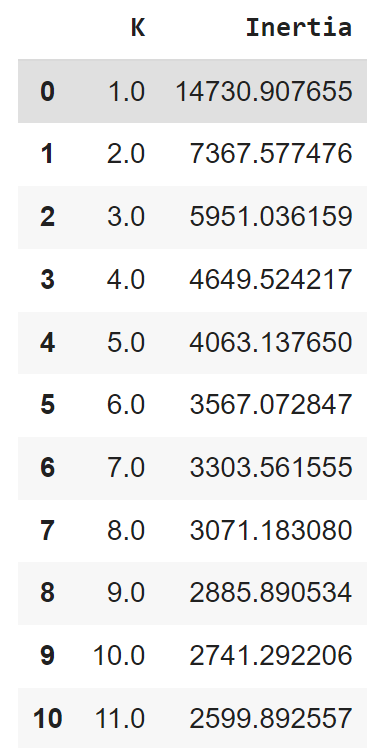
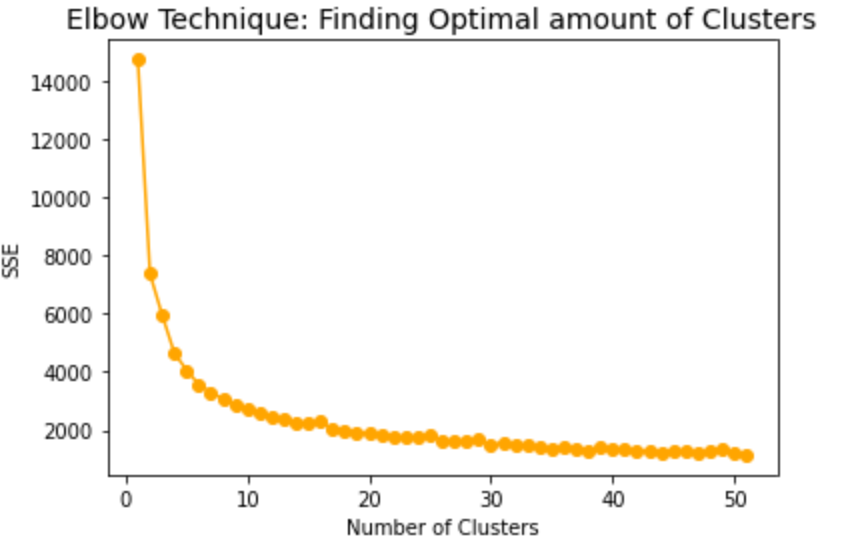
# **Part 1: Predicting Optimal Prices**

# Our high level strategy was to solve the cold start problem with k-means clustering, predict product affinity for each user with matrix factorization, train a logistic regression to estimate personalized demand curves, and use optimal one-shot pricing by maximizing total revenue.

## **Cold Start Problem: K Means Clustering**

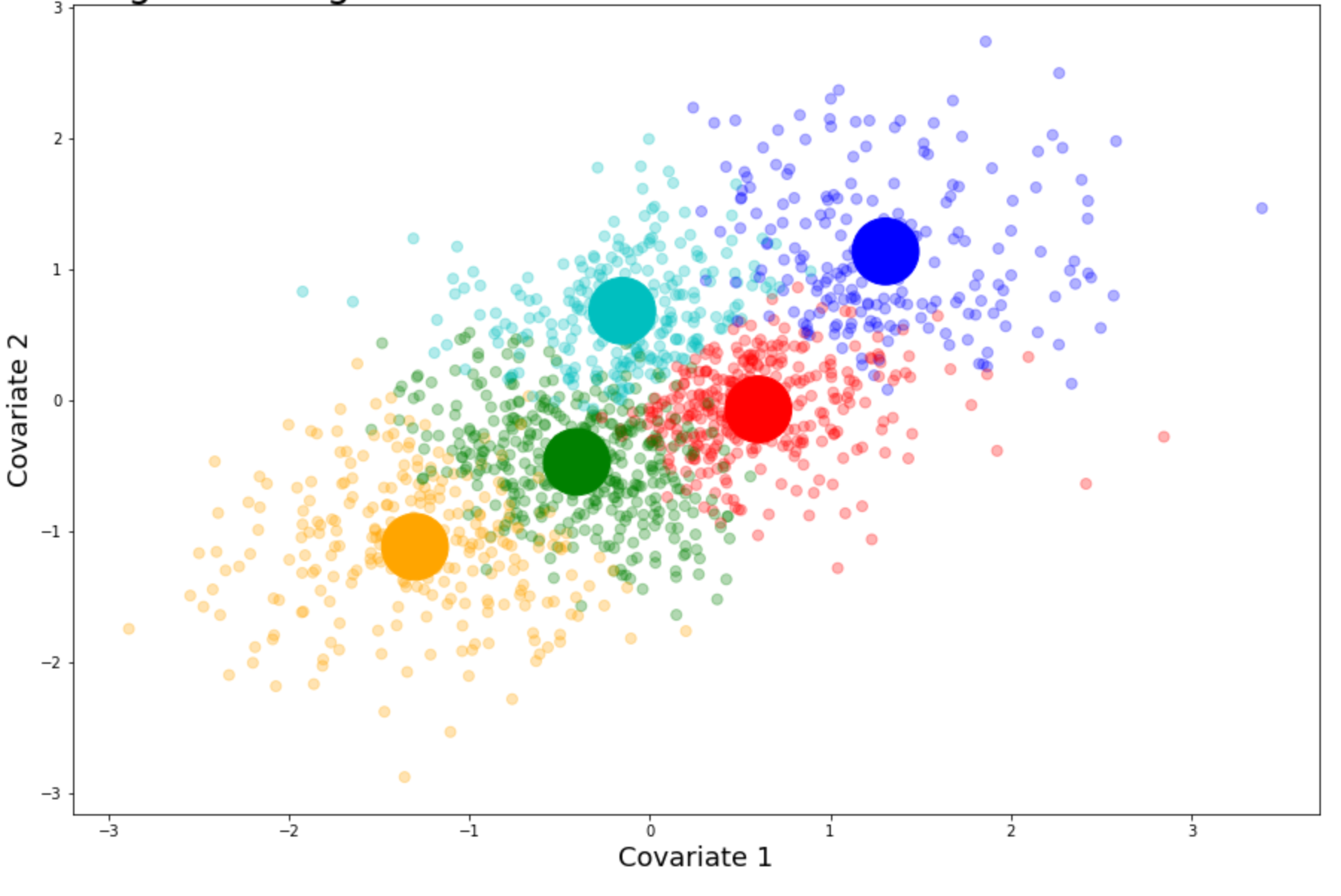
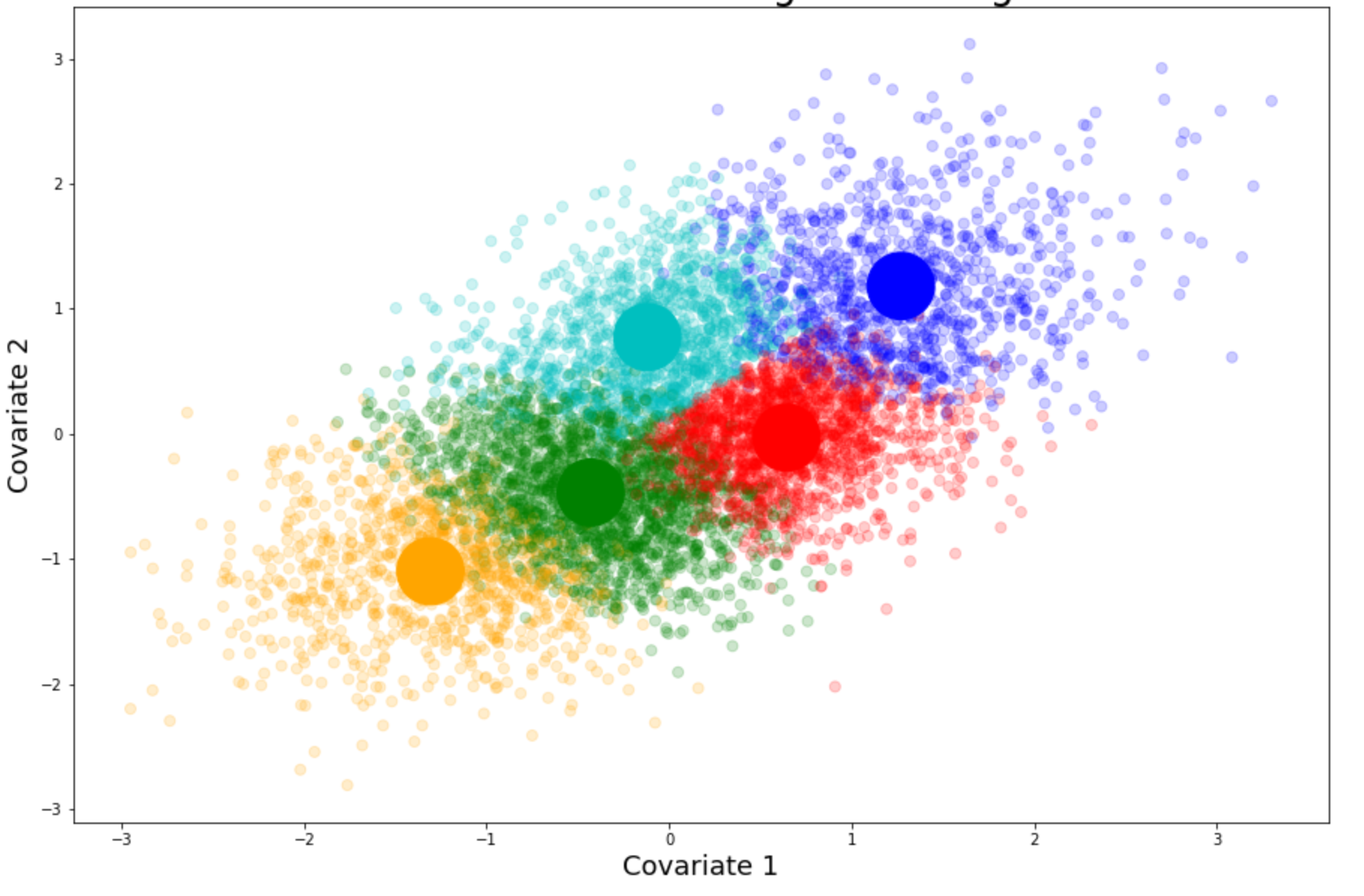
Using the train buyer covariate set, we trained an unsupervised k-means clustering machine learning model. K-means partitions the data into K clusters where each observation is assigned to the nearest mean centroid to minimize within cluster variance (squared Euclidean distance). We used the elbow technique to find the optimal number of clusters (between 1-50) that minimized SSE without overfitting the data. In Figure 1, the elbow of the graph occurs around five, resulting in an inertia of 3567 (Table 1). While SSE does decrease slightly with six or more clusters, we found that implementation complexity of downstream steps increased with additional clusters. After choosing five clusters, we trained the k-means model on the training data and for each cluster averaged the noisy user embeddings within an assignment (Figure 2). Through this process we created five user vectors and when given a set of new users, our k-means model assigns the new user a cluster (Figure 3) and their corresponding embedding (Table 2 in Appendix). This approach to solve the cold start problem assumes that all the users who enter the store belong to five distinct categories. Using Figure 2 and Figure 3, we were able to see our k-means model successfully assigning new users a cluster in a way that made sense, allowing us to move forward.





*Figure 1. Elbow Technique to Minimize SSE*

*Table 1. K Means SSE*



## 

*Figure 2. K Means on Existing Users Figure 3. K Means with New Users*

**Recommendation System: Matrix Factorization**

## We then used a matrix factorization approach to derive two new features for each user ‘item0\_pred’ and ‘item1\_pred’ by taking the dot product between the user and item embeddings which represent their likelihood to buy each item (Table 2 in Appendix). Since this approach taught in class is quick and effective we did not experiment with any other recommendation systems.

**Demand Estimation: Logistic Regression**

Our multi-class logistic regression estimates the likelihood a user will buy Item 0, Item 1 or neither. The features fed into the model are; ‘Price of Item 0’, ‘Price of Item 1’, ‘Item 0 Prediction’ , ‘Item 1 Prediction’, ‘Covariate 1’, Covariate 2’, ‘Covariate3’ and the target column is ‘Item Bought”.

Keeping in mind HW 3 (where the Logistic Regression demand estimation technique most closely followed the true demand curve), we had an inclination that Logistic Regression would perform best (with an efficient run time). However, it was still important to evaluate our method. We decided to compare Logistic Regression against a Random Forest Classifier - a method that could provide more granularity in prediction. Splitting the training data into a training and a testing set (95%-5% split), we evaluated both the accuracy and the mean squared error of both models - it was important to take both measures in order to understand how many points are incorrect, and exactly how incorrect those points are. The Logistic Regression provided a higher accuracy of 88.1% compared to 86.1%, and a lower mean squared error of 19.6% compared to 23.7%, so we chose it over the Random Forest - especially because Random Forest proved to be too slow for the optimal one shot pricing.

| **Classifier** | **Accuracy** | **Mean Squared Error** |
| --- | --- | --- |
| Logistic Regression | 0.881 | 0.196 |
| Random Forest | 0.861 | 0.237 |

Calculating the accuracy of 88.1% and MSE of 19.6% gave us the confidence to move forward - with a cold start method (aka not true training data), close to 90% accuracy was a great result, and a MSE of close to 0.2 showed the model did not do a terrible job at predicting what item a customer would buy at certain prices. We then used predict\_proba to estimate the demand for an item at a certain price.

**Optimal Pricing: Single Shot Revenue Maximization**

When the logistic regression model outputs a demand estimation, we use a brute-force method to create a price set. Evaluation mainly consisted of optimizing revenue - we were not predicting anything new at this step and our classifier had already passed our evaluation, we just had to make sure the price optimization could produce a high revenue within a reasonable time. Ideally, we would test every possible price (i.e. taking prices every 1 cent), but this was too slow to be feasible.

We found every 5 cents was the most granular we could get within a reasonable time frame (taking 1 hour to run). The price set consisted of 5 cent interval prices from the minimum to maximum realized price for both item 0 and item 1. This created 3600 combinations of prices for our code to search through for each user and picked the combination of prices that maximizes revenue. Calculating our expected revenue on the test set with our predictions, we expected a revenue of $4110dollars. Comparing this with taking predictions every 20 cents (taking only 3 minutes to run), we expected a revenue of $3964. Since the 5-cent revenue was higher, we chose those outputs. For analysis, we calculated expected revenue using other team’s prices but our demand estimation model (full table can be seen in appendix). Taking the absolute value of the difference between expected revenue based on our demand model and the team’s expected revenue, the average difference of teams that placed above us was only $93, leading us to believe our demand estimation model was similar to higher achieving teams - but perhaps there is room for improvement in our price optimization model. In teams that placed below us, the average difference is $520, implying that our demand estimation model is significantly different.

**Part 2: Pricing with Competition**

**Alpha Strategy Iteration**

Moving into the competition we wanted to adjust our prices based off an alpha strategy so that we would slightly undercut the opponent every round and and make the sale. Originally, we went with a naive alpha strategy where we would make the alpha higher by 0.1 if we got the last sale and lower by 0.1 if the opponent got the last sale.

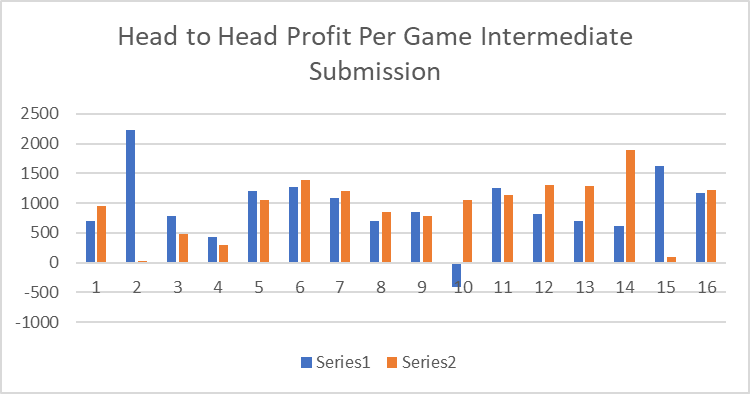
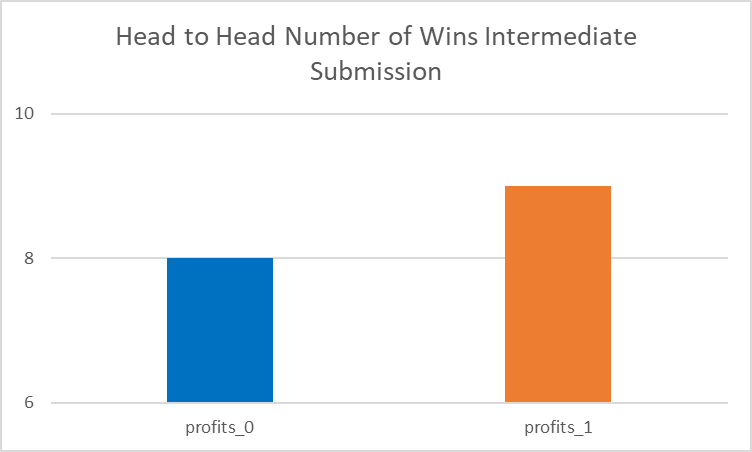
(*Formula 1)*

In the first run of part 2 we did relatively well with this strategy (placing 4th) but this may have been because there were few competitors (9 excluding dummies). For the next run of part 2 we decided to learn our opponents' prices and undercut them by $0.05. This was done via a linear regression model that took in the covariates and predicted the opponent's price starting at epoch 50 and was updated every round. To evaluate this method we ran our naive strategy against this new learning strategy and found that the predictions for item 1 prices had a low accuracy. This dissuaded us from learning prices for item 1 and so we decided to keep the naive strategy for item 1. The performance of our linear regression model on item 0 was sufficient so we kept this strategy for item 0 prices.

*(Formula 3)*

*(Formula 4)*

Then this strategy was run in the intermediate submission we won 8 games and lost 9 (as seen in Figure 5) but our overall average revenue was at a loss (thus we placed last overall) because we lost a substantial amount of money in the head to head against teamsvm as seen in Figure 4. While we were quite surprised at this result, we figured out that we forgot a crucial check in our strategy (needing to check that our prices stay between a reasonable price threshold). When the opponent set low prices in the first 50 epochs, our algorithm was predicting negative prices.

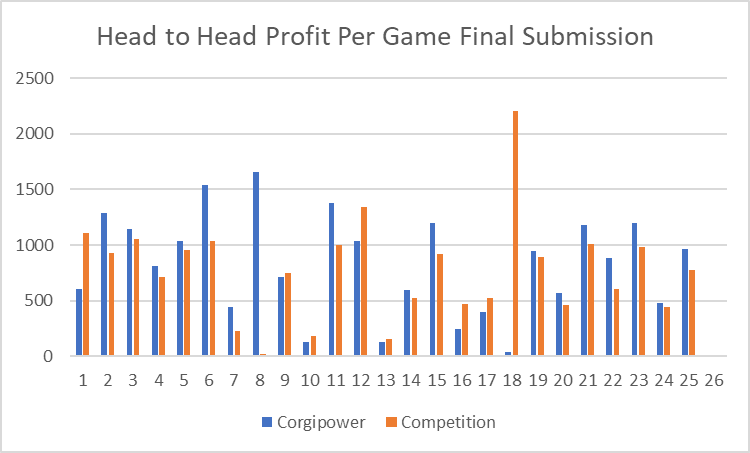
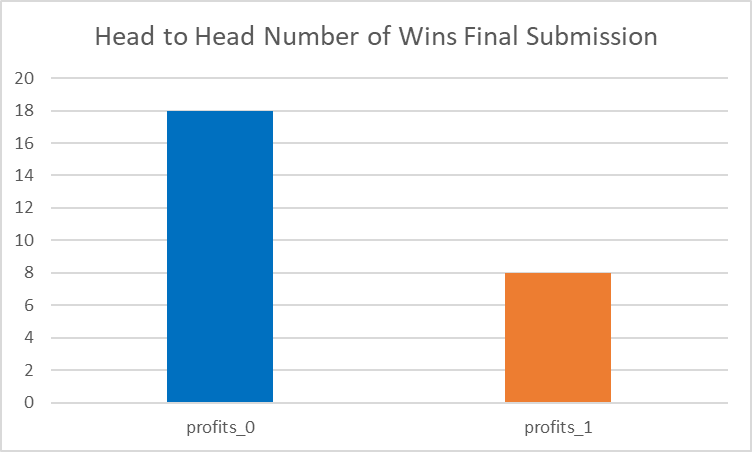


*Figure 4: Profit per game intermediate submission. Figure 5: Results from intermediate.*

To ensure that this would not be the case in the final round we checked that our predicted prices of opponents were between $0.8 and $3.00 and if not set the prices to the naive strategy. This defended opponents from sabotaging our predictions. Before submitting our code to the final submission we ran our agent against all of the dummies and our previous strategies to ensure that we made improvements.

**Final Results**

Our team ended up in the middle of the leaderboard (13th/27) with 18 head to head wins and 8 losses. This is a substantial improvement from the intermediate submission. Our average revenue per game was $838.



*Figure 6: Profit per game final submission. Figure 7: Results from final submission.*

**Conclusion**

Our team was satisfied with our strategies improvement over the course of the project. Although we did not come in the top 3, our team did relatively well in most head to head games. One thing that we could have improved on is thinking of a level 2 strategy (what our opponents will do) rather than just level one strategy. We went with many of the ideas listed in the strategy document that other teams took advantage of (like teamsvm setting negative prices in the beginning to throw off our linear regression). The extreme price setting also caused the alpha to skyrocket opening an opportunity for teamsvm to compete on higher revenue items. If we were to do this project again, we would try to do a similar strategy to take advantage of opponents and naive price strategies. This project was both exciting and informative and will definitely translate to our careers as data scientists. One addition to this project that may have been interesting to explore is having capacity constraints on each item. This would have made the project more realistic. Overall, this project and class in general sparked our interest in how to navigate systems that interact with people and we hope to do more of this in the future.

# 

# 

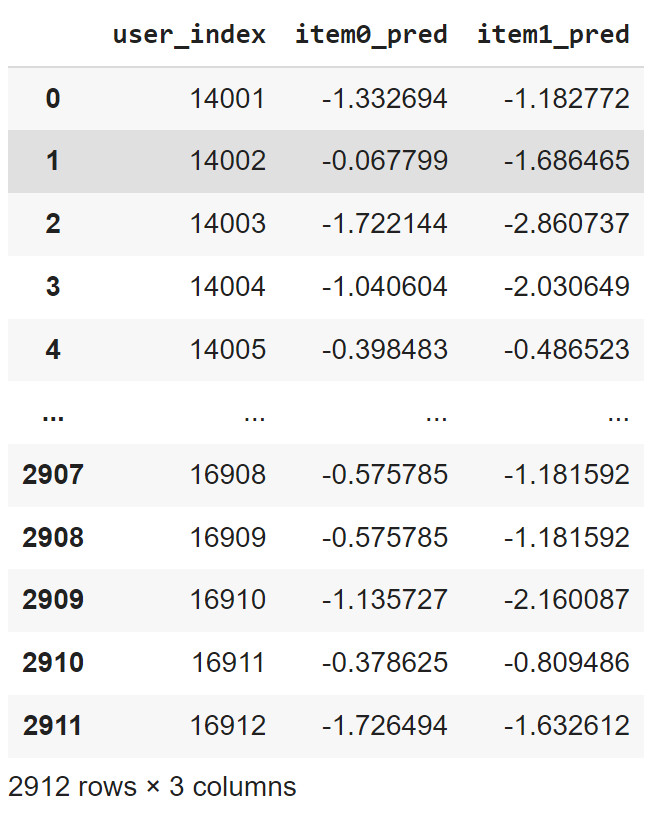
# 

# 

# 

# 

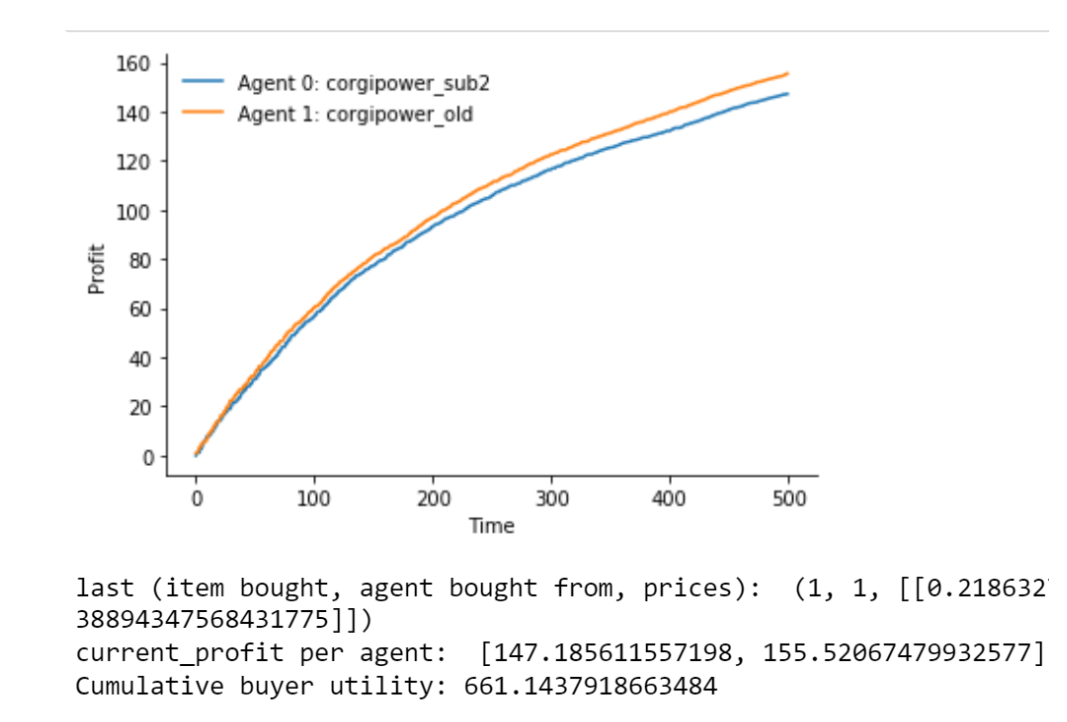
# **Appendix**

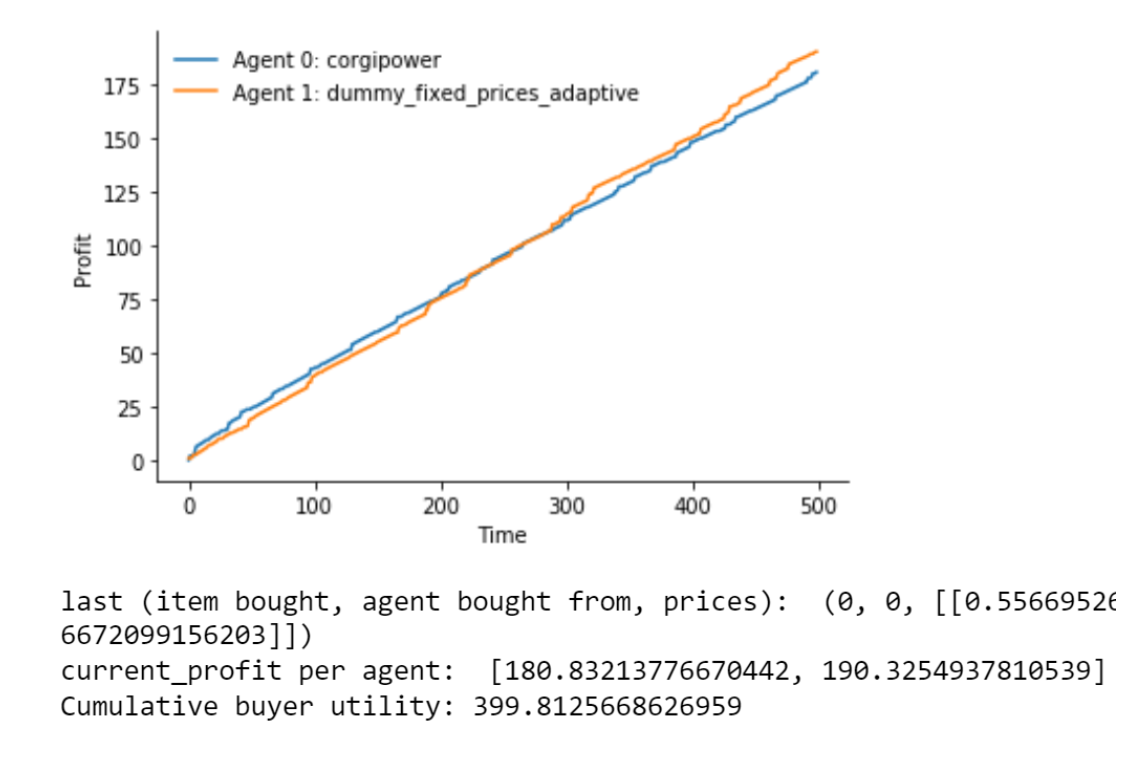


*Table 2. Dot Product of User Item Vectors*

**Part 1: Calculating Expected Revenue with Our Demand Model**

| **Our Expected Position** | **True Position** | **Team Name** | **True Revenue** | **Team's Expected Revenue** | **Our Expected Revenue** | **Absolute Difference in Expected Revenue** |
| --- | --- | --- | --- | --- | --- | --- |
| 15 | 1 | teamsvm | 4334.553 | 4289.279 | 4195 | 94.279 |
| 16 | 2 | thethreemusketeers | 4309.4 | 4151.414 | 4193 | 41.586 |
| 8 | 3 | datapeople | 4251.037 | 4119.667 | 4331 | 211.333 |
| 11 | 4 | tjmin | 4236.616 | 4148.034 | 4265 | 116.966 |
| 17 | 5 | pricegouge | 4204.874 | 4172.643 | 4187 | 14.357 |
| 13 | 6 | pgrk | 4172.16 | 4076.71 | 4226 | 149.29 |
| 12 | 7 | cs\_ppl\_suck | 4166.519 | 4287.109 | 4260 | 27.109 |
| 18 | 8 | corgipower | 4161.178 | 4110.191 | 4110 | 0.191 |
| 4 | 9 | harvestpea | 4147.505 | 4186.203 | 4439 | 252.797 |
| 14 | 10 | wolfpack | 4113.018 | 4049.616 | 4216 | 166.384 |
| 21 | 11 | cmgang | 4112.44 | 4509.293 | 3818 | 691.293 |
| 20 | 12 | booleanbetches | 4091.724 | 4145.599 | 3933 | 212.599 |
| 5 | 13 | yyds | 4039.74 | 4116.078 | 4439 | 322.922 |
| 22 | 14 | suibianwan | 4004.368 | 4514.776 | 3439 | 1075.776 |
| 6 | 15 | teamrocket | 4003.426 | 4044.636 | 4340 | 295.364 |
| 2 | 16 | arl | 3940.029 | 3973.201 | 4609 | 635.799 |
| 10 | 17 | johnandaliya | 3779.3 | 4378.804 | 4282 | 96.804 |
| 19 | 18 | grl\_power | 3670.316 | 3791.273 | 3995 | 203.727 |
| 9 | 19 | pricemaker | 3617.22 | 3572.486 | 4301 | 728.514 |
| 1 | 20 | espionage | 2944.696 | 3181.28 | 4683 | 1501.72 |
| 3 | 21 | noname | 2760.95 | 4290.567 | 4497 | 206.433 |
| 7 |  |  |  | 5238.179 | 4338 | 900.179 |

**

**