f2 = g2 + heuristic[x2][y2]
                                    cell.append([f2, g2, x2, y2])
                                    closed[x2][y2] = 1
                                    action[x2][y2] = i
            invpath = []
            x = goal[0]
            y = goal[1]
            invpath.append([x, y])  # we get the reverse path from here
            while x != init[0] or y != init[1]:
                x2 = x - delta[action[x][y]][0]
                y2 = y - delta[action[x][y]][1]
                x = x2
                y = y2
                invpath.append([x, y])

            path = []
            for i in range(len(invpath)):
                path.append(invpath[len(invpath) - 1 - i])

#             print("ACTION MAP")
#             for i in range(len(action)):
#                 print(action[i])

            return path
        return search(grid, init, goal, cost, heuristic)

    # Simulates Choice of Object to Buy and Deletes that Object from Object List
    def choose_pdt(self):
        if len(self.pdt_list) == 0:
            print("RAN OUT OF ITEMS IN SHOP")
            return -1
        self.pdt_list.sort(key=lambda x: x.popularity, reverse=True)
        index = min(random.randint(0, 5), len(self.pdt_list) -1)
        chosen_pdt = copy.deepcopy(self.pdt_list[index])
        return chosen_pdt

    def delete_pdt(self, product_index):
        for i in range(len(self.pdt_list)):
            if self.pdt_list[i].index == product_index:

                if self.pdt_list[i].qty == 1:
                    self.pdt_list.pop(i)
                else:
                    self.pdt_list[i].qty -= 1

                break

        return

    # Find Product in Sheleves
    def find_shelf(self, chosen_pdt_index):
        for shelf_index, shelf in enumerate(self.shelf_list):
                # print("Checking shelf", shelf_index)
                if chosen_pdt_index in shelf.pdt_set:
                    chosen_shelf_index = shelf_index
        return chosen_shelf_index

    def delete_pdt_from_shelf(self, chosen_pdt_index, shelf_index):
        # Remove product from shelf
        shelf = self.shelf_list[shelf_index]
        for i in range(len(shelf.pdts)):
            if shelf.pdts[i].index == chosen_pdt_index:
                # print("QTY of pdt", shelf.pdts[i].qty)
                    if shelf.pdts[i].qty <= 1:
                        del shelf.pdts[i]

                    else:
                        shelf.pdts[i].qty-= 1
                        shelf.pdts[i].set_popularity()
                    break

        last_product_of_type = True
        for j in range(len(shelf.pdts)):
            if shelf.pdts[j].index == chosen_pdt_index:
                last_product_of_type = False
                # print("##@@##@###@###@#####@#@#@###@##Meow")
                break
        if last_product_of_type:
            shelf.pdt_set.discard(chosen_pdt_index)
        return



    def walk(self, init, goal, cur_size, cur_price):
        def fn(y,x):

            if x >= 0 and x < len(self.grid) and y >= 0 and y < len(self.
↪grid[0]):
                if goal == self.exit:
                    [y,x] = [goal[0], goal[1]]
                path = self.a_star(init, [y,x])
                if path == "FAIL":
                    z=0
                    return -1
                else:
                    for cell in path:
                        self.shopper_density_grid[cell[0]][cell[1]] += cur_size
                        self.price_density_grid[cell[0]][cell[1]] += cur_price
                    cur_pos = [y,x]
                    return cur_pos
            else:
                return -1

        # Conduct A*
        delta = [(-1,0), (1,0), (0,-1), (0,1)]
        random.shuffle(delta)
        for d in delta:
            y = goal[0]+d[0]
            x = goal[1]+d[1]
            new_pos = fn(y,x)
            if new_pos != -1:
                return new_pos

        # If the shelf is not reachable, try to each an adjacent shelf
        for d in delta:
            for d2 in delta:
                y = goal[0]+d[0]+d2[0]
                x = goal[1]+d[1]+d2[1]
                new_pos = fn(y,x)
                if new_pos != -1:
                    return new_pos

    # Simulates a person entering shop
    def new_shopper(self):
        max_number_of_products = pdts_per_shopper
        max_size = max_shopper_capacity

        cur_number_of_pdt = 0
        cur_size = random.randint(mean_shopper_size-10, mean_shopper_size+10)
        cur_price = 0


        # Random Entrance Square
        cur_pos = random.choice(self.entrance)

        if len(self.pdt_list) == 0:
            return -1
        while cur_size < max_size and cur_number_of_pdt < 3 and len(self.
↪pdt_list) > 0:

            # Choose what Product to Buy
            chosen_pdt = self.choose_pdt()
            if cur_size + chosen_pdt.size > max_size:
                break


            # Find Shelf Index
            chosen_shelf_index = self.find_shelf(chosen_pdt.index)

            # Find Location of Shelf
            for y in range(len(self.grid)):
                for x in range(len(self.grid[0])):
                    if self.grid[y][x] == chosen_shelf_index:
                        shelf_location = (y,x)
            # Walk from cur_pos to another shelf while tracking the movement of␣
↪the shopper
            cur_pos = self.walk(cur_pos, shelf_location, cur_size, cur_price)
            self.delete_pdt(chosen_pdt.index)
            self.delete_pdt_from_shelf(chosen_pdt.index, chosen_shelf_index)
            self.value_of_goods_bought += chosen_pdt.discounted_price
            cur_number_of_pdt +=1
            cur_price += chosen_pdt.discounted_price
            cur_size += chosen_pdt.size

        # Walk to Counter
        cur_pos = self.walk(cur_pos, random.choice(self.counter), cur_size,␣
↪cur_price)
        cur_pos = self.walk(cur_pos, random.choice(self.exit), cur_size, 0)

        self.shopper_size += cur_size

        return 0

    # Get adjusted Price Grid
    def get_price_grid(self):
        total_price = 0
        for i in range(len(self.price_density_grid)):
            for j in range(len(self.price_density_grid)):
                total_price += self.price_density_grid[i][j]

        adjusted_grid = copy.deepcopy(self.price_density_grid)
        for i in range(len(self.price_density_grid)):
            for j in range(len(self.price_density_grid)):
                adjusted_grid[i][j] *= (self.value_of_goods_bought/total_price)
                adjusted_grid[i][j] = int(adjusted_grid[i][j])

        return adjusted_grid

    def get_shopper_grid(self):
        total_size = 0
        for i in range(len(self.shopper_density_grid)):
            for j in range(len(self.shopper_density_grid)):
                total_size += self.shopper_density_grid[i][j]

        adjusted_grid = copy.deepcopy(self.shopper_density_grid)
        for i in range(len(self.shopper_density_grid)):
            for j in range(len(self.shopper_density_grid)):
                adjusted_grid[i][j] *= (self.shopper_size/total_size)
                adjusted_grid[i][j] = int(adjusted_grid[i][j])
        return adjusted_grid

    # Collision Damage / Self-Drops
    def loss(self, shopper_density, price_density):
        p_collision = (min(shopper_density, 300)) **2 / (300*300)
        return p_collision*price_density

    def total_loss(self):
        shopper_density_grid = self.get_shopper_grid()
        price_density_grid = self.get_price_grid()

        net_loss = 0
        for y in range(len(shopper_density_grid)):
            for x in range(len(shopper_density_grid)):
                net_loss += self.loss(shopper_density_grid[y][x],␣
↪price_density_grid[y][x])

        return net_loss
```

## 2 Simulate Layout in Figure 1

### 2.1 Simulation Cells

[13]:
```
template='''
ffffffffffffffffffffffffffffffffffffffffffffffff
000000000000000000000000000000000000000000000000
0gg000000000000000000000000000000000000fffff000f
0gg000000000000000000000000000000000000000ff000f
0gg000000000000000000000000000000000000000ff000f
0gg00dd0000eeeeee00000eeeeee00000000000000ff000f
0gg00dd0000eeeeee00000eeeeee00000000000000ff000f
0gg00dd0000000000000000000000000000000000000000f
0gg00dd0000eeeeee000000eeeeeeee0000cccc00000000f
0gg00dd0000eeeeee000000eeeeeeee0000cccc000ff000f

0gg00dd00000000000000000000000000000000000ff000f
0gg00dd0000eeeeee000000eeeeeeee00000000000ff000f
0gg00dd0000eeeeee000000eeeeeeee0000cccc000ff000f
0gg00dd0000000000000000000000000000cccc00000000f
0gg00dd0000eeeeee000000eeeeeeee0000000000000000f
0gg00000000eeeeee000000eeeeeeee00000000000ff000f
0gg00000000000000000000000000000000cccc000ff000f
0gg00000000eeeeee000000eeeeeeee0000cccc000ff000f
0gg00000000eeeeee000000eeeeeeee00000000000ff000f
0gg00000000000000000000000000000000000000000000f
```

```
0gg00bb0000eeeeee00000eeeeee0000000cccc00000000f
0gg00bb0000eeeeee00000eeeeee0000000cccc000ff000f
0gg00bb00000000000000000000000000000000000ff000f
0gg00bb00000000000000000000000000000000000ff000f
0gg00bb0000000000000000000000000000cccc000ff000f
0gg00bb0000000000000000000000000000cccc00000000f
0gg00bb00aa00aa00aa00aa00aa00aa0000000000000000f
0gg00bb00aa00aa00aa00aa00aa00aa00000000000ff000f
00000bb00aa00aa00aa00aa00aa00aa0000cccc000ff000f
00000bb00aa00aa00aa00aa00aa00aa0000cccc000ff000f

000000000aa00aa00aa00aa00aa00aa00000000000ff000f
000000000aa00aa00aa00aa00aa00aa0000000000000000f
00000000000000000000000000000000000cccc00000000f
00000000000000000000000000000000000cccc00000000f
00000000000000000000000000000000000000000000000f
00000gg00aa00aa00aa00aa00aa00aa0000000000000000f
00000gg00aa00aa00aa00aa00aa00aa0000ggcc00000000f
00000gg00aa00aa00aa00aa00aa00aa0000ggcc00000000f
00000gg00aa00aa00aa00aa00aa00aa0000000000000000f
00000000000000000000000000000000000000000000000f

00000000000000000000000000000000000000000000000f
00000000000000000000000000000000000000000000000f
00000000000000000000000000000000000000000000000f
00000000000000000000000000000000000000000000000f
000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000
'''
departments = ["a","f","c","d","e","b","g"]

def generate_layout(a,f):
    if a == f:
        return ""
    index = 2
    department_used = [0,0,0,0,0,0,0,0] # Whether the number has been used
    department_used[a] =department_used[f] = 1
    print("A:", a, "F:",f)
    layout = (template)
    layout = layout.replace(str(a), "a")
    layout = layout.replace(str(f), "f")

    for i in range(1,8):
        if not department_used[i]:
            print("replacing", i, "with", departments[index])
            layout = layout.replace(str(i), departments[index])
            department_used[i] = 1
            index += 1
    return layout

# for a in range(1,8):
#     for b in range(1,8):
#         print(generate_layout(a,b))
```

[14]:
```
def split_row(word):
    return [char for char in word]

def evaluate_layout(test_layout, counters = [(40, 6),(40, 7),(40, 8),(40,␣
↪9),(40, 10), (40, 11), (40,12), (40,13)]):
    # Parse Layout
    test_layout = test_layout.replace('\n\n', '\n')
    test_layout = test_layout.strip("\n")

    test_layout = test_layout.split("\n")
    for i in range(len(test_layout)):
        test_layout[i] = split_row(test_layout[i])

    department_dictionary = {
        "a": "Appliances",
        "b": "Audio",
        "c": "Cameras",
        "d": "Cell Phones",
        "e": "Computers&Tablets",
        "f": "TV&Home Theater",
        "g": "Video Gaming",
    }
    shelf_list = [shelf("")]
    shelf_counter = 1
    for (letter, department_name) in department_dictionary.items():
        for y in range(len(test_layout)):
            for x in range(len(test_layout)):
                if str(test_layout[y][x]) == letter:
                    # print("found shelf", shelf_counter, "department:",␣
↪department_name)
                    test_layout[y][x] = str(shelf_counter)
                    shelf_counter += 1
                    shelf_list.append(shelf(department_name))

    for y in range(len(test_layout)):
        for x in range(len(test_layout)):
            if not str(test_layout[y][x]).isdigit():
                test_layout[y][x] = str(shelf_counter)
                shelf_counter +=1
            test_layout[y][x] = int(test_layout[y][x])



    max_shelf_index = shelf_counter


    # Put The Objects in Shelves
    pdt_index = 0
    pdt_list.sort(key=lambda x: x.department)

    for shelf_index in range(1, max_shelf_index):
        # print("\nShelf Index: ", shelf_index)
        cur_shelf = shelf_list[shelf_index]
        # print("Shelf Department -- ", cur_shelf.department)

        for i in range(max_pdt_per_shelf):
            next_pdt = copy.deepcopy(pdt_list[pdt_index])
            if next_pdt.department == "Appliances" and cur_shelf.department !=␣
↪"Appliances":
                print("Insufficient Appliance Space")
                return 9999999
            # print("Products Department", next_pdt.department)
            if cur_shelf.add_pdt(next_pdt) != -1:
                # print("Added pdt id", next_pdt.index)
                pdt_index += 1
                if pdt_index == len(pdt_list):
                    break
            else:
                break
        # print("Products in shelf" , shelf_index, ":", cur_shelf.pdt_set)

        if pdt_index == len(pdt_list):
            if next_pdt.department != "Video Gaming":
                print("ERROR INSUFFICIENT SHELVES")
                return 9999999
            print("--- Finished All Products with", shelf_index, "out of",␣
↪max_shelf_index, "shelves --- ")
            break

    # Set up Object
    ## Test Layout with Model
    # counters = [(40, 6),(40, 7),(40, 8),(40, 9),(40, 10), (40, 11), (40,12),␣
↪(40,13)]
    entrances = [[47,30],[47,29],[47,28]]
    exits = [[48,35],[48,36],[48,37]]
    test_layout_object = layout(test_layout , counters, entrances, exits,␣
↪shelf_list, pdt_list)
    # print("Number of products in layout:", len(test_layout_object.pdt_list))


    # Do Simulation
    gc.collect()
    for i in progressbar.progressbar(range(600)):
        try:
            shopper_density_grid = test_layout_object.new_shopper()
            if shopper_density_grid == -1:
                print("Exhausted Shop with approximately", i, "shoppers")
                break
        except:
            pass

    # Get Results
    price_density_grid = test_layout_object.get_price_grid()
    shopper_density_grid = test_layout_object.get_shopper_grid()

    print("Shopper Density SD:", np.std(shopper_density_grid, axis=(0,1)))
    print("Price Density SD:", np.std(price_density_grid, axis=(0,1)))

    net_loss = test_layout_object.total_loss()
    print("Damage Level:", net_loss)

    shelf_grid = copy.deepcopy(test_layout_object.a_star_grid)
    for counter in counters:
        shelf_grid[counter[0]][counter[1]] = 0.7
    for entrance in entrances:
        shelf_grid[entrance[0]][entrance[1]] = 0.5
    for exit in exits:
        shelf_grid[exit[0]-1][exit[1]] = 0.5
    return (net_loss, price_density_grid, shopper_density_grid, shelf_grid)
```

```
[ ]: # Rotate Departments A and F Around the Various Positions
min_loss = 999999
min_a_f = (9,9)

results = []
for a in range(1,8):
    for f in range(1,8):
        print("##################################################","TESTING A␣
↪F", a, f, "##################################################")
        if a == f:
            continue
        cur_loss = evaluate_layout(generate_layout(a,f))
        if cur_loss == 9999999:
            continue

        cur_loss = cur_loss[0]
        min_loss = min(cur_loss, min_loss)
        if min_loss == cur_loss:
            min_a_f = (a,f)
            print("New Best :)")

        results.append((cur_loss,(a,f)))

print("Best a,f", min_a_f)
results.sort()
print("Results", results)
```

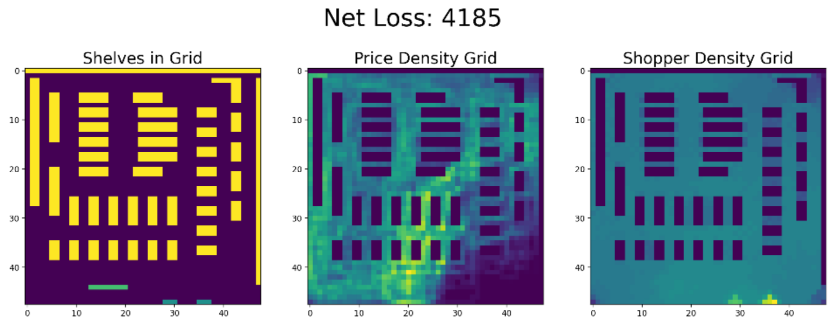
## 3 Simulate Our Layout

[15]:
```
our_layout='''
ffffffffffffffffffffffffffffffffffffffffffffffff
000000000000000000000000000000000000000000000000
0gg000000000000000000000000000000000000fffff000f
0gg000000000000000000000000000000000000000ff000f
0gg000000000000000000000000000000000000000ff000f
0gg00dd0000eeeeee00000eeeeee00000000000000ff000f
0gg00dd0000eeeeee00000eeeeee00000000000000ff000f
0gg00dd0000000000000000000000000000000000000000f
0gg00dd0000eeeeee000000eeeeeeee0000cccc00000000f
0gg00dd0000eeeeee000000eeeeeeee0000cccc000ff000f

0gg00dd00000000000000000000000000000000000ff000f
0gg00dd0000eeeeee000000eeeeeeee00000000000ff000f
0gg00dd0000eeeeee000000eeeeeeee0000cccc000ff000f
0gg00dd0000000000000000000000000000cccc00000000f
0gg00dd0000eeeeee000000eeeeeeee0000000000000000f
0gg00000000eeeeee000000eeeeeeee00000000000ff000f
0gg00000000000000000000000000000000cccc000ff000f
0gg00000000eeeeee000000eeeeeeee0000cccc000ff000f
0gg00000000eeeeee000000eeeeeeee00000000000ff000f
0gg00000000000000000000000000000000000000000000f

0gg00bb0000eeeeee00000eeeeee0000000cccc00000000f
0gg00bb0000eeeeee00000eeeeee0000000cccc000ff000f
0gg00bb00000000000000000000000000000000000ff000f
0gg00bb00000000000000000000000000000000000ff000f
0gg00bb0000000000000000000000000000cccc000ff000f
0gg00bb0000000000000000000000000000cccc00000000f
```

```
0gg00bb00aa00aa00aa00aa00aa00aa0000000000000000f
0gg00bb00aa00aa00aa00aa00aa00aa00000000000ff000f
00000bb00aa00aa00aa00aa00aa00aa0000cccc000ff000f
00000bb00aa00aa00aa00aa00aa00aa0000cccc000ff000f

000000000aa00aa00aa00aa00aa00aa00000000000ff000f
000000000aa00aa00aa00aa00aa00aa0000000000000000f
00000000000000000000000000000000000cccc00000000f
00000000000000000000000000000000000cccc00000000f
00000000000000000000000000000000000000000000000f
00000gg00aa00aa00aa00aa00aa00aa0000000000000000f
00000gg00aa00aa00aa00aa00aa00aa0000ggcc00000000f
00000gg00aa00aa00aa00aa00aa00aa0000ggcc00000000f
00000gg00aa00aa00aa00aa00aa00aa0000000000000000f
00000000000000000000000000000000000000000000000f

00000000000000000000000000000000000000000000000f
00000000000000000000000000000000000000000000000f
00000000000000000000000000000000000000000000000f
00000000000000000000000000000000000000000000000f
000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000
'''
```

```
[16]: a=7
f=1
# counters = [(44, 6),(44, 7),(44, 8),(44, 9),(44, 10), (44, 11), (44,12),␣
 ↪(44,13)] # Basic Counters
counters = [[44, 13],[44, 14],[44, 15],[44, 16],[44, 17], [44, 18], [44,19],␣
 ↪[44,20]] # Optimised Counters
# counters = [(44, 6),(44, 7),(44, 8),(44, 9),(44, 10), (44, 11), (44,12),␣
 ↪(44,13),[42, 25],[42, 26],[42, 27],[42, 28], [42, 29], [42,30], [42,31]] #␣
 ↪Double Position Counters
net_loss, price_density_grid, shopper_density_grid, shelf_grid = ␣
 ↪evaluate_layout(our_layout,counters=counters)


fig=plt.figure(figsize=(18, 6.5), dpi= 200, facecolor='w', edgecolor='k')
# plt.suptitle("A: "+ str(a) + " | F: "+ str(f), fontsize=30)
plt.suptitle("Net Loss: " + str(int(net_loss)), fontsize=30)
plt.subplot(1, 3, 1)
plt.title("Shelves in Grid",fontsize=20)
plt.imshow(shelf_grid)
plt.subplot(1, 3, 2)
plt.title("Price Density Grid",fontsize=20)
plt.imshow(price_density_grid)
plt.subplot(1, 3, 3)
plt.title("Shopper Density Grid",fontsize=20)
plt.imshow(shopper_density_grid)


plt.figure(num=None, figsize=(100, 100), dpi=900, facecolor='w', edgecolor='k')
plt.show()
```

```
N/A% (0 of 600) |                       | Elapsed Time: 0:00:00 ETA:  --:--:--

--- Finished All Products with 577 out of 581 shelves ---

 85% (513 of 600) |##################    | Elapsed Time: 0:00:37 ETA:   0:00:05

Exhausted Shop with approximately 515 shoppers
Shopper Density SD: 9.191316869807244
Price Density SD: 311.266008371722
Damage Level: 4185.8410111111125
```



```
<Figure size 90000x90000 with 0 Axes>
```

## 3.1 Sensitivity Test Helper Code

```
[ ]: # Change Single Variables and Graph Output
x = np.arange(160, 490, 30)
y = []

print("Number of Samples,", len(x))

counters = [(44, 6),(44, 7),(44, 8),(44, 9),(44, 10), (44, 11), (44,12),␣
 ↪(44,13)]
```

```
for x_val in x:
    print("### Testing", x_val, "###")
    max_shopper_capacity = x_val
    net_loss, price_density_grid, shopper_density_grid, shelf_grid = ␣
 ↪evaluate_layout(our_layout,counters=counters)
    y.append(net_loss)
y = np.array(y)
```

```
[ ]: # Specific for Counters
x = np.arange(0, 4, 1)
y = []

raw_counters = [[44, 13],[44, 14],[44, 15],[44, 17], [44, 18], [44,19],␣
 ↪[44,20],[42, 13],[42, 14],[42, 15],[42, 17], [42, 18], [42,19], [42,20]]


for x_val in x:
    print("### Testing", x_val, "###")
    counters = copy.deepcopy(raw_counters)

    for i in range(x_val):
        last = copy.deepcopy(counters[len(counters)-1])
        last[1] += 1
        counters.append(last)
    print("Counters",counters)
    net_loss, price_density_grid, shopper_density_grid, shelf_grid = ␣
 ↪evaluate_layout(our_layout,counters=counters)
    y.append(net_loss)
y = np.array(y)
```

```
[ ]: plt.figure(num=None, figsize=(12, 8), dpi=80, facecolor='w', edgecolor='k')
plt.rc('xtick', labelsize=12)     # fontsize of the tick labels
plt.rc('ytick', labelsize=12)
plt.xlabel("Counter Number on Right", fontsize=20)
plt.ylabel("Net Loss", fontsize=20)
plt.plot(x,y)
plt.show()
```

```
[ ]:
```