Regression Project Report - Total Sales for Each State and Month Prediction

Introduction

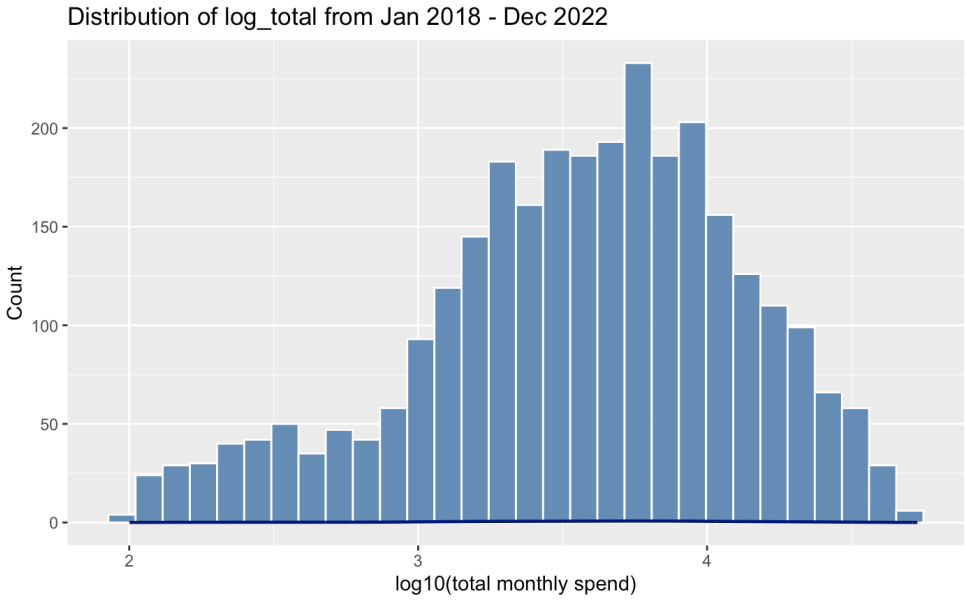
The goal of this research project is to predict *log_total* for each state-month-year of Amazon spending using a survey and order records from approximately 5000 Amazon customers from January 2018 to December 2022. In our model, we excluded *order_totals* to avoid data leakage, and included predictors such as time and geography (year, month, state), customer demographics (gender, age, income, overall count, etc.), and customer engagement (Amazon usage by customer and household).

In addition to these predictors from the raw data, we engineered new predictors from the supplementary datasets (*amazon_order_details*) that we believe are associated with the response variable *log_total*.

- ***month_est*:** estimated monthly spending; we took the average price of each category, multiplied by the quantity that was purchased, and summed all those up for the respective month
- ***month_var_ratio*:** price stability score; we took price variance for each category, divided by its average price, then averaged that ratio across the items bought in that respective month
- ***bin1_count…bin5_count*:** products sorted by price tier; we sorted categories by average price, then split into 5 bins/groups then counted the total number of purchased items for each group in that respective month
- ***month_est_low_var*:** stable category spend; similar approach as *month_est*, but we only included categories with low price variability

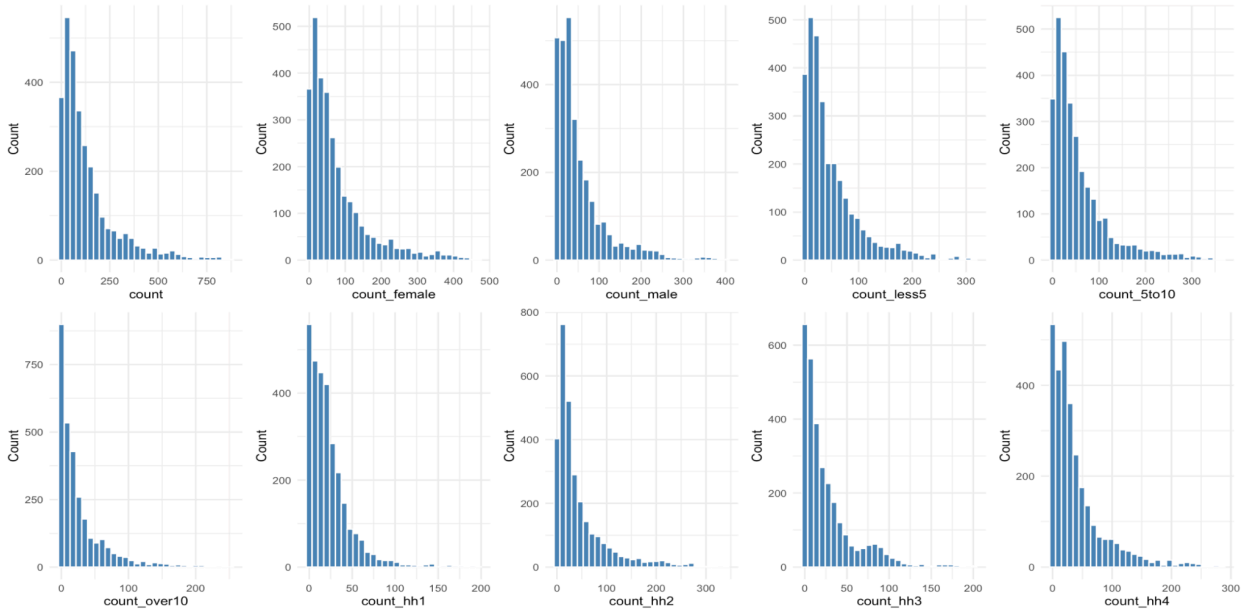
Combining all these predictors together into the main frames, *train_expanded* and *test_expanded*, allows us to develop accurate monthly *log_total* predictions across states from 2018 to 2022.

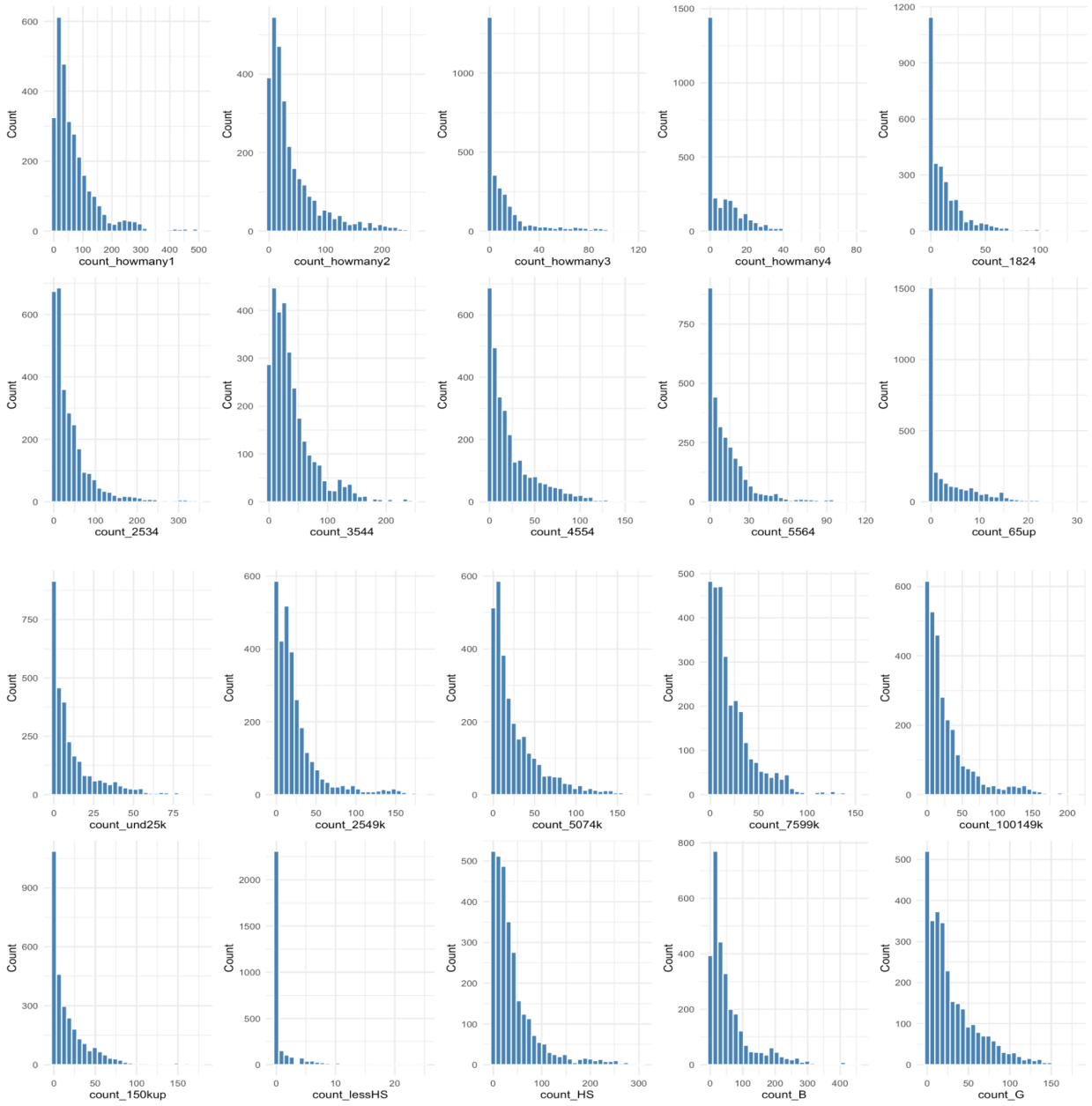
Exploratory Data Analysis



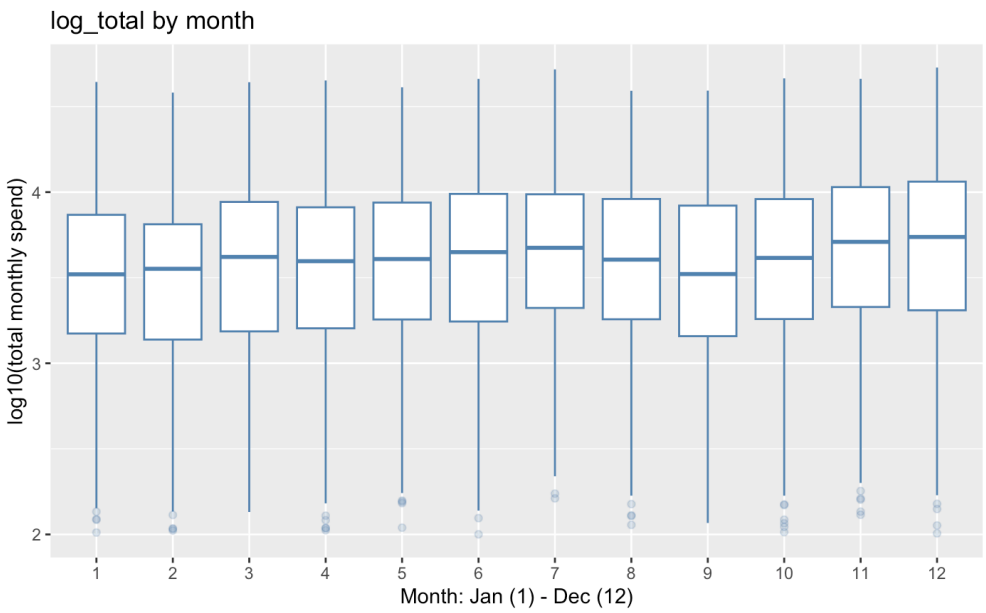
To understand our response variable *log_total*, we created a histogram to check for skew and outliers before modeling. Our x-axis is log10 of total monthly spend for each state X year X month, and our y-axis indicates the number of state-month in each bin. Based on the shape, it's single-peaked around 3.6-3.9, suggesting most state-months roughly spend $4k-$8k in total on Amazon. There is a slight skew on the left tail, but the log10 transformation has already reduced most extremes. This confirms that modeling on the log scale and using RMSE to predict *log_total* is appropriate.

Distributions of 30 predictors from train.csv

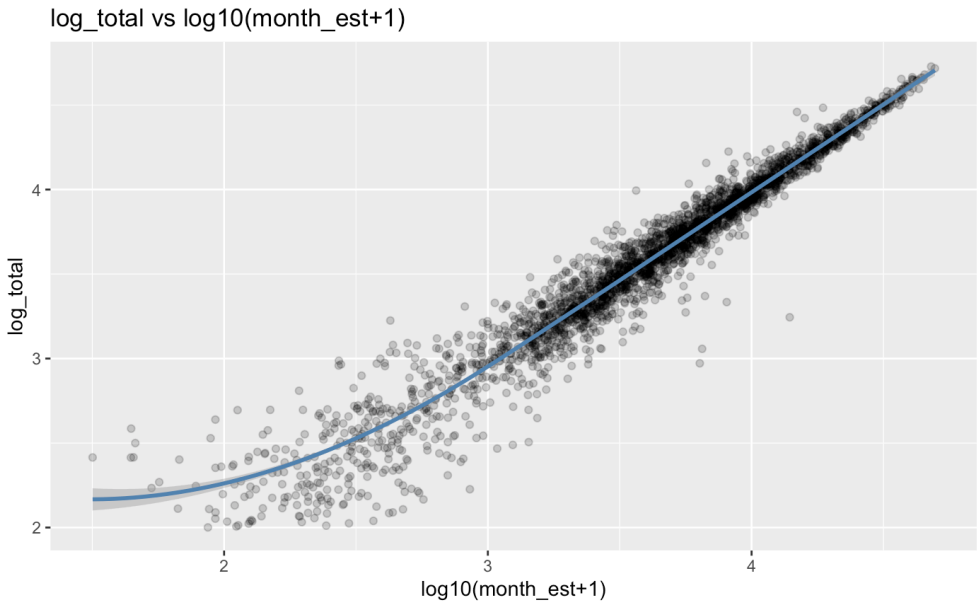




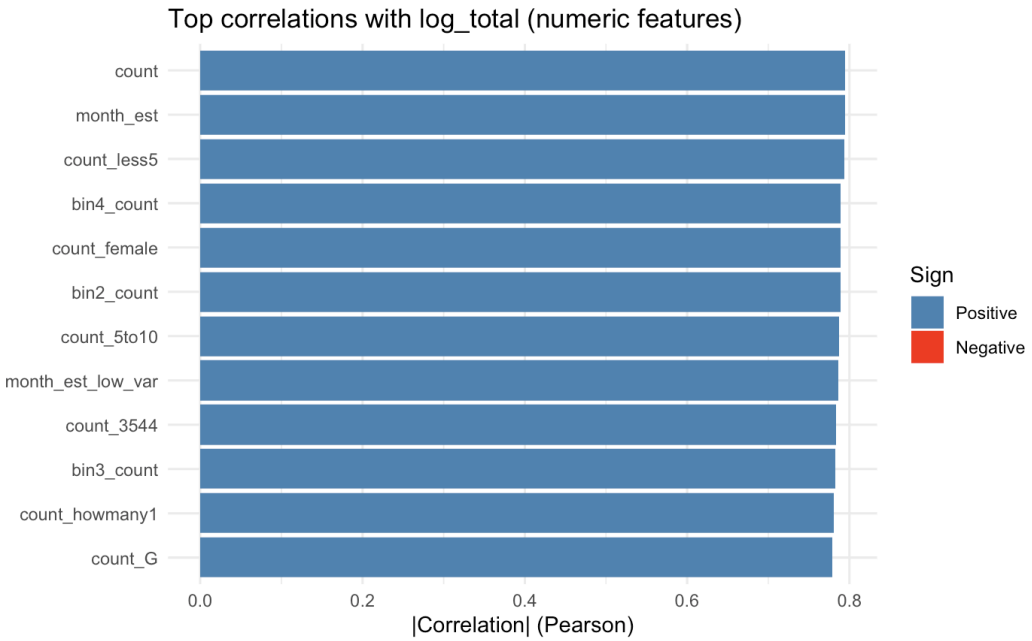
For all of the 30 numeric predictor variables, we plotted their distributions from train.csv, revealing a consistent right skew across the 30 variables. To address this, we explored recipes that applied a log transformation to all of these predictors, helping each distribution more closely resemble a normal distribution. Though, we did not include this transformation in our final code, as our highest performing models used for the Kaggle competition ended up being random forest and XGBoost models, which do not benefit from this transformation.



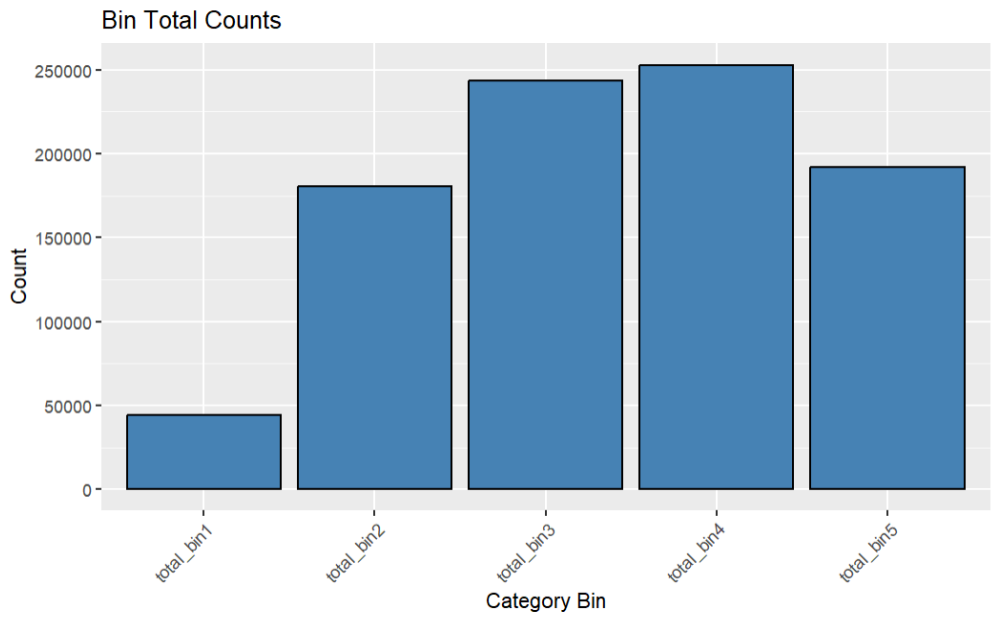
We created a month-by-month boxplot to see clear differences in distribution across all states and years for that corresponding month. The x-axis is "Month", starting from 1 = January to 12 = December, and our y-axis is log10 of total monthly spend for each state, year, and month. From our boxplot, we see that the median for each month varies with some months having wider boxes and whiskers, indicating greater variability in some months compared to others. This observation supports our preprocessing choice to convert the month to a factor instead of leaving it numeric, since the model can learn a separate effect for each month rather than assuming a single linear trend on 1-12.



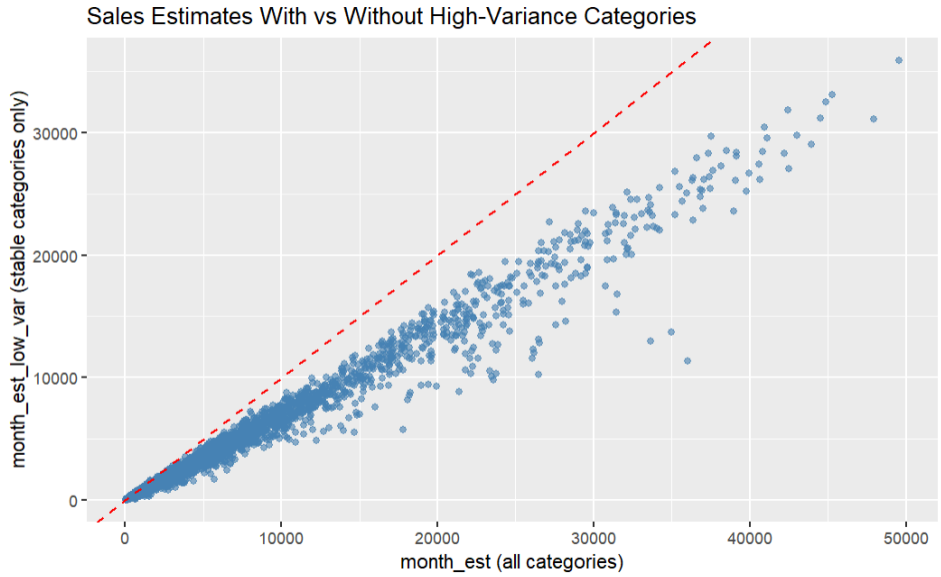
We created this scatterplot to check if one of our engineered predictors, *month_est*, is a good predictor for our model. We first log10(*month_est* + 1) since both axes are in log scale, making it easier to interpret the visualization. Also, we added an offset of 1 on log10(*month_est*) to handle zeros, preventing errors since log10(0) is undefined. Now looking at this plot, there is a little curve at the beginning, however, this is normal at small values near 0. As we go more to the right, the curve shifts into a linear upward line, suggesting a strong relationship between *month_est* and *log_total*. This is great for us because it supports keeping *month_est* in our model as it is a strong predictor for *log_total*.



This bar plot we created displays the top 12 numeric predictors on the y-axis, which support our model for predicting *log_total* before modeling. The x-axis is the absolute Pearson correlation from 0 to 1, where 0 represents no straight-line association and 1 represents perfect association. From this plot, month_est ranks near the top, matching our previous scatterplot, but we also see other engineered predictors that demonstrate strong associations, such as *bin4_count*, *bin2_count*, *month_est_low_var*, and *bin3_count*. This plot only analyzes predictors' correlation with *log_total* one at a time, so any other predictors that aren't on here could still be useful; hence, this is a great motivator for us to attempt the interaction recipe in our model.

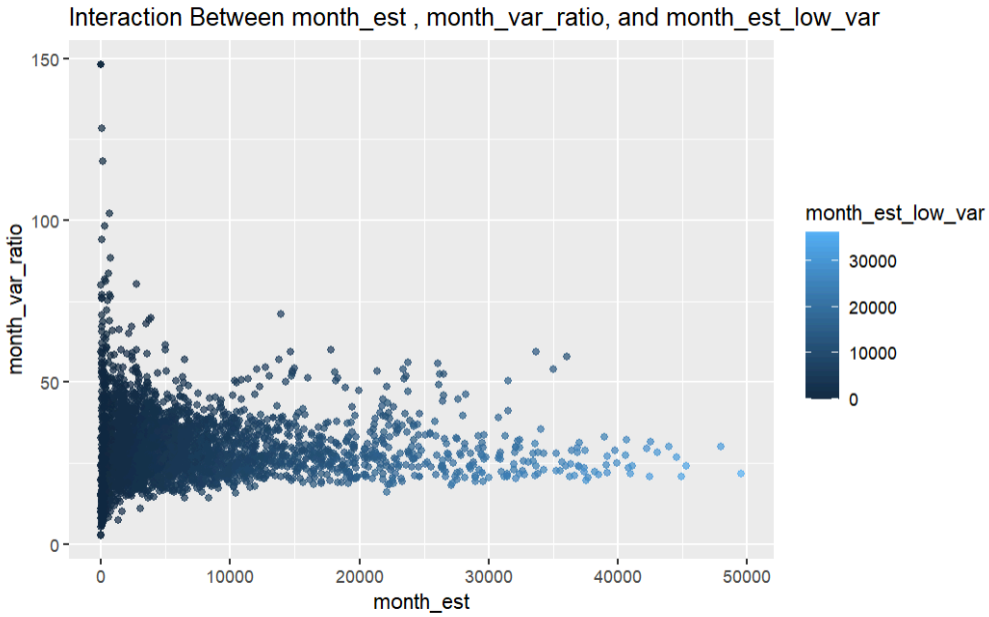


This bar plot shows the total orders within each of the five price bins, with Bin 1 representing the most expensive categories and Bin 5 the cheapest. The clear differences in counts suggest that price-based category bins matter. If all counts were equal, categories would contribute little predictive power, but the observed variance indicates that categories are an important variable for explaining and forecasting total sales.



This plot compares estimated sales using all categories (*month_est*) with estimates excluding high-variance categories (*month_est_low_var*). By looking at this plot, we see that most values fall under the dashed line, meaning total sales estimates (*month_est*) are larger than the stable-category estimates (*month_est_low_var*). This suggests volatile categories tend to inflate sales totals. Including both features allows the regression model to capture both stable demand

and additional contributions from unstable categories. This gave us the confidence to include both variables in our regression model.



This interaction plot suggests that higher sales estimates are linked to lower price variability and higher stable-category sales. As *month_est* increases, we also observe that *month_est_low_var* increases while *month_var_ratio* shows a downward trend. These observations provide exploratory evidence that meaningful interactions may exist among the three variables, potentially in both two-way and three-way forms.

Preprocessing

As mentioned earlier, we sought to utilize additional information from the *amazon_order_details* csv files, as these contained details on the categories of individual products ordered and the prices of individual units and orders. We saw this as a major opportunity for information gain, as the original *train* and *test* files primarily focus on the demographic information of customers in each observation. We included the three supplementary predictors in our final models, which are based on the average prices of a good (*cat_mean*) and the variance of the price within a category (*cat_var*) for each of the 1,818 categories represented in *amazon_order_details_train.csv*. The three supplementary predictors are calculated for each row (group of orders within a month, year, and state) in *train.csv* and *test.csv*, and summarized below:

- ***month_est*** = sum (*purchase_price_per_unit* * *quantity*)
- ***month_var_ratio*** = mean (*cat_var* / *cat_mean*)
- ***month_est_low_var*** = sum (*purchase_price_per_unit* * *quantity*)
    - Restricted to items of categories where *cat_var* / *cat_mean* < *45*

Utilizing these variables makes a key assumption that the mean price of each category of good is the same between the training and testing datasets, which we took to be reasonable after seeing major improvements in our model performance upon the variables' inclusion. We also chose to consider the two and three way interactions between these predictors because a high *month_var_ratio* for a given observation would imply a decreased reliability of month_est in predicting *order_totals*, and possibly an increased reliability of *month_est_low_var*.

Recipes

Recipe 1: generic_recipe

```
#create baseline recipe with all predictors in expanded dataset
generic_recipe <- recipe(log_total ~ ., data = train_expanded) %>%
      #convert month to factor
      step_mutate(month = factor(month)) %>%
      #want predictors numeric-only for RF and XGB
      step_dummy(all_nominal_predictors()) %>%
      #impute median to NA values
      step_impute_median(all_numeric_predictors())
```

Inside our *generic_recipe*, we applied three preprocessing operations - step_mutate(), step_dummy(), and step_impute_median() - in order to prepare us for the models.

- **step_mutate(month = factor(month)):** In the raw data, *month* was listed as a numeric value from 1-12 so if we were to leave it numeric, the models would treat months as values on a straight line, potentially creating an incorrect linear trend. We converted *month* into categorical levels "1"-"12" so that as a factor, the models can learn to distinguish the effect for each month without assuming any ordering.
- **step_dummy(all_nominal_predictors(), month):** We then converted all the categorical predictors (state, month, etc.) into 0/1 dummy columns so that it creates a numeric matrix for the model to work with. We know that most engines like XGBoost only accept numeric variables, so this step was to ensure the models run well, while still letting them learn to separate the effect for each level without imposing any numeric order on the categories.
- **step_impute_median(all_numeric_predictors()):** There were times where we saw NAs in our numeric columns, so this preprocessing operation fills any missing values with the median of the corresponding column from the training data. We did this because models can't train with NAs, so we ensure every column is complete. Deciding between median, mean, or mode, we decided median was best since the original predictors were often right-skewed, so median would have a more accurate value compared to mean, while

mode would create some issues since it works better for categorical data, which wasn't the case here after step_dummy().

Recipe 2: interact_recipe

```
#interact_recipe adds to generic_recipe interaction terms between our
supplementary variables
interact_recipe <- generic_recipe %>%
      #2 and 3 way interactions
      step_interact(terms = ~ (month_est +
                                 month_var_ratio +
                                 month_est_low_var)^3)
```

Inside our *interact_recipe*, we applied one preprocessing operation, *step_interact*, to add interaction features between our three engineered predictors: *month_est*, *month_var_ratio*, and *month_est_low_var*. We chose to limit interaction terms to these three predictors, as including interactions between every predictor would incur a large computational cost.

- **step_interact(terms = ~ (month_est + month_var_ratio + month_est_low_var)^3):** In this preprocessing operation, this step adds four new columns in addition to the three original columns of the engineered predictors. These four new columns showcase all interaction terms between the three predictors:
    - 2-way: *month_est* x *month_var_ratio*, *month_est* x *month_est_low_var*, *month_var_ratio* x *month_est_low_var*
    - 3-way: *month_est* x *month_var_ratio* x *month_est_low_var*

  If we didn't include interactions, the model would treat each predictor on its own and add them up, so with the interaction columns, the model can see the effect of one predictor change depending on another predictor (e.g. influence of *month_est* could be larger when *month_var_ratio* is high). We would later use cross-validation (CV) to check whether *interact_recipe* improves our RMSE score and to avoid overfitting.

Now, we're ready to use these preprocessed data to fit some models.

Candidate Models, Model Evaluation, & Tuning

Candidate Models Tested:

Model	Type of Model	Engine	Recipe	Hyperparameters
Generic Linear Regression Model	Linear Regression	lm	1	N/A
Interact Linear Regression Model	Linear Regression	lm	2	N/A
Generic Random Forest Model	Random Forest	ranger	1	trees = 977 min_n = 6 mtry = 18
Interact Random Forest Model	Random Forest	ranger	2	trees = 977 min_n = 6 mtry = 18
Generic XGBoost Model	XGBoost	xgboost	1	trees = 2829 tree_depth = learn_rate = 0.01076 min_n = 14 loss_reduction = 1.950372e-04 sample_size = 0.9857849 mtry = 28
Interact XGBoost Model	XGBoost	xgboost	2	trees = 2829 tree_depth = learn_rate = 0.01076 min_n = 14 loss_reduction = 1.950372e-04 sample_size = 0.9857849 mtry = 28

Generic Linear Regression Model

.metric	.estimator	mean	n	std_err	.config
rmse	standard	0.1384573	10	0.003450047	Preprocessor1_Model1
rsq	standard	0.9390964	10	0.002732265	Preprocessor1_Model1

With the generic recipe, we fit a simple linear regression that models log_total as an additive, straight-line function of each predictor without including any interactions. We used the generic linear regression model because it is simple and fast to run and can be used as a baseline to

compare to other complex models. The downside is that it is prone to underfitting compared to the other models as it may fail to capture nonlinear relationships present in this dataset.

Interact Linear Regression Model

.metric	.estimator	mean	n	std_err	.config
rmse	standard	0.1299861	10	0.002875808	Preprocessor1_Model1
rsq	standard	0.9464845	10	0.002149210	Preprocessor1_Model1

Similar to the generic linear regression model, but this time we use the interact recipe which implements the generic recipe and adds another step to the recipe. This extra step implements an interaction between three of our variables which creates four new interaction columns (three 2-way and one 3-way) into our linear regression model. We use this model to see if the interaction terms do help reduce RMSE compared to the RMSE in the generic linear regression model.

Generic Random Forest Model

mtry	trees	min_n	.metric	.estimator	mean	n	std_err	.config
12	517	5	rmse	standard	0.09690968	10	0.002599487	Preprocessor1_Model1
11	980	11	rmse	standard	0.09770195	10	0.002697944	Preprocessor1_Model2
12	870	4	rmse	standard	0.09686765	10	0.002644554	Preprocessor1_Model3
11	626	7	rmse	standard	0.09720612	10	0.002583496	Preprocessor1_Model4
18	977	6	rmse	standard	0.09603646	10	0.002549553	Preprocessor1_Model5

Using the generic recipe, we trained a Random Forest model, fitting many decision trees on random resamples of the data and taking the average of their predictions. This model captures nonlinear patterns well and afterwards, we made sure to check its 10-fold CV RMSE to see if its RMSE is better than the linear models.

Interact Random Forest Model

mtry	trees	min_n	.metric	.estimator	mean	n	std_err	.config
12	517	5	rmse	standard	0.09671484	10	0.002308941	Preprocessor1_Model1
11	980	11	rmse	standard	0.09746733	10	0.002317327	Preprocessor1_Model2
12	870	4	rmse	standard	0.09672972	10	0.002334544	Preprocessor1_Model3
11	626	7	rmse	standard	0.09701104	10	0.002265357	Preprocessor1_Model4
18	977	6	rmse	standard	0.09661157	10	0.002359414	Preprocessor1_Model5

With the same setup as the generic model, this time we use the interact recipe, which added four interaction columns (three 2-way and one 3-way). Although Random Forests may discover interactions and nonlinear patterns on their own, we still included the interaction terms to test if there is any improvement in the 10-fold CV RMSE for this model compared to the generic.

Generic XGBoost Model

mtry	trees	min_n	tree_depth	learn_rate	loss_reduction	sample_size	.metric	.estimator	mean	n	std_err	.config
2	869	17	10	0.0321620647	1.038791e-06	0.5058731	rmse	standard	0.09836340	10	0.002225042	Preprocessor1_Model1
16	1567	12	5	0.4662039320	1.107488e-01	0.6078755	rmse	standard	0.11105419	10	0.002614767	Preprocessor1_Model2
18	2040	7	8	0.0045811494	1.062894e+00	0.7521158	rmse	standard	0.11044101	10	0.002512579	Preprocessor1_Model3
28	2829	14	7	0.0107610031	1.950372e-04	0.9857849	rmse	standard	0.09271658	10	0.002522646	Preprocessor1_Model4
33	2357	3	4	0.0004111828	4.176206e-03	0.8513679	rmse	standard	1.18925881	10	0.001736902	Preprocessor1_Model5

XGBoost is a machine learning algorithm that uses decision trees like random forests, but differs in how those trees are built. Random forests grow many trees independently and average their results, while XGBoost builds trees sequentially, with each new tree correcting the errors of the previous ones. We believed that it would be one of the best tools to use in terms of predictive accuracy if parameters were properly tuned and implemented.

Interact XGBoost Model

mtry	trees	min_n	tree_depth	learn_rate	loss_reduction	sample_size	.metric	.estimator	mean	n	std_err	.config
2	869	17	10	0.0321620647	1.038791e-06	0.5058731	rmse	standard	0.09651361	10	0.001958791	Preprocessor1_Model1
16	1567	12	5	0.4662039320	1.107488e-01	0.6078755	rmse	standard	0.11034328	10	0.002209741	Preprocessor1_Model2
18	2040	7	8	0.0045811494	1.062894e+00	0.7521158	rmse	standard	0.11077707	10	0.002401466	Preprocessor1_Model3
28	2829	14	7	0.0107610031	1.950372e-04	0.9857849	rmse	standard	0.09318101	10	0.002441209	Preprocessor1_Model4
33	2357	3	4	0.0004111828	4.176206e-03	0.8513679	rmse	standard	1.18905318	10	0.001736476	Preprocessor1_Model5

While XGBoost can capture interactions implicitly through tree splits, the interact recipe allowed us to test whether providing these interactions directly would improve performance. We were optimistic that the interaction terms would highlight important relationships. However, we still needed to test whether these interactions would offer a measurable benefit once combined with XGBoost's own tree-based structure.

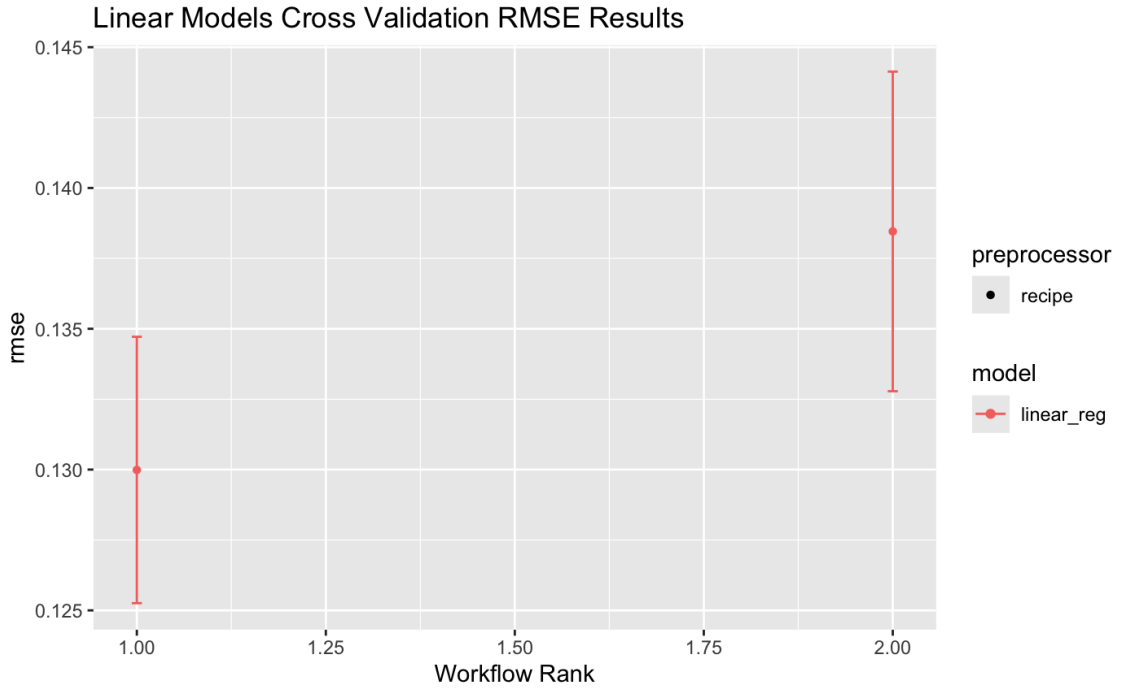
Discussion of Candidate Models

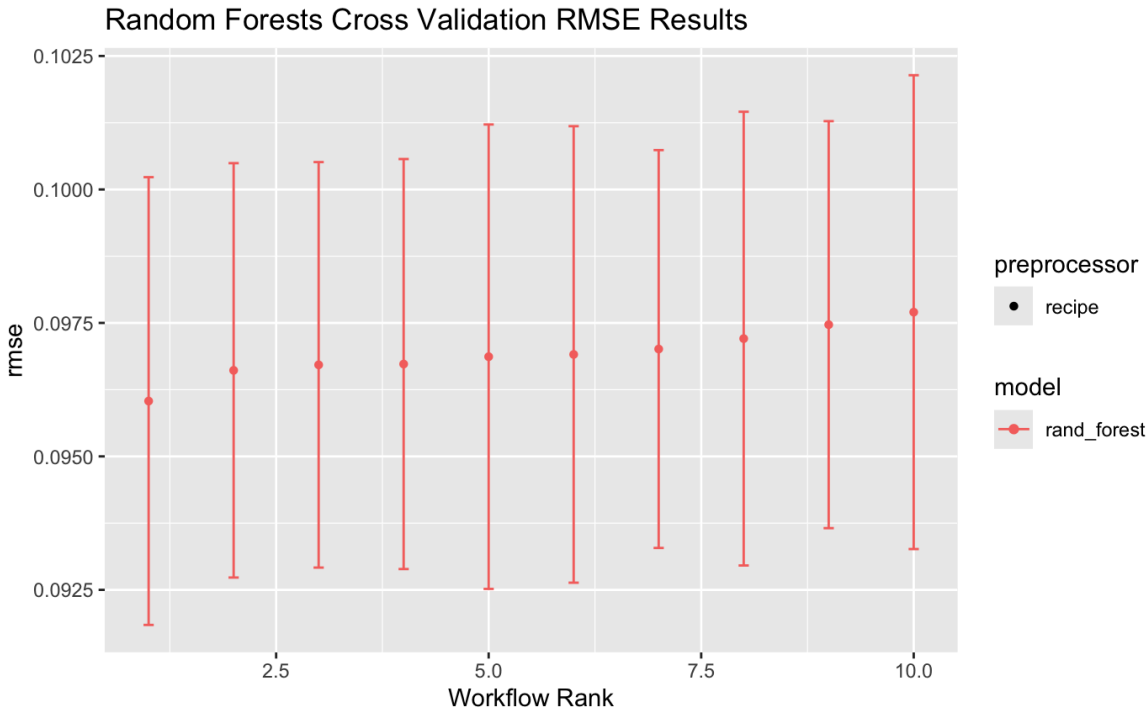
We tested several models with two kinds of preprocessing: a generic recipe and an interaction recipe. After, we compared all models using 10-fold cross-validation with stratification on log_total and the Generic XGBoost model clearly performed best. It had the lowest RMSE ≈ 0.0927 with a small standard error of around 0.0025. It beats Generic Random Forest with RMSE ≈ 0.0960 and Generic Linear Regression with RMSE ≈ 0.1385 and RMSE ≈ 0.1300 with interactions. We also tried XGBoost with the interaction recipe, but that version was slightly worse with RMSE ≈ 0.0932. This tells us that tree models like XGBoost already learn interactions through their splits, so the simpler generic recipe was the right choice here.

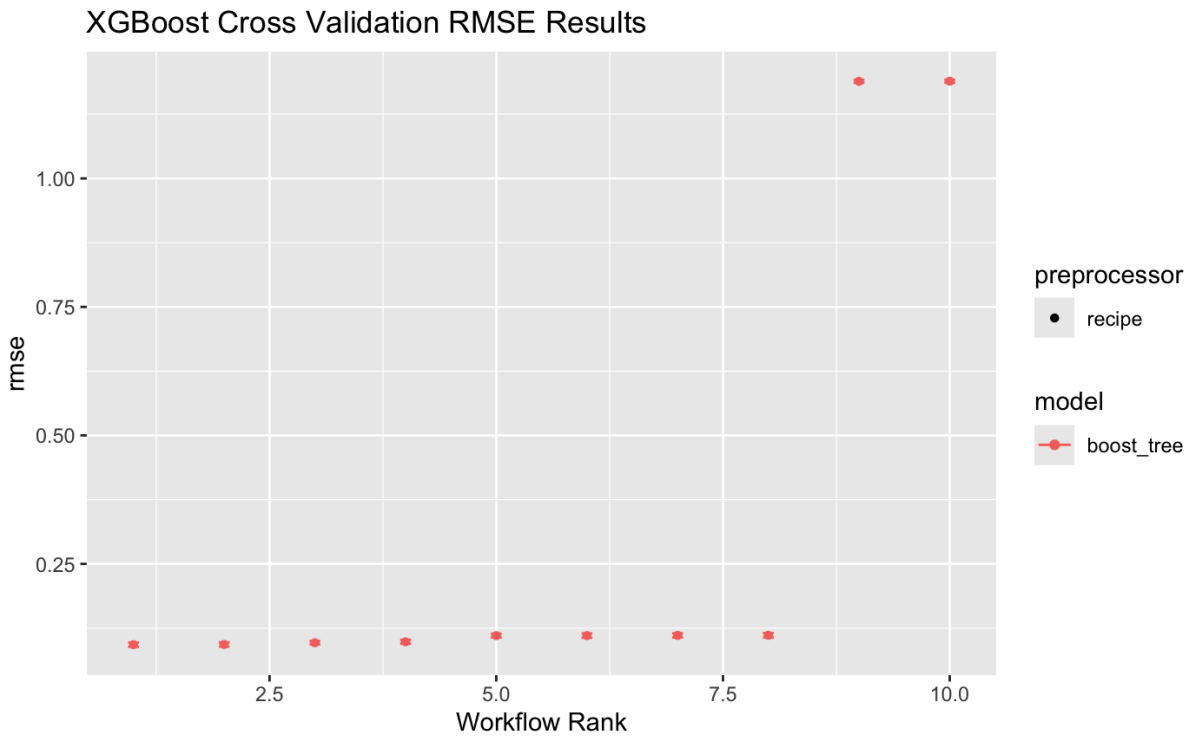
Model Performance:

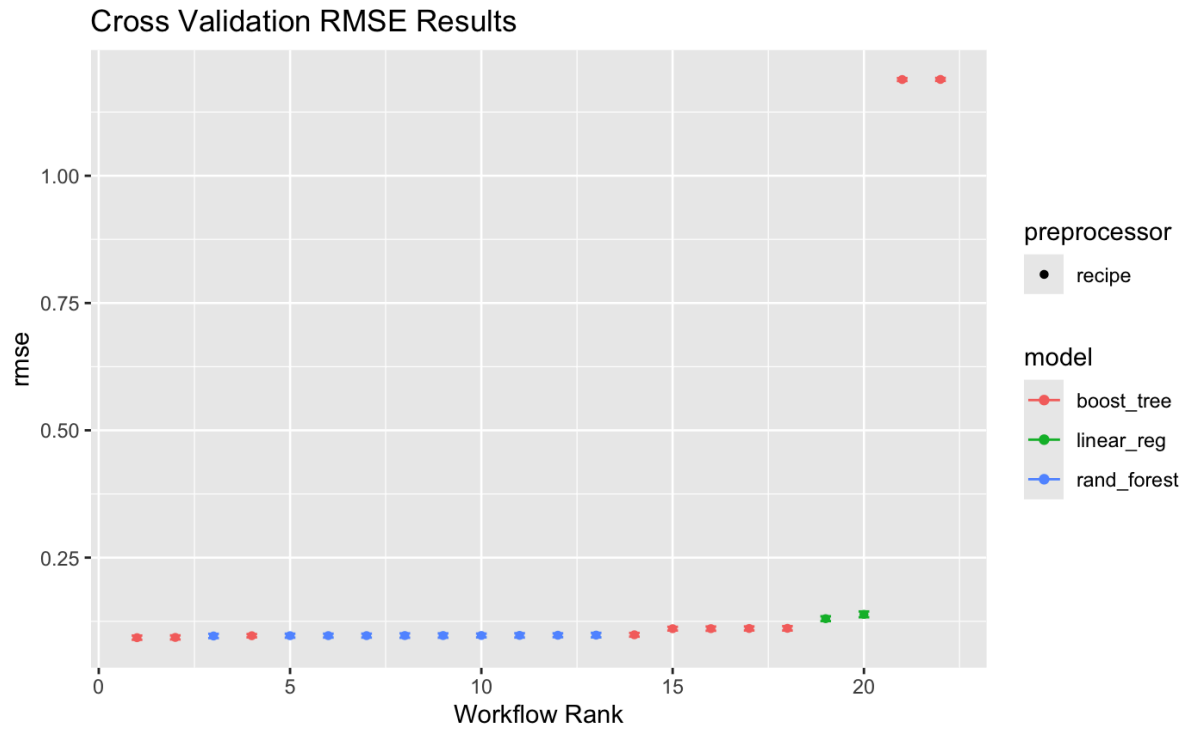
Model	RMSE	SE of Metric (if applicable)
Generic Linear Regression Model	0.1384573	0.003450047
Interact Linear Regression Model	0.1299861	0.002875808
Generic Random Forest Model	0.09603646	0.002549553

Interact Random Forest Model	0.09661157	0.002359414
Generic XGBoost Model	0.09271658	0.002522646
Interact XGBoost Model	0.09318101	0.002441209






XGBoost Cross Validation RMSE Results
rmse
1.00
0.75
0.50
0.25
2.5
5.0
7.5
10.0
Workflow Rank
preprocessor
recipe
model
boost_tree


Cross Validation RMSE Results
rmse
1.00
0.75
0.50
0.25
0
5
10
15
20
Workflow Rank
preprocessor
recipe
model
boost_tree
linear_reg
rand_forest

Discussion of Final Model

In these results, the Generic XGBoost model was best. It achieved the lowest RMSE = 0.0927, beating the Random Forest with RMSE ≈ 0.0966 on both generic recipe and interact recipe, and Linear Regression with RMSE ≈ 0.1385 on generic recipe and RMSE ≈ 0.1300 on interact recipe. On the other hand, XGBoost with the interaction recipe was slightly worse with RMSE ≈ 0.0932, which suggests the tree might already learn interactions through its own splits. The final model was tuned with a small Latin-hypercube grid of size = 5 per workflow with hyperparameters: trees ≈ 2,829, tree_depth = 7, learn_rate ≈ 0.01076, min_n = 14, loss_reduction ≈ 1.95e^-4, sample_size ≈ 0.9858, and mtry = 28.

The model's main strengths are clear in our results. First, it had the best accuracy in our table as it has the lowest RMSE. Second, it looked stable across folds because the standard error was small of around 0.0025, so one lucky split did not drive the score. Third, it worked well with simple preprocessing; we did not need many extra interaction features for the tree model, which keeps our pipeline easier to understand and faster to run. We also saw two weaknesses. First, XGBoost can be sensitive to hyperparameters; in the XGBoost CV scatter plots, a few configurations spiked to very large errors, which shows how the model is prone to overfitting. Second, our tuning grid was very small as we only used five combinations per workflow, which kept runtime short but may have missed better settings near the best region. However, despite the weaknesses, we managed to tune the model to the right direction that outperformed all of the other models.

There are several practical improvements we can make while staying inside the data and code we already have. With more computational power, we can expand the tuning search around the best area in depth near 7, trees around 2.8k–3k, smaller learn rate, and try more than five combinations as our code already supports a larger grid. We can try the simple mean ensemble already written in our script, where we average the two XGBoost predictions to reduce variance and make results more stable. In addition, because we have year and month, we can switch to time-aware cross-validation (blocked or rolling) so training folds come from earlier months and validation folds from later months; this matches how the data change over time and gives a more honest view of generalization. Also, we can add lagged and rolling features from the monthly signals we already compute. For example, last month's month_est or a 3-month average of month_var_ratio without using any outside data. Then, we can test the cutoff used in month_est_low_var (cat_var_ratio < 45) by trying nearby values to see if the score improves. Finally, since all joins use state, year, and month, we should audit the joins for groups with few or no orders and confirm that median imputation behaves as expected in both train and test; cleaning up these edge cases can protect accuracy. If we later collect more rows of the same kind of data, for example, more months with the same columns or better coverage for state-month groups with few orders, the model should learn seasonal and state patterns even better without changing our overall design.

Appendix: Final Annotated Script

Note: Kaggle Competition Script Submission (regression_xgboost.Rmd) reproduces the exact csv file submitted to Kaggle, if needed for score verification. Combined Model Script is a more streamlined version of our entire process training XGBoost, Random Forests, and Multiple Linear Regression. It includes XGBoost models of similar performance to our Kaggle submission, but we had issues with reproducibility when streamlining the code, so the output csv files are slightly different from our best Kaggle files.

Kaggle Competition Script Submission
(XGBoost Method only)

```
+ Load the tidyverse and tidymodels

```{r}
library(tidyverse)
library(tidymodels)
library(lubridate)
library(workflowsets)
library(dials)
```

Read in data

```{r}
train <- read_csv("train.csv") # Main training dataset (with the target variable)
test <- read_csv("test.csv") # Test dataset (no target column, only features))


# Order-level details (rows are orders) for train/test
train_orders <- read_csv("amazon_order_details_train.csv")
test_orders <- read_csv("amazon_order_details_test.csv")

# Customer information for train/test
train_customers <- read_csv("customer_info_train.csv")
test_customers <- read_csv("customer_info_test.csv")

```

Data Mining:

Summarize and group categories by their average order cost:

```{r}
category_means <- train_orders %>%
  group_by(category) %>%                                        # group all rows by product
category
  summarize(
    cat_mean = mean(purchase_price_per_unit, na.rm = TRUE),     # average unit price
    cat_var  = var(purchase_price_per_unit,  na.rm = TRUE),     # price variance
    n = n(),                                                    # number of rows
    cat_var_ratio = cat_var / cat_mean                          # stability score
  ) %>%
  mutate(cat_bin = ntile(desc(cat_mean), 5)) %>%                #  split into 5 price tiers
  arrange(desc(cat_bin))                                        #  show expensive tiers first
```
```

Create supplementary predictors, bind to original train and test datasets:

```{r}
# Train supplement
train_supplement <- train_orders %>%
  mutate(
    year = year(order_date), # extract year from date
    month = month(order_date), # extract month (from Jan to Dec)
     # Turn state abbr -> full name
    q_demos_state = state.name[match(shipping_address_state, state.abb)]
  ) %>%
  left_join(category_means, by = "category") %>% # add category stats to each order
  left_join(train_customers %>% select(-q_demos_state), #  add customer info (avoid duplicate
col)
            by = "survey_response_id") %>%
  group_by(q_demos_state, year, month) %>%                       # aggregate to state x year x
month
  summarize(
    month_est        = sum(cat_mean * quantity, na.rm = TRUE),   # est. spend = avg price *
qty
    month_var_ratio  = mean(cat_var_ratio, na.rm = TRUE),        # avg stability
    bin1_count       = sum(cat_bin == 1, na.rm = TRUE),          # count by price tier
    bin2_count       = sum(cat_bin == 2, na.rm = TRUE),
    bin3_count       = sum(cat_bin == 3, na.rm = TRUE),
    bin4_count       = sum(cat_bin == 4, na.rm = TRUE),
    bin5_count       = sum(cat_bin == 5, na.rm = TRUE),
    month_est_low_var = sum(cat_mean * quantity *                # stable spend
                            (cat_var_ratio < 45), na.rm = TRUE),
    .groups = "drop"
  )

# Test supplement
test_supplement <- test_orders %>%
  mutate(
    year = year(order_date),
    month = month(order_date),
    q_demos_state = state.name[match(shipping_address_state, state.abb)]
  ) %>%
  left_join(category_means, by = "category") %>%
  left_join(test_customers %>% select(-q_demos_state), by = "survey_response_id") %>%
  group_by(q_demos_state, year, month) %>%
  summarize(
    month_est        = sum(cat_mean * quantity, na.rm = TRUE),
    month_var_ratio  = mean(cat_var_ratio, na.rm = TRUE),
    bin1_count       = sum(cat_bin == 1, na.rm = TRUE),
    bin2_count       = sum(cat_bin == 2, na.rm = TRUE),
    bin3_count       = sum(cat_bin == 3, na.rm = TRUE),
    bin4_count       = sum(cat_bin == 4, na.rm = TRUE),
    bin5_count       = sum(cat_bin == 5, na.rm = TRUE),
    month_est_low_var = sum(cat_mean * quantity * (cat_var_ratio < 45), na.rm = TRUE),
    .groups = "drop"
  )

# Join to main frames
train_expanded <- train %>%
  left_join(train_supplement, by = c("q_demos_state", "year", "month")) # match rows by keys

test_expanded <- test %>%
  left_join(test_supplement, by = c("q_demos_state", "year", "month")) # same for test
```

Prevent data leakage

```{r}
# order_totals is too closely tied to the target (log_total), so remove it

train_expanded <- train_expanded %>% select(-order_totals) # drop the leaking column
```


Model Cross Validation

```{r}
set.seed(123)
# make 10 folds, stratified on log_total so folds have similar target distribution
train_folds <- vfold_cv(train_expanded, v = 10, strata = log_total)
```

```{r}
head(train_expanded) # quick peek at the expanded training data
```

Recipes

```{r}
generic_recipe <- recipe(log_total ~ ., data = train_expanded) %>%
  step_mutate(month = factor(month)) %>% # treat month as a category, not a number line
  step_dummy(all_nominal_predictors()) %>% # convert categories to 0/1 dummy columns
  step_impute_median(all_numeric_predictors()) # fill missing numbers with median

interact_recipe <- generic_recipe %>%
  # create interaction and polynomial combos up to 3rd order among these three features
  step_interact(terms = ~ (month_est + month_var_ratio + month_est_low_var)^3)

# pack both recipes so we can try both with the same model
recipes <- list(
  generic  = generic_recipe,
  interact = interact_recipe
)
```

XGBoost model setup

```{r}
xgb_model <- boost_tree(
  trees          = tune(),  # to be tuned: how many trees to grow
  tree_depth     = tune(),  # to be tuned: how deep each tree can go
  learn_rate     = tune(),  # to be tuned: step size for learning
  min_n          = tune(),  # to be tuned: min samples in a leaf (smaller -> more splits)
  loss_reduction = tune(),  # to be tuned: min loss improvement (gamma) to split
  sample_size    = tune(),  # to be tuned: row subsampling ratio per boosting round
  mtry           = tune()   # to be tuned: number of predictors tried at each split
) %>%
  set_mode("regression") %>%  # we predict a number (log_total)
  set_engine("xgboost")       # use xgboost backend

```

```{r}
# Build all recipe × model combinations in one set
xgb_set <- workflow_set(
  preproc = recipes, # try both recipes
  models  = list(xgb = xgb_model), # with the same xgb model spec
```

```
  cross   = TRUE # make both combos: generic_xgb, interact_xgb
)
```


Build tuning grids for each workflow

```{r}
# get only the XGB workflow IDs (e.g., "generic_xgb", "interact_xgb")
xgb_ids <- xgb_set$wflow_id[grepl("_xgb$", xgb_set$wflow_id)]

for (id in xgb_ids) {
  wf_i <- extract_workflow(xgb_set, id) #   pull out this recipe+model combo

  pset <- extract_parameter_set_dials(wf_i) %>%
    update(
      trees          = trees(c(800, 3000)), #  try 800 ~ 3000 trees
      tree_depth     = tree_depth(c(3, 10)), # try tree depths 3 ~ 10
      learn_rate     = learn_rate(range = c(-3.5, -0.3)),# log10 scale (10^-3.5 ~ 10^-0.3)
      min_n          = min_n(c(1, 20)), # leaf size 1 ~ 20
      loss_reduction = loss_reduction(c(-6, 1)), # log10 scale for gamma
      sample_size    = sample_prop(c(0.5, 1)),  # subsample 50% ~ 100% of rows
        # set an appropriate upper bound for mtry based on the data columns
      mtry           = finalize(mtry(), train_expanded %>% select(-log_total))
    )

  set.seed(123)
 # make a small but diverse grid of 5 settings using Latin hypercube sampling
  grd <- grid_latin_hypercube(pset, size = 5)

  # attach this parameter info and grid to the workflow set
  xgb_set <- option_add(xgb_set, id = id, param_info = pset, grid = grd)
}
```

Train with tuning(10-fold CV over the grid)

```{r}
set.seed(123)
xgb_res <- xgb_set %>%
  workflow_map(
    "tune_grid", #  grid search
    resamples = train_folds,  # using the 10-fold CV we made
    metrics   = metric_set(rmse), # evaluate with RMSE (notice that lower is better!)
    control   = control_grid(save_pred = TRUE, # keep CV predictions (optional diagnostics)
                       parallel_over = "resamples"),  # parallelize over folds if a backend is
set
    seed      = 123,
    verbose   = TRUE
  )
```

Collect performance

```{r}
performance <- xgb_res %>%
  collect_metrics() %>%                 # gather CV scores for all grid points
  dplyr::filter(.metric == "rmse") %>%  # keep only RMSE rows
  arrange(mean) %>%                     # sort by best average RMSE
  dplyr::select(wflow_id, mean)         # show which workflow did best on average

print(performance, n = 10)
```

```

### Finalize best settings and fit on full train
```{r}
pred_list <- list()

for (id in xgb_ids) {
  # pull CV results for this workflow and pick the best hyperparameters
  res_i  <- extract_workflow_set_result(xgb_res, id)
  best_i <- tune::select_best(res_i, metric = "rmse")

  # lock the workflow to the best settings, then fit on ALL training data
  wf_i       <- extract_workflow(xgb_set, id)
  final_wf_i <- finalize_workflow(wf_i, best_i)
  fit_i      <- fit(final_wf_i, data = train_expanded)

  # quick training RMSE to sanity-check over/underfit (also, notice that lower is better)
  tr_rmse <- predict(fit_i, train_expanded) %>%
    bind_cols(train_expanded) %>%
    rmse(truth = log_total, estimate = .pred)
  cat(sprintf("[Train RMSE | %s] %.5f\n", id, tr_rmse$.estimate))

  # make predictions on the test set; name column uniquely for each workflow
  pred_i <- predict(fit_i, test_expanded) %>%
    bind_cols(test_expanded) %>%
    select(id, .pred) %>%
    rename(!!paste0("pred_", id) := .pred)

  pred_list[[id]] <- pred_i
}

# combine predictions (left joins by id) so we can preview side by side
pred_merged <- Reduce(function(x, y) dplyr::left_join(x, y, by = "id"), pred_list)
head(pred_merged, 15)
```

Write submission files
```{r}

for (id in xgb_ids) {
  pred_i <- pred_list[[id]]

  submission_i <- pred_i
  names(submission_i)[2] <- "log_total"      #  rename to match submission format

  out_file <- sprintf("sub_%s_%s.csv", id, Sys.Date()) # file per workflow, dated
  write_csv(submission_i, out_file)
  cat(sprintf("\n[Wrote] %s\n", out_file))
  print(head(submission_i, 15))
}

# (optional) simple ensemble: mean of the two
# ensemble <- pred_merged %>%
#   mutate(log_total = rowMeans(dplyr::select(., starts_with("pred_")), na.rm = TRUE)) %>%
#   select(id, log_total)
# write_csv(ensemble, sprintf("sub_ensemble_mean_%s.csv", Sys.Date()))
```
```

Combined Models Script
Combined (XGBoost method + Random Forest method + Multiple Linear Regression)

+ Load the tidyverse and tidymodels

```{r}
library(tidyverse)
library(tidymodels)
library(lubridate)      # year(), month()
library(workflowsets)   # workflow_set(), workflow_map()
library(dials)          # parameter ranges, grids
```

Read in data

```{r}
train <- read_csv("train.csv")
test <- read_csv("test.csv")
train_orders <- read_csv("amazon_order_details_train.csv")
test_orders <- read_csv("amazon_order_details_test.csv")
train_customers <- read_csv("customer_info_train.csv")
test_customers <- read_csv("customer_info_test.csv")
```

Data Mining:

Summarize and group categories by their average order cost:

```{r}

category_means <- train_orders %>% #detailed individual orders
  group_by(category) %>% #group by 1818 categories
  summarize(
    cat_mean = mean(purchase_price_per_unit, na.rm = TRUE), #average price of one unit within
category
    cat_var  = var(purchase_price_per_unit,  na.rm = TRUE), #variance of prices within
category
    n = n(), #number of items in category
    cat_var_ratio = cat_var / cat_mean #ratio between variance and price within category
  ) %>%
  mutate(cat_bin = ntile(desc(cat_mean), 5)) %>% # 5 bins for different categories organized
by price (this wasn't used in order_est, just another idea)
  arrange(desc(cat_bin)) #arrange categories in descending order by price

```

Create supplementary predictors, bind to original train and test datasets:

```{r}

### Train supplement: Create supplementary training columns
train_supplement <- train_orders %>%
  mutate(
    year = year(order_date), #organize date by year and month
    month = month(order_date),
    q_demos_state = state.name[match(shipping_address_state, state.abb)] #converts two letter
state code to state name (CA -> California)
  ) %>%
```

```
  left_join(category_means, by = "category") %>% #join order category data
  left_join(train_customers %>% select(-q_demos_state), by = "survey_response_id") %>% #join
customer data by survey_response_id
  group_by(q_demos_state, year, month) %>% # restructure supplement table to 1 row for each
month/year/state (format of main training dataset)
  summarize(
    month_est = sum(cat_mean * quantity, na.rm = TRUE), #create estimate of sales based on
category means for each month/year/state
    month_var_ratio = mean(cat_var_ratio, na.rm = TRUE), #mean cat_var_ratio for each
month/year/state, an uncertainty metric for month_est
    month_est_low_var = sum(cat_mean * quantity * (cat_var_ratio < 45), na.rm = TRUE), #same
as month_est but exclude categories with high variance in price relative to category mean
price
    bin1_count = sum(cat_bin == 1, na.rm = TRUE), #number of sales in each category-price-bin
    bin2_count = sum(cat_bin == 2, na.rm = TRUE),
    bin3_count = sum(cat_bin == 3, na.rm = TRUE),
    bin4_count = sum(cat_bin == 4, na.rm = TRUE),
    bin5_count = sum(cat_bin == 5, na.rm = TRUE),
    .groups = "drop"
  )

### Test supplement: Create supplementary testing columns
test_supplement <- test_orders %>%
  mutate(
    year = year(order_date), #organize date by year and month
    month = month(order_date),
    q_demos_state = state.name[match(shipping_address_state, state.abb)] #converts two letter
state code to state name (CA -> California)
  ) %>%
  left_join(category_means, by = "category") %>% #join order category data
  left_join(test_customers %>% select(-q_demos_state), by = "survey_response_id") %>%  #join
customer data by survey_response_id
  group_by(q_demos_state, year, month) %>%  # restructure supplement table to 1 row for each
month/year/state (format of main training dataset)
  summarize(
    month_est = sum(cat_mean * quantity, na.rm = TRUE), #create estimate of sales based on
category means for each month/year/state
    month_var_ratio = mean(cat_var_ratio, na.rm = TRUE), #mean cat_var_ratio for each
month/year/state, an uncertainty metric for month_est
    month_est_low_var = sum(cat_mean * quantity * (cat_var_ratio < 45), na.rm = TRUE), #same
as month_est but exclude categories with high variance in price relative to category mean
price
    bin1_count = sum(cat_bin == 1, na.rm = TRUE), #number of sales in each category-price-bin
    bin2_count = sum(cat_bin == 2, na.rm = TRUE),
    bin3_count = sum(cat_bin == 3, na.rm = TRUE),
    bin4_count = sum(cat_bin == 4, na.rm = TRUE),
    bin5_count = sum(cat_bin == 5, na.rm = TRUE),
    .groups = "drop"
  )

### Join supplemental data to main train and test data frames
train_expanded <- train %>%
  left_join(train_supplement, by = c("q_demos_state", "year", "month"))

test_expanded <- test %>%
  left_join(test_supplement, by = c("q_demos_state", "year", "month"))
```

Remove the variable `order_totals` as it will allow you to predict `log_total` perfectly

```{r}
```

```
train_expanded <- train_expanded %>% select(-order_totals)

```

Model Cross Validation

Fold the training data into a 10-fold cross-validation set. Stratify on `log_total`. Set seed 123.

```{r}

set.seed(123)
train_folds <- vfold_cv(train_expanded, v = 10, strata = log_total)

```

```{r}

head(train_expanded) #show head of full expanded training dataset

```

Create recipes for model fitting:

```{r}

generic_recipe <- recipe(log_total ~ ., data = train_expanded) %>% #create baseline recipe with all predictors in expanded dataset
  step_mutate(month = factor(month)) %>% #convert month to factory
  step_dummy(all_nominal_predictors()) %>% # want predictors numeric-only for RF and XGB
  step_impute_median(all_numeric_predictors()) #impute median to NA values (common in var_ratio for categories with only 1 observation)

interact_recipe <- generic_recipe %>% #interact_recipe adds to generic_recipe interaction terms between our supplementary variables
  step_interact(terms = ~ (month_est + month_var_ratio + month_est_low_var)^3) #2 and 3 way interactions

recipes <- list(
  generic  = generic_recipe,
  interact = interact_recipe
)

```

Create models, tuning, and workflows:

```{r}

lm_model = linear_reg() %>% #prepare multiple linear regression
  set_engine('lm') %>%
  set_mode('regression')

#prepare random forest with tunable parameters trees, mtry, min_n
rf_model <- rand_forest(
  trees=tune(),
  mtry=tune(),
  min_n=tune()
  ) %>%
  set_engine('ranger') %>%
  set_mode('regression')
```

```
#prepare XGBoost with tunable parameters trees, tree_depth, etc.
xgb_model <- boost_tree(
  trees          = tune(),
  tree_depth     = tune(),
  learn_rate     = tune(),
  min_n          = tune(),
  loss_reduction = tune(),
  sample_size    = tune(),
  mtry           = tune()
) %>%
  set_mode("regression") %>%
  set_engine("xgboost")

# define parameter ranges for random forest
rf_params <- parameters(
  mtry(range = c(10L, 20L)), # restricts mtry search space
  trees(range = c(500, 1000)), # restrict trees
  min_n(range = c(2L, 15L)) #restricts min_n
)

rf_grid <- grid_random(rf_params, size = 5) #randomly generate 5 parameter combinations for
random forest cv
# for this we originally considered 30 candidates, but this was reduced to 5 for demonstration
purposed and to improve runtime

```

Create workflow sets. The three models have separate workflows because we approach tuning slightly differently

```{r}

lm_set <- workflow_set(
  preproc = recipes,
  models = list(lm = lm_model),
  cross = TRUE
)

rf_set <- workflow_set(
  preproc = recipes,
  models = list(rf = rf_model),
  cross = TRUE
)

xgb_set <- workflow_set(
  preproc = recipes,
  models  = list(xgb = xgb_model),
  cross   = TRUE
)

```

Prepare tuning grid for XGBoost

```{r}

xgb_ids <- xgb_set$wflow_id[grepl("_xgb$", xgb_set$wflow_id)]

for (id in xgb_ids) {
  wf_i <- extract_workflow(xgb_set, id)

  pset <- extract_parameter_set_dials(wf_i) %>%
```

```
    update(
      trees          = trees(c(800, 3000)),
      tree_depth     = tree_depth(c(3, 10)),
      learn_rate     = learn_rate(range = c(-3.5, -0.3)),  # 10^-3.5 ~ 10^-0.3
      min_n          = min_n(c(1, 20)),
      loss_reduction = loss_reduction(c(-6, 1)),           # 10^-6 ~ 10^1
      sample_size    = sample_prop(c(0.5, 1)),
      mtry           = finalize(mtry(), train_expanded %>% select(-log_total))
    )

  set.seed(123)
  grd <- grid_latin_hypercube(pset, size = 5)

  xgb_set <- option_add(xgb_set, id = id, param_info = pset, grid = grd)
}

```

Perform cross validation on all mlr and rf model candidates (roughly 4-5 minutes to run)

```{r}

lm_fit <- lm_set %>% #fit models with 10 fold cv and all recipe combinations
  workflow_map('fit_resamples',
               seed = 123,
               verbose = TRUE,
               resamples = train_folds
               )

rf_fit <- rf_set %>% #fit models with 10 fold cv and all recipe combinations
  workflow_map('tune_grid',
               grid = rf_grid,
               seed = 123,
               verbose = TRUE,
               resamples = train_folds
               )

```

Perform cross validation on XGBoost model candidates (~ 8 min runtime)

```{r}

set.seed(123)
xgb_fit <- xgb_set %>%
  workflow_map(
    "tune_grid",
    resamples = train_folds,
    metrics   = metric_set(rmse),
    control   = control_grid(save_pred = TRUE, parallel_over = "resamples"),
    seed      = 123,
    verbose   = TRUE
  )

```

Extract performance metrics (rmse) from cross validation

```{r}

all_fits <- list(
  lm  = lm_fit,
```

```
  rf  = rf_fit,
  xgb = xgb_fit
)

performance <- map_dfr(
  names(all_fits),
  ~ {
    wf <- all_fits[[.x]]
    wf$wflow_id %>%
      map_dfr(~ extract_workflow_set_result(wf, .x) %>%
                collect_metrics() %>%
                filter(.metric == "rmse") %>%
                mutate(workflow = .x)
      )
  },
  .id = "fit_group"
) %>%
  arrange(mean)



# print combined rmse performance table for all 3 model types with hyperparameters
# model workflow has naming convention recipe_model (i.e. interact_rf is random forest with
interaction terms included)
print(performance, n = 10)

```

```{r}

pred_list <- list()

for (id in xgb_ids) {
  # best params for this recipe
  res_i  <- extract_workflow_set_result(xgb_fit, id)
  best_i <- tune::select_best(res_i, metric = "rmse")

  # finalize & fit on full training
  wf_i       <- extract_workflow(xgb_set, id)
  final_wf_i <- finalize_workflow(wf_i, best_i)
  fit_i      <- fit(final_wf_i, data = train_expanded)

  # optional: quick train RMSE
  tr_rmse <- predict(fit_i, train_expanded) %>%
    bind_cols(train_expanded) %>%
    rmse(truth = log_total, estimate = .pred)
  cat(sprintf("[Train RMSE | %s] %.5f\n", id, tr_rmse$.estimate))

  # predict on test; keep distinct column name
  pred_i <- predict(fit_i, test_expanded) %>%
    bind_cols(test_expanded) %>%
    select(id, .pred) %>%
    rename(!!paste0("pred_", id) := .pred)

  pred_list[[id]] <- pred_i
}

# merge predictions (for preview)
pred_merged <- Reduce(function(x, y) dplyr::left_join(x, y, by = "id"), pred_list)
head(pred_merged, 15)

```
```
```

```{r}

for (id in xgb_ids) {
  pred_i <- pred_list[[id]]

  submission_i <- pred_i
  names(submission_i)[2] <- "log_total"      # second column → log_total

  out_file <- sprintf("sub_%s_%s.csv", id, Sys.Date())
  write_csv(submission_i, out_file)
  cat(sprintf("\n[Wrote] %s\n", out_file))
  print(head(submission_i, 15))
}

# (optional) simple ensemble: mean of the two
# ensemble <- pred_merged %>%
#   mutate(log_total = rowMeans(dplyr::select(., starts_with("pred_")), na.rm = TRUE)) %>%
#   select(id, log_total)
# write_csv(ensemble, sprintf("sub_ensemble_mean_%s.csv", Sys.Date()))
```

Appendix: Team Member Contributions

Data Mining/Manipulation/Pre-Preprocessing: Alfred Mastan, Quinn Koch, Joseph Choi
Model Selection/Hyperparameter Tuning: Kwanhee Yoon, Johnathan Pham, Alfred Mastan
Report: Alfred Mastan, Quinn Koch, Joseph Choi, Johnathan Pham, Kwanhee Yoon