Statistics 101C - Kaggle Regression Project

Summer 2024

By: Albert Putranegoro, James Joyce, Benedetta Gasbarra, Mikayla Silverman,
Kyle Alexander Slaughter

Introduction:

The dataset at hand originates from a comprehensive survey conducted on approximately 5,000 Amazon customers between January 2018 and December 2022. The primary training file, train, aggregates data on customer and order counts across different states, months, and years. The key response variable in this dataset is log_total, which represents the base-10 logarithm of order_totals. The order_totals themselves are calculated by summing the costs of items from the amazon_order_details files.

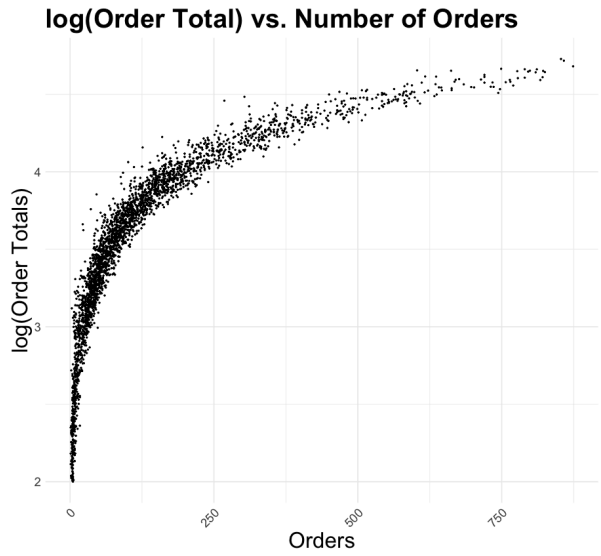
Customer purchasing behavior can exhibit trends and patterns related to temporal and geographical factors. Thus, it is reasonable to predict the total order value (log_total) based on the provided dataset. Historical studies on consumer spending have indicated that factors such as location and seasonal trends can significantly impact purchase behaviors. For instance, regional economic conditions and seasonal sales can lead to variations in order totals, making them important predictors of overall spending patterns.

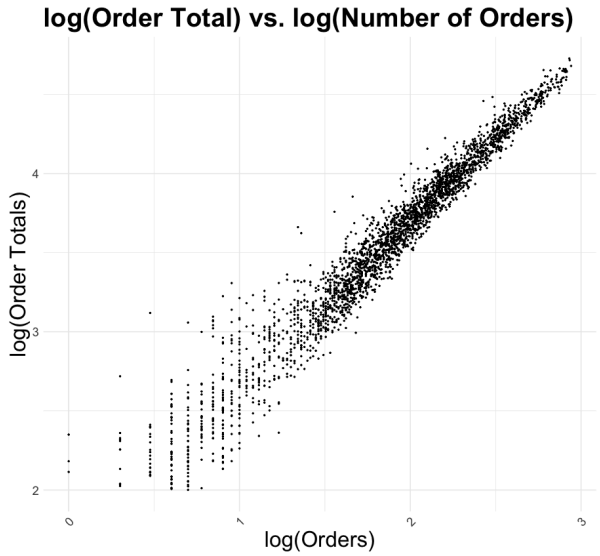
In the subsequent sections, we will delve deeper into the dataset to explore its distributions and interactions among variables. By applying regression models, we aim to uncover the relationships between customer and order characteristics and how they influence the log-transformed total order values.

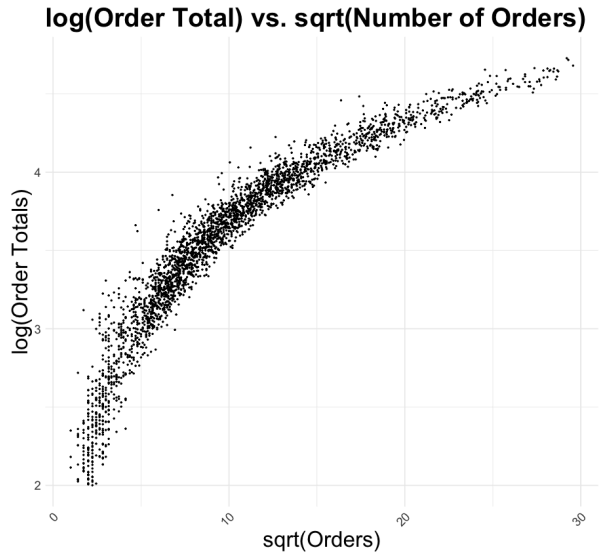
Exploratory analysis:

In this section, we conduct an exploratory analysis of the dataset to examine its key characteristics, uncover patterns, identify anomalies, and provide insights for subsequent analysis.

Graph 1-4:



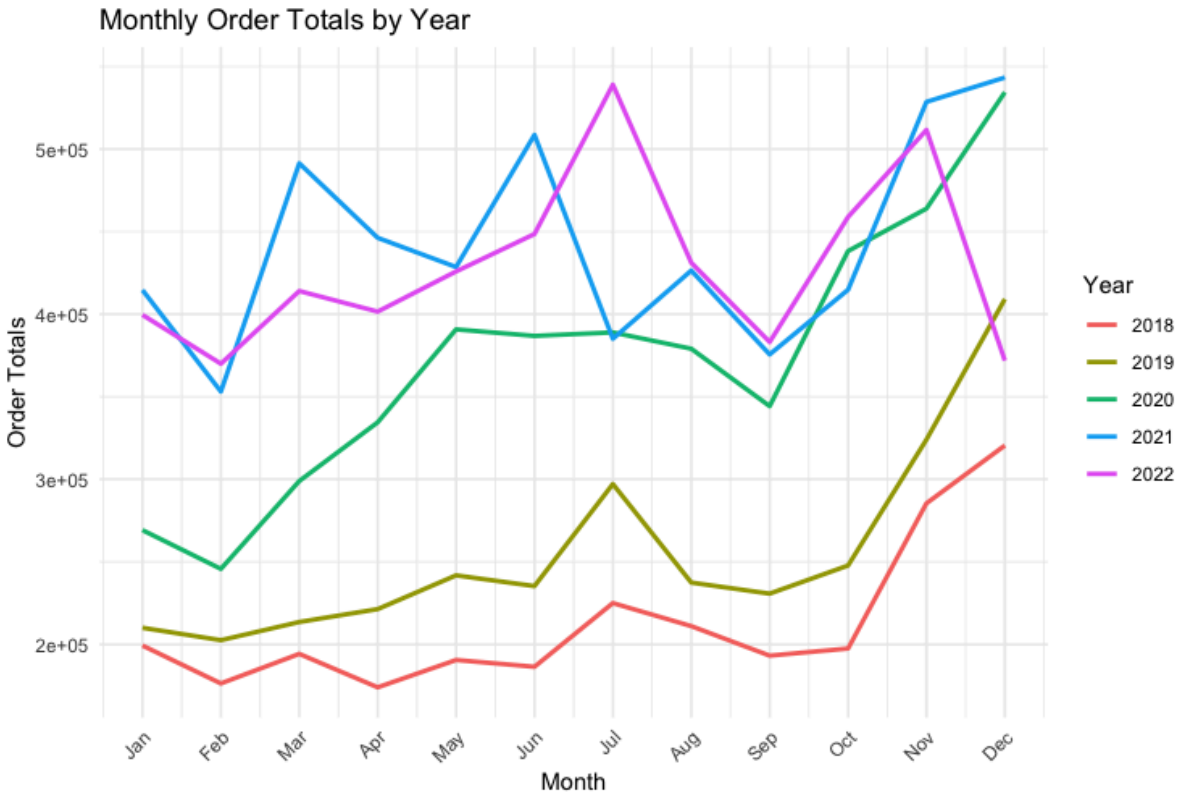




Description:

The above ***graphs 1 - 4*** illustrate the relationship between the number of orders and the cost of the orders. There appears to be a linear relationship between the order totals and the number of orders; for the project we will be predicting the log of the order total variable. Both variables put under log transformation lead to a more even and linear distribution of data. The square root transformation does not linearize the relationship.

Graph 5:

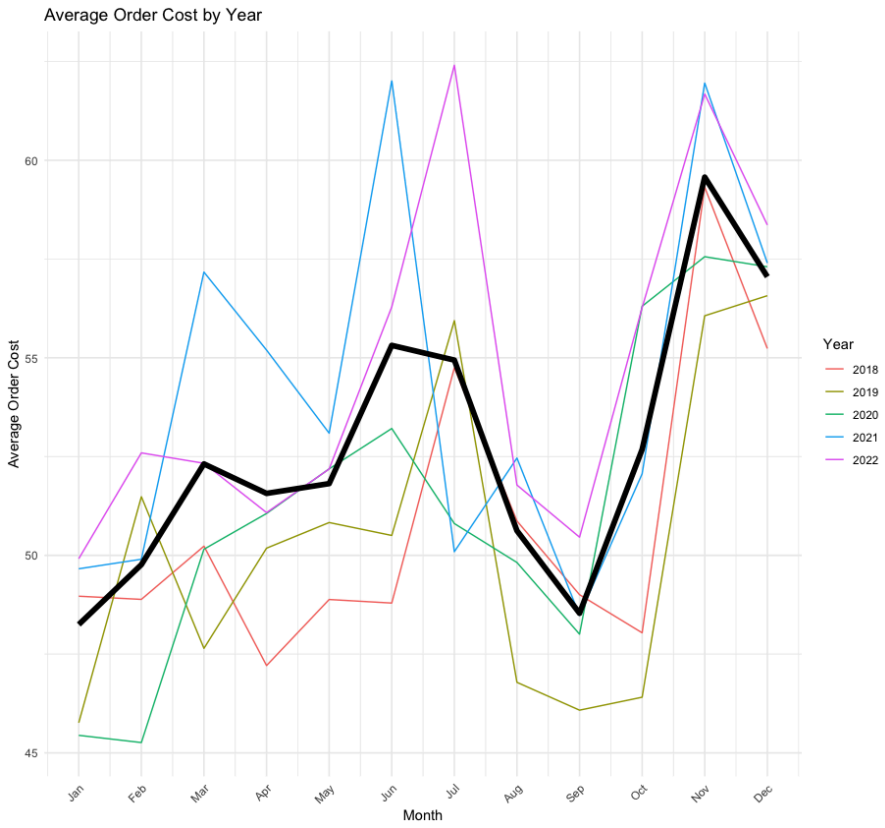


Description:

This line ***graph 5*** illustrates the monthly amount of money spent on Amazon orders across different years. Each year, there is a general increase in expenditure, with a notable surge around November and December, corresponding to holiday events such as Christmas, Hanukkah, Black Friday, or Cyber Monday, which drive higher shopping activity. In 2022, there is a noticeable drop in December, which may be attributed to incomplete data for that month. Additionally, there is an unexpected spike in orders from March to May 2020, aligning with the onset of the

COVID-19 pandemic and the subsequent shift towards increased online shopping due to stay-at-home restrictions. Furthermore, there are steeper increases in purchasing in July 2018, July 2019, October 2020, June 2021, and July 2022, which correspond to that year's Amazon Prime Day, an annual event that puts deals on many Amazon products.

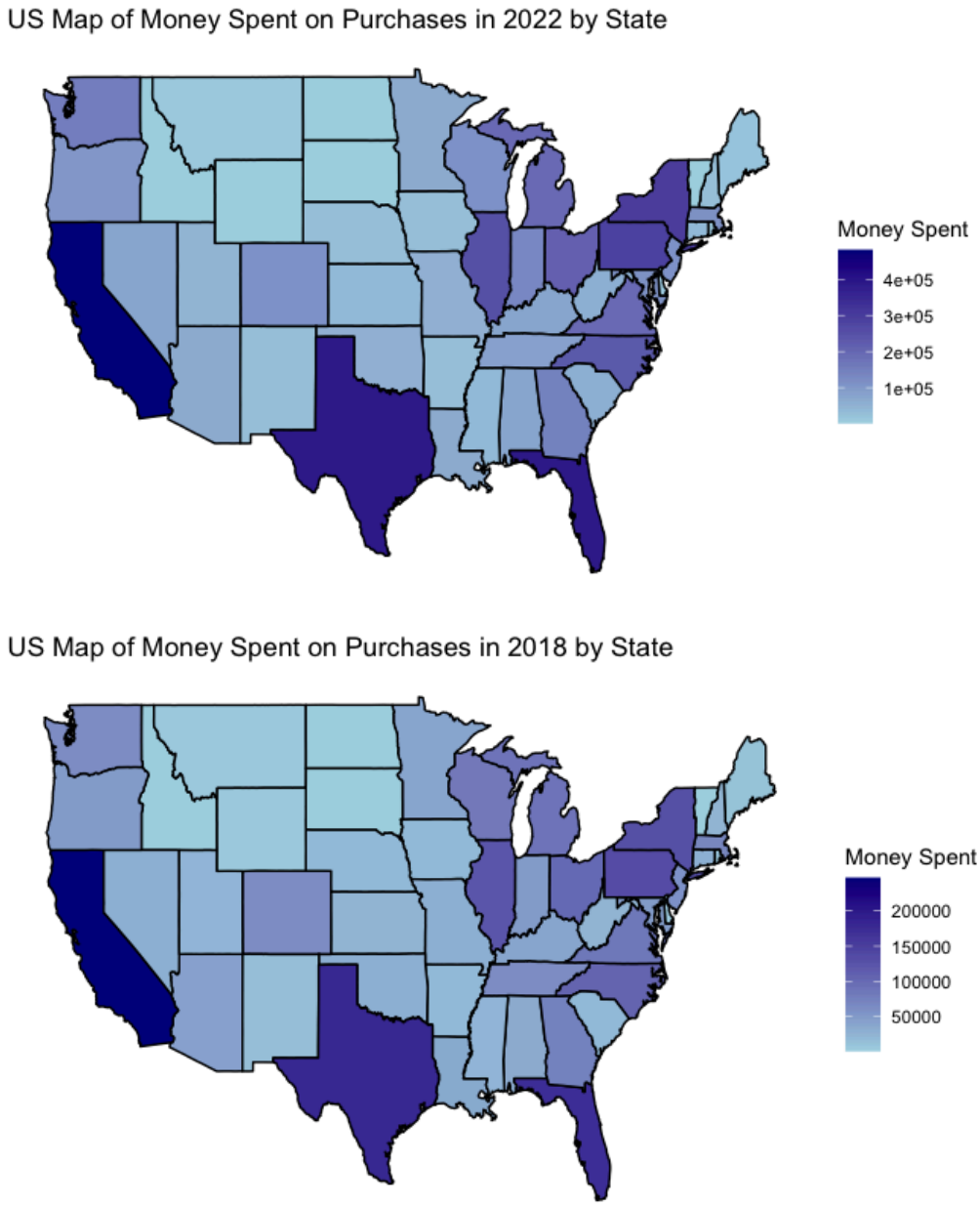
Graph 6:



Description:

In this line ***graph 6***, the average order cost is found by dividing the total amount spent on orders by the total number of orders. The thick black line is the average order cost over the five years. The cost per order appears to be more consistent over the five years, as shown by more overlapping lines than the total amount of money spent, as depicted in the previous graph. The more expensive purchases follow the same patterns with holidays and purchasing events such as Black Friday and Amazon Prime Day as described above.

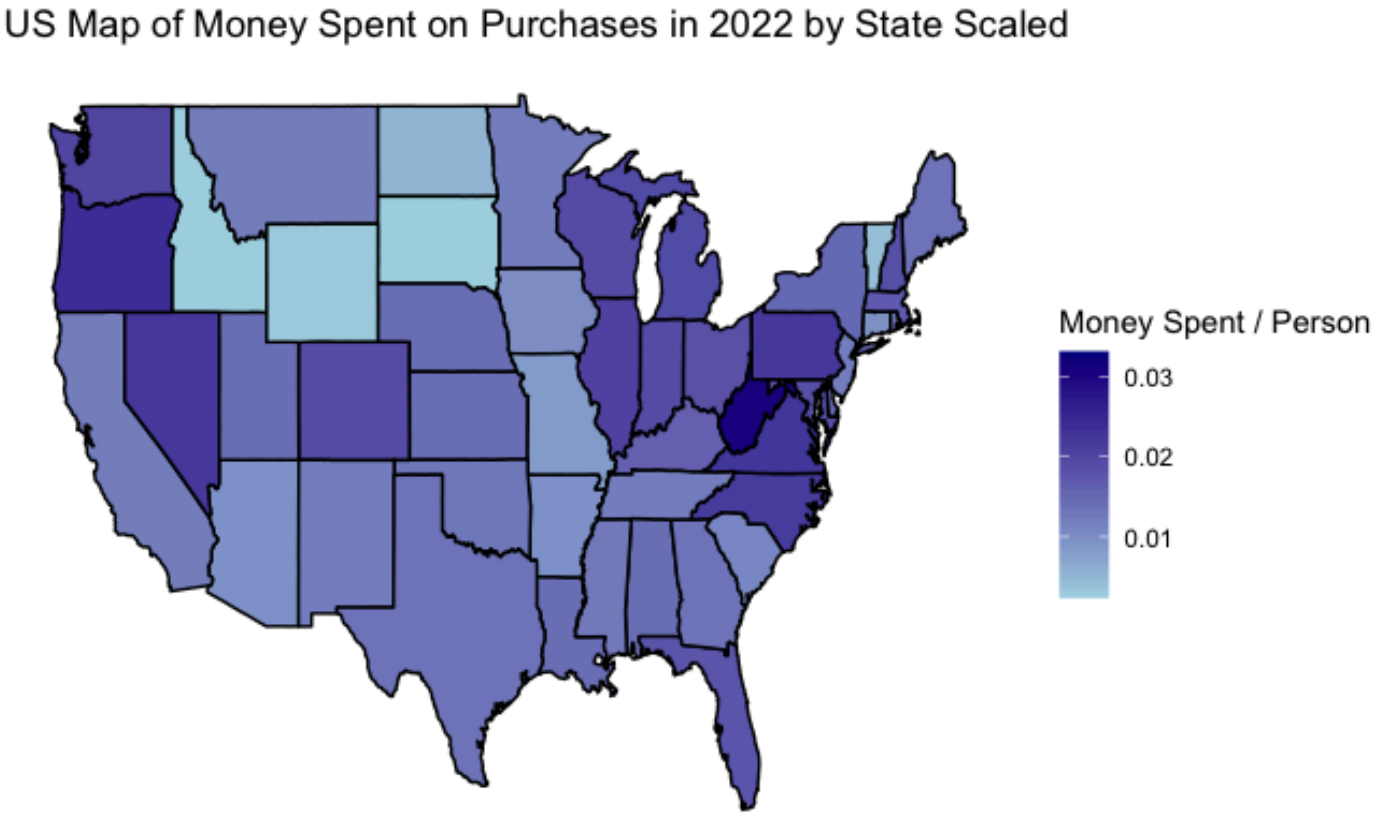
Graph 7-8:



Description:

The above ***graphs 7-8*** illustrate the amount of money spent across the continental United States in 2018 and 2022, highlighting which states have the highest expenditure. The spending patterns show a consistent distribution, with California, Texas, and Florida emerging as the top spenders, aligning with their large populations. Notably, the total expenditure across these states increased by a factor of 200 over the four-year period.

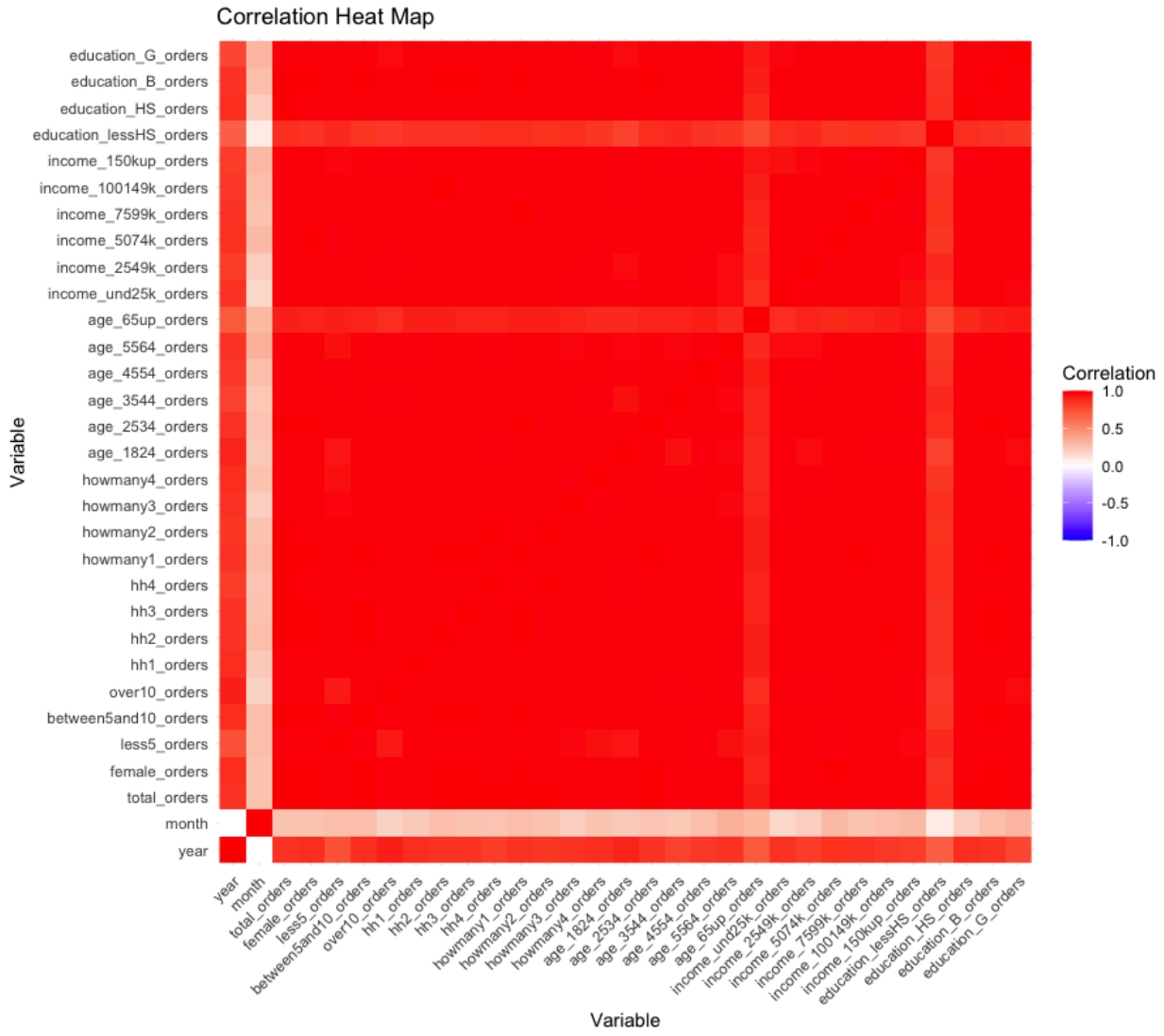
Graph 9:



Description:

Graph 9 illustrates the amount of money spent per person across different states in 2022. The expenditure per capita is calculated by dividing the total spending in each state by its population, sourced from the US Census Bureau. This graph provides a detailed look at spending behavior on a per-person basis, revealing significant differences between states. While California, Texas, and Florida lead in total expenditure due to their large populations, the per capita data shows a different picture. States like West Virginia emerge with the highest per capita expenditure, suggesting that, on average, individuals in these states spend more on Amazon orders compared to those in larger states.

Graph 10:

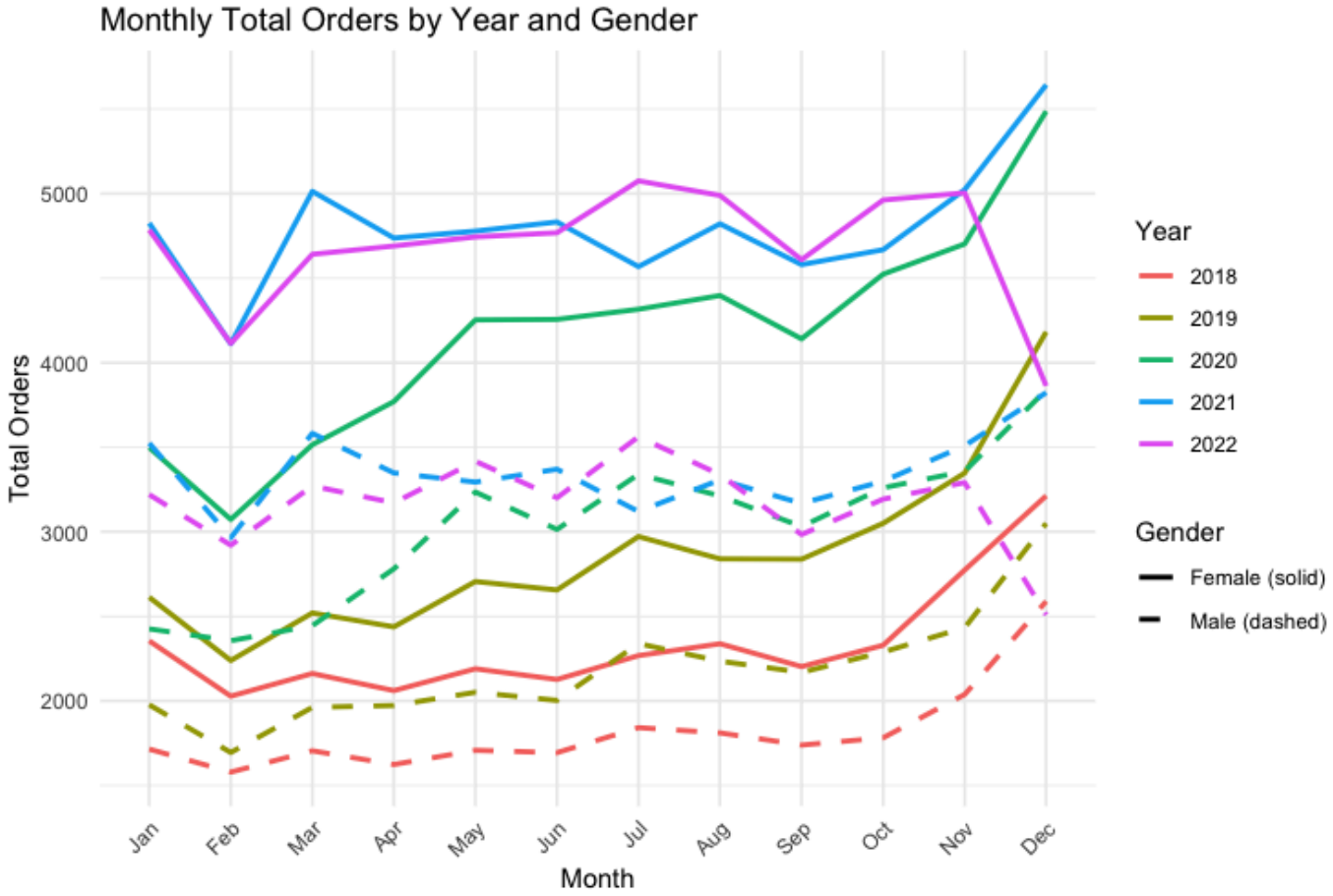


Description:

Graph 10 presents a correlation heat map illustrating the relationships between different variables in the dataset. The heat map uses color gradients to indicate the strength and direction of correlations. The key variables include total order value (log_total), number of orders, and various demographic factors such as age, income, and education level.

Most variables show high correlations with each other. However, the month variable exhibits a relatively low correlation with other variables, suggesting that monthly changes have a minimal direct impact on the overall dataset. Another pair of variables with lower correlation are those with less than a high school education and those making orders over 65.

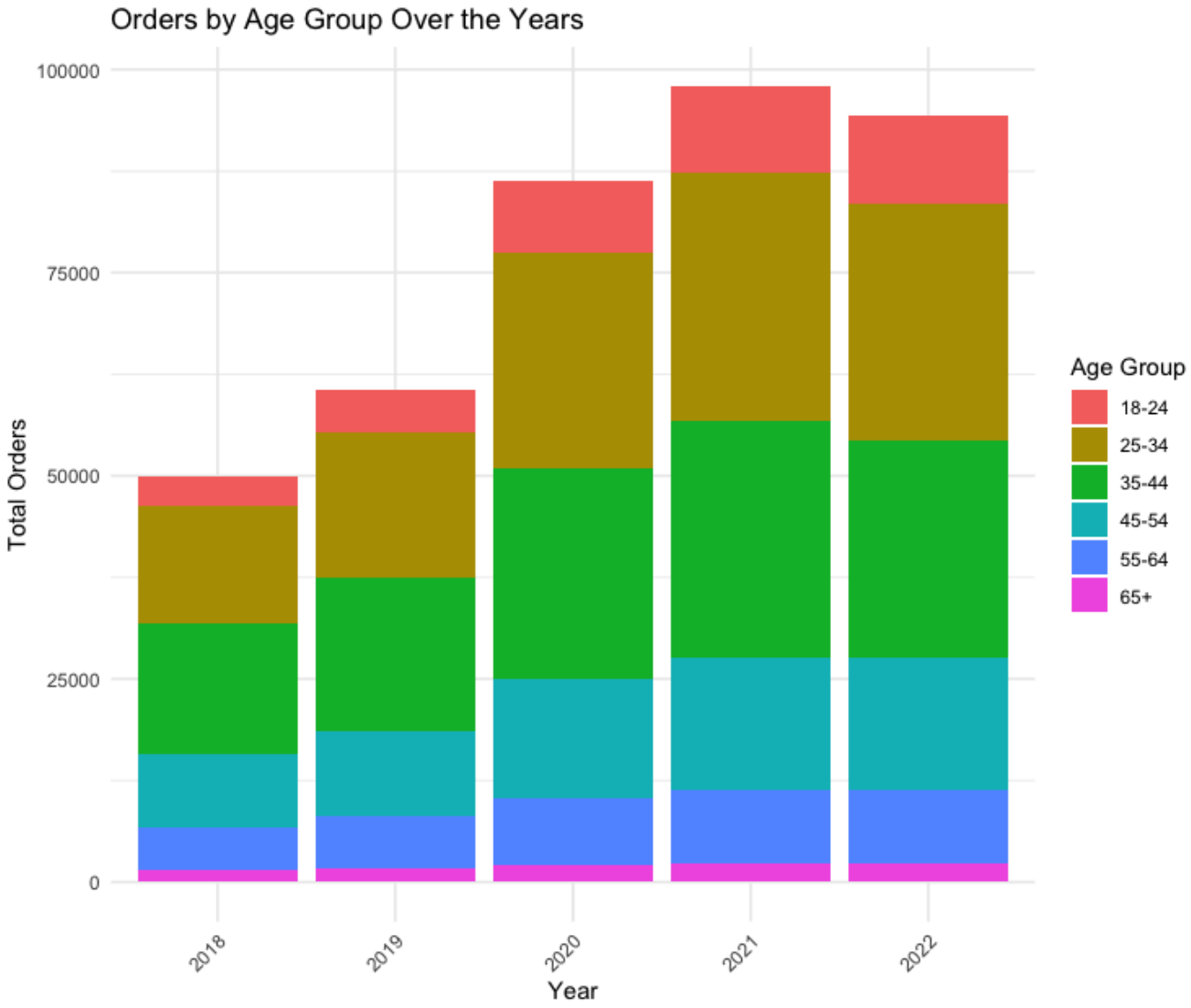
Graph 11:



Description:

Graph 11 displays a line graph representing the monthly order data segmented by year and gender. The solid lines denote female respondents, while the dashed lines represent male respondents. The graph reveals that consistently over the five-year period, female consumers placed more orders each month compared to their male counterparts. This trend is evident across all years, with females showing a higher volume of orders in almost every month.

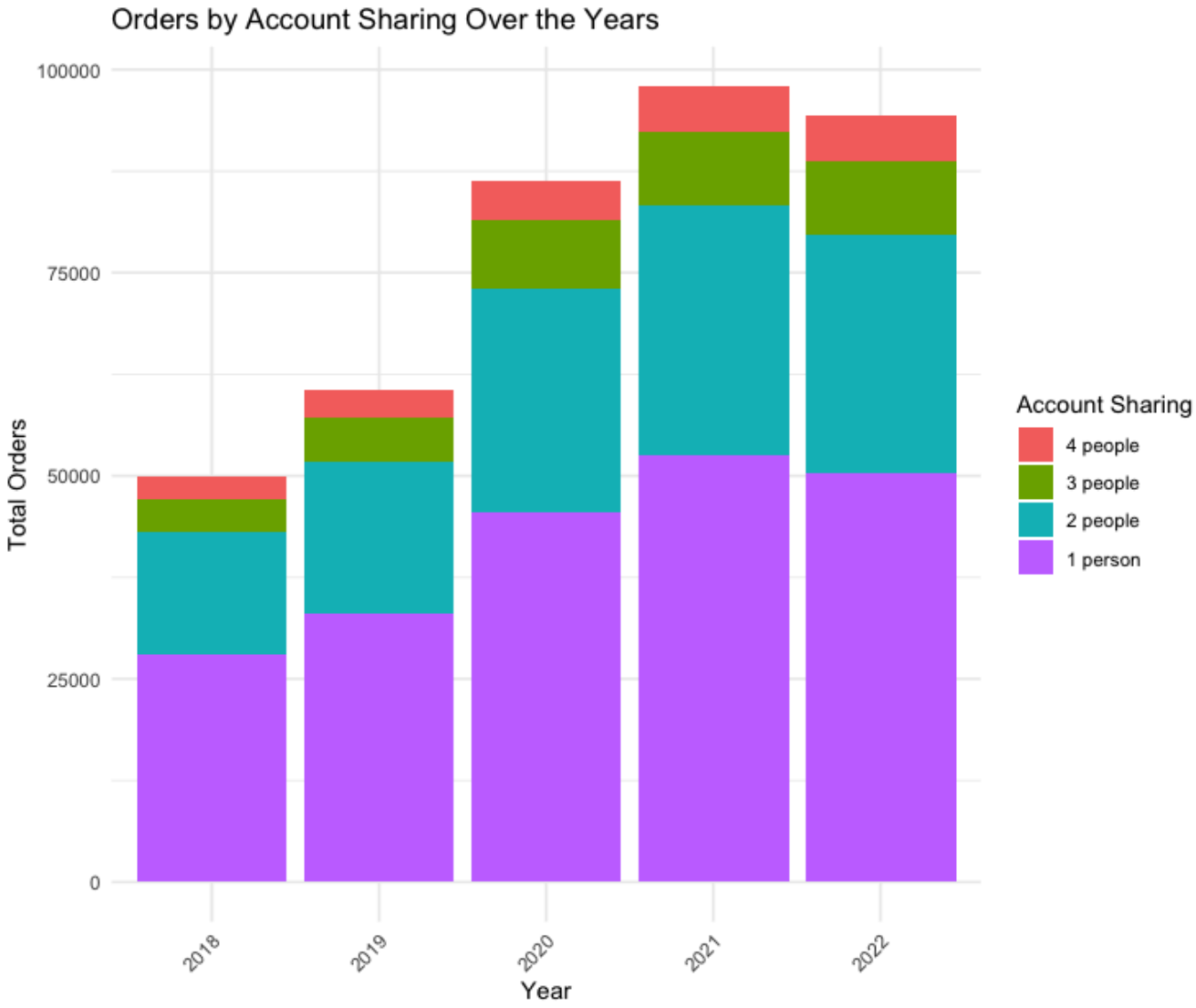
Graph 12:



Description:

In ***graph 12,*** a bar plot shows the trend of purchases based on age. The total number of orders increased each year, with significant growth observed in the 25-34 and 35-44 age groups. In contrast, the number of orders for the 55-64 and 65+ age groups showed only modest increases.

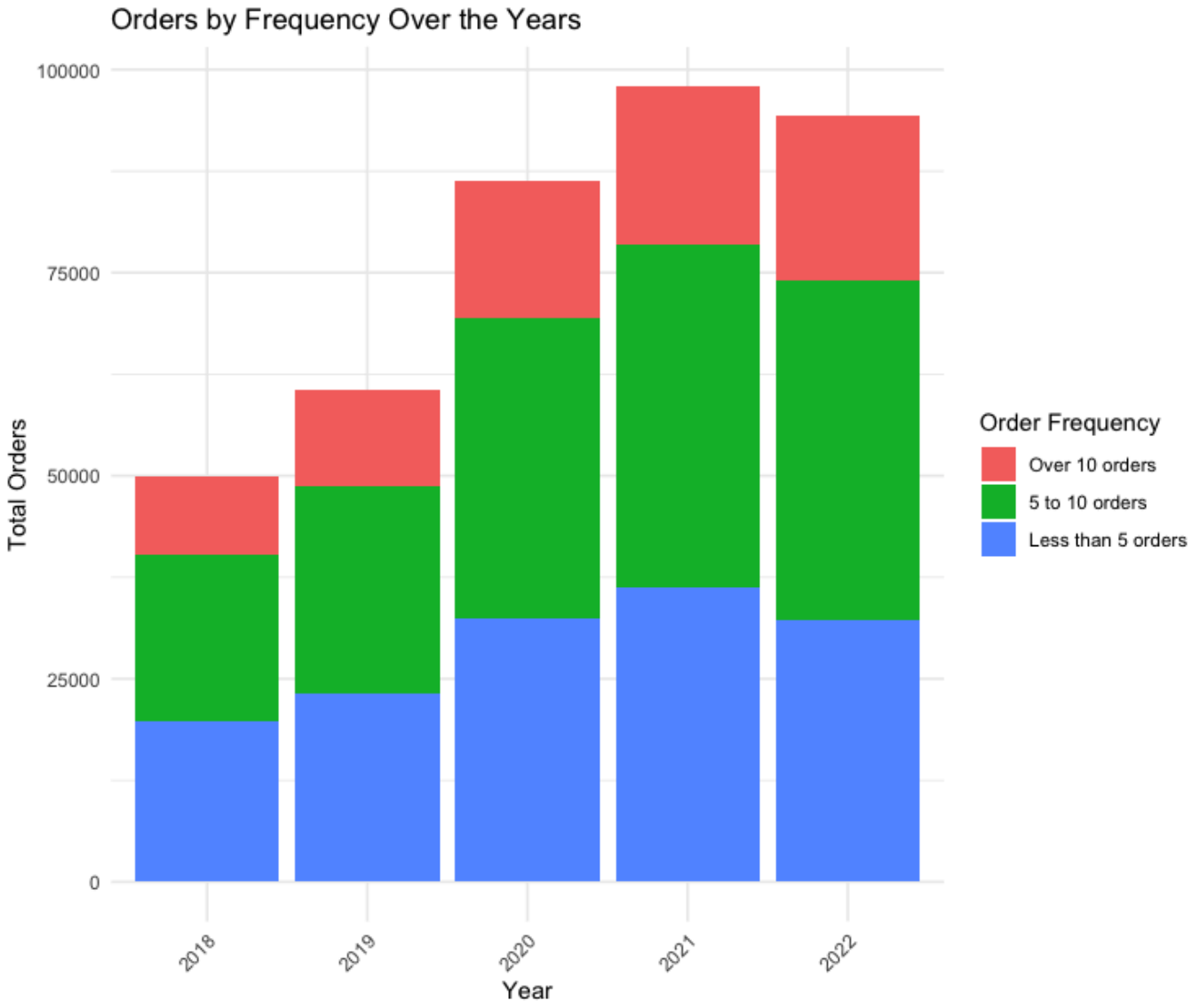
Graph 13:



Description:

Graph 13 is a bar plot that illustrates the trend of orders based on account-sharing status. Over the observed period, the majority of orders were placed from accounts that were not shared. This pattern remained consistent each year, indicating that most customers prefer using their own accounts for purchases rather than shared accounts.

Graph 14:



Description:

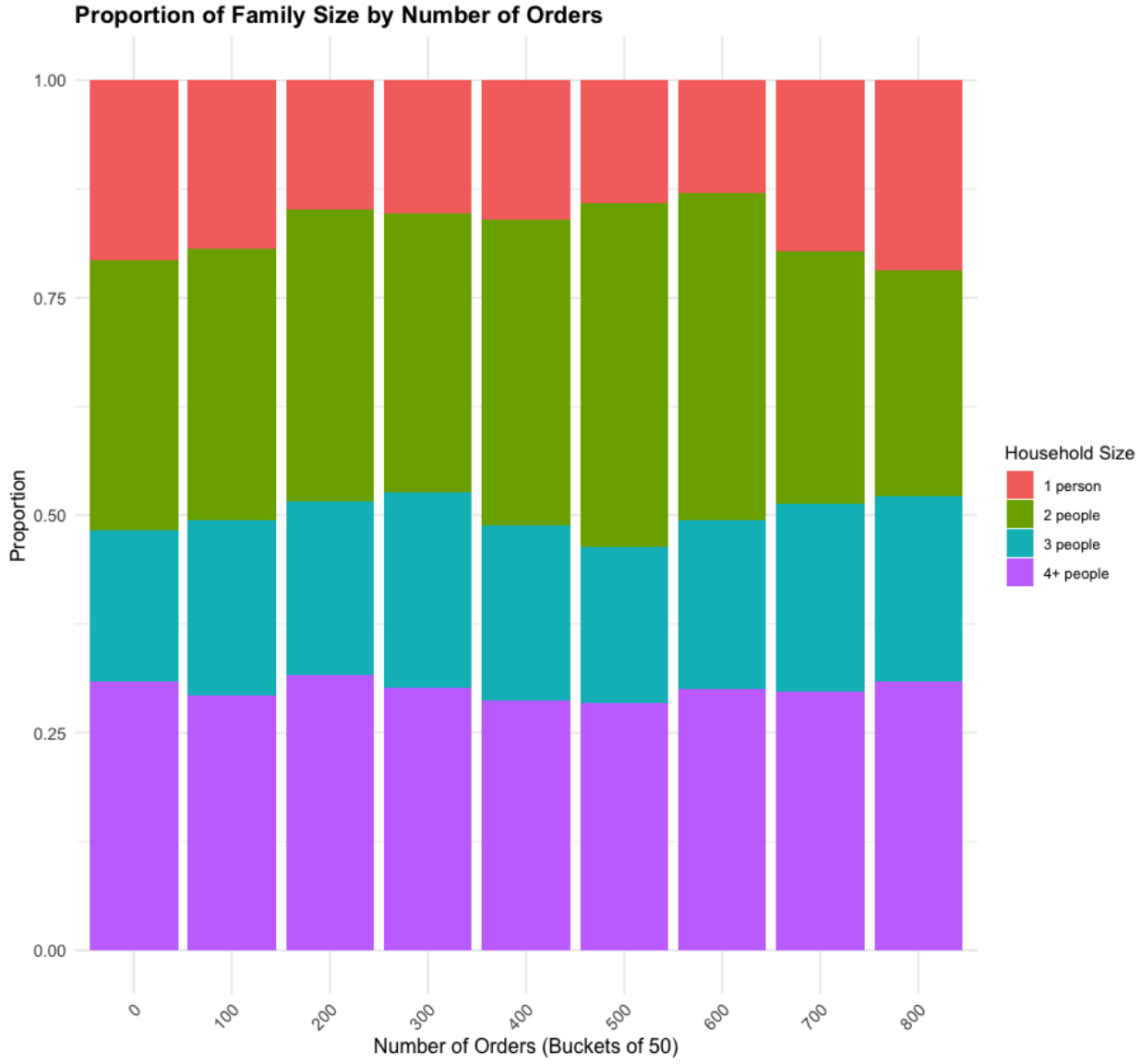
In the above ***graph 14*** a bar plot depicts the trend in order frequency over the five-year period. The data shows that the total number of orders increased each year, with most customers placing between 5 to 10 orders annually. The stable frequency range of 5 to 10 orders per year may indicate a strong baseline of regular customers.

Graph 15:

Description:

In ***graph 15***, a stacked bar graph displays the proportions of orders for each income level over the five years. The income brackets are under $25k, $25-49k, $50-74k, $75-99k, $100-149k, and over $150k. The distribution of purchases across these income levels remained relatively stable over the years, with middle-income levels ($50-74k) consistently having the highest proportion of purchases.

Graph 16:



Description:

In ***graph 16***, a bar plot illustrates the trend in orders based on household size. The majority of orders came from households with two people and those with four or more people. The number of orders from these households increased each year. The dominance of orders from two-person and larger households may suggest that Amazon's services are particularly appealing to small and medium-sized households. This might be due to the convenience and bulk purchasing benefits that Amazon offers.

Preprocessing:

From the original dataset, we removed the order_totals variable because it would enable perfect predictions of log_total, leading to perfect model performance. By excluding this variable during data processing, we ensured that our model could make accurate predictions for the unseen data that did not include order_totals. This approach prevented us from making predictions based on information that directly reveals the target variable.

```
82  # Load the training
83  train <- read_csv("train.csv")
84  
85  # Inspect the data
86  glimpse(train)
87  
88  # Remove order_totals from train data
89  train <- train %>%
90    select(-order_totals)
91
```

Candidate Models, Models Evaluations and Tuning :

We agreed to use 10-fold cross-validation on the training dataset, with a seed of 146 to ensure consistency. Initially, we divided the work and selected five different candidate models to evaluate their performance, intending to narrow down the options based on their results. The five models we chose include Boosting Trees, Neural Networks, Random Forests, Support Vector Machines, and Linear Regression. Each group member conducted their own model analysis, and we then shared our findings and compared metrics such as MSE and RMSE. Based on these comparisons, we selected the two best-performing models for this specific dataset. Below is a detailed analysis of the models we chose.

```r
# Set seed for reproducibility
set.seed(146)

# Create cross-validation folds
cv_folds <- vfold_cv(train, v = 10, strata = log_total)
```

Linear Regression:

The initial ***linear regression model*** was set engine to lm.

```r
set.seed(146)

#define linear regression model
lm_model <-
  linear_reg() %>%
  set_engine("lm")

#fit model to data
lm_form_fit <-
  lm_model %>%
  fit(log_total ~ ., data = train)
```

The initial linear model exhibited an RMSE of 0.1441 and an SE = 0.0027 but underperformed on Kaggle, achieving a score of 0.22617, which suggested overfitting. By removing highly correlated variables, the Kaggle score improved to 0.06986, though the RMSE increased to 0.1519 and the was SE = 0.0028. Despite this improvement, the results did not match the success of other models such as Random Forest and Gradient Boosting models, leading us to concentrate our efforts on those models.

```r
# Calculate the correlation matrix
cor_matrix <- cor(numeric_train, use = "pairwise.complete.obs")
#print(cor_matrix)

high_cor_pairs <- findCorrelation(cor_matrix, cutoff = 0.8, names =
TRUE)
print(high_cor_pairs)

# Remove highly correlated variables from the data
reduced_train <- train %>%
  select(-one_of(high_cor_pairs))

# Fit the linear regression model with the reduced data
lm_form_fit_reduced <- lm_model %>%
  fit(log_total ~ ., data = reduced_train)
```

Support Vector Regression

The second model chosen was the ***Support Vector Regression (SVR) model.*** To optimize the SVR model, we conducted hyperparameter tuning with a cost and RBF_sigma , and the engine was set to Kernlab.

```r
# Define the SVR model with tuning parameters
svr_model <- svm_rbf(
  mode = "regression",
  cost = tune(),
  rbf_sigma = tune()
) %>%
  set_engine("kernlab")
```

The hyperparameters were set to coast range on a log2 scale from -2 to 10, and an RBF sigma range on a log2 scale from -10 to 0, and 5 levels were applied.

```r
# Define a refined grid of hyperparameters to tune
svr_params <- parameters(svr_model) %>%
  update(
    cost = cost(range = c(-2, 10), trans = scales::log2_trans()),
    rbf_sigma = rbf_sigma(range = c(-10, 0), trans = scales::log2_trans())
  )

# Create a refined grid of values to try
svr_grid <- grid_regular(svr_params, levels = 5)
```

The model was subsequently tuned to select the optimal hyperparameters for achieving the lowest RMSE, and its performance was assessed using the root mean squared error (RMSE) metric.

```
# Tune the hyperparameters using cross-validation
svr_tune <- svr_workflow %>%
  tune_grid(
    resamples = train_folds,
    grid = svr_grid,
    metrics = metric_set(rmse),
    control = control_grid(save_pred = TRUE)
  )

# Select the best hyperparameters
best_svr <- svr_tune %>%
  select_best(metric = "rmse")

# Finalize the workflow with the best hyperparameters
final_svr_workflow <- svr_workflow %>%
  finalize_workflow(best_svr)
```

After applying the select_best function the optimal hyperparameters were cost equals 16 and rbf_sigma equals 0.03125, and the final model achieved a Kaggle score of 0.03819. The RMSE for this model was 0.1654604, with a standard error of 0.01071127. Despite these results, the SVR model did not outperform the Random Forest and Gradient Boosting models. Consequently, further testing and optimization of the SVR model were not pursued, and we focused our efforts on enhancing the other models.

Neural Network

Another model we chose to run was a Long Short-Term Memory(LSTM) Neural Network model. We chose this model as we wanted to include a neural network model and LTSM neural networks are better suited for capturing trends within time series data. This was performed in Python due to familiarity with Tensorflow for creating neural networks. After creating the initial model with 2 hidden layers, each with 32 units. For each layer, a dropout rate of .1 was implemented to prevent overfitting at each layer and a learning rate of .001. Lastly, I set the epoch and batch size to 30 and 64 respectively to make the tuning process more computationally efficient while also ensuring that each update is more accurate and less affected by outliers. Then, I performed a grid search with 2 possible parameters of 64 and 128 for the unit sizes, .01 and .001 for the learning rate, and .1 and .2 for the dropout rate. These are typical values to fit a LTSM model to and

were the reason these values were chosen. Due to the computationally expensive nature of tuning a neural network and the processing power of my laptop, I kept each one to two parameters to ensure I got results back in a reasonable timeframe.

```
param_grid = {
    'model__units_1': [64,128],
    'model__dropout_1': [0.1,.2],
    'model__units_2': [64,128],
    'model__dropout_2': [0.1,.2],
    'model__learning_rate': [.01,0.001]
}
```

After running the LTSM neural network model based on these parameters, the ones chosen for the model were_. The model's test statistics were along the average at with an RMSE of .129.
However, the MSE we submitted on Kaggle was significantly higher than the other models we tested. One reason could be that the sample size of the training data isn't large enough to capture more complex patterns in the model. Additionally, due to the computationally expensive nature of tuning a LTSM neural network, we could not try many different parameters with the grid search. This resulted in fewer tested combinations of parameters, which could lead to inaccurate fitting. Lastly, we had other models with lower RMSE and stronger results on Kaggle, showing it generalized better to unseen data. This could be due to dataset wasn't large enough to capture complex trends in the time-series data

Gradient Boosted Decision Trees

The second to last model chosen was the Gradient Boosted Decision Tree model. After creating the initial model object with the following parameters, a grid search was conducted on the three most influential hyper-parameters: the number of trees to be used (trees), the maximum depth of each tree (tree_depth), and the step size for each iteration (learn_rate). Eventually, the hyper-parameters were set as follows: trees = 150, tree_depth = 1, learn_rate = 0.1, and the engine was set to xgboost.

```
grad_boost <-
  boost_tree(
    trees = 150,
    tree_depth = 1,
    learn_rate = 0.1
  ) %>%
  set_engine("xgboost") %>%
  set_mode("regression")
```

```
# Create a workflow
# Add a formula to predict log_total from all other remaining variables
grad_boost_workflow <-
  workflow() %>%
  add_formula(log_total ~ .) %>%
  add_model(grad_boost)

# Fit Model on training data set:
grad_boost_fit <- grad_boost_workflow %>% fit(data = train)
```

After the first round of hyper-parameter tuning, it became clear that the MSE didn't significantly decrease as tree_depth increased. This was substantiated even more during subsequent tuning when the range of values for tree_depth was restricted even lower. It eventually became clear that using tree_depth of one seemed optimal as this would significantly reduce model complexity and be less prone to overfitting while at the same time yielding roughly the same MSE that higher tree depths would provide.

```
recipes_param_1 <- extract_parameter_set_dials(grad_boost)

# We can change this range of values to be tried
recipes_param_1 <-
  recipes_param_1 %>%
  update(trees = trees(range = c(1, 2000))) %>%
  update(tree_depth = tree_depth(range = c(1, 15))) %>%
  update(learn_rate = learn_rate(range = c(-3, -0.5), trans = scales::log10_trans()))

# We use a regular grid search to optimize parameters
# 3 levels with 3 parameters means 3^3 = 27 combinations
parm_grid_1 <- grid_regular(recipes_param_1, levels = 3)
```

```
# Now we re-sample using grid search; take very long time!
grad_boost_reg_tune_1 <-
  grad_boost_workflow %>%
  tune_grid(
    resamples = train_folds,
    grid = parm_grid_1
    )

# We find the best hyper-parameters
iteration1 <- show_best(grad_boost_reg_tune_1, metric = "rmse")
iteration1
```

```
## # A tibble: 5 x 9
##   trees tree_depth learn_rate .metric .estimator  mean     n std_err .config
##   <int>      <int>      <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1  2000          1     0.0178 rmse    standard   0.114    10 0.00304 Preprocess~
## 2  1000          1     0.0178 rmse    standard   0.115    10 0.00282 Preprocess~
## 3  2000          1     0.316  rmse    standard   0.116    10 0.00329 Preprocess~
## 4  1000          1     0.316  rmse    standard   0.116    10 0.00320 Preprocess~
## 5  1000          8     0.0178 rmse    standard   0.116    10 0.00283 Preprocess~
```

```
metrics_1 <- collect_metrics(grad_boost_reg_tune_1)

metrics_summary_1 <- metrics_1 %>%
  filter(.metric == "rmse") %>%
  group_by(tree_depth) %>%
  summarize(mean_rmse = mean(mean), .groups = 'drop')
metrics_summary_1
```

```
## # A tibble: 3 x 2
##   tree_depth mean_rmse
##        <int>     <dbl>
## 1          1      1.16
## 2          8      1.15
## 3         15      1.15
```

Regarding the other two parameters, trees and learn_rate, the range of values for these two was eventually restricted. However, it wasn't until the model could be tested on the test data set a couple of times that the exact hyperparameters could finally be narrowed down. Specifically, trees were eventually set to 150 while learn_rate was set to 0.1. Afterward, several other less important hyperparameters were tuned, including min_n, loss_reduction, and mtry. However, none of these saw any improvement in the test MSE.

Overall, the model performed fairly well, with an MSE on the test set of 0.01773, but likely had problems with underfitting the data.

Random forest

The ***Random Forest model*** was selected for its potential to handle complex datasets and produce accurate predictions. Initially to improve the performance of this model, it was defined with tuning parameters for mtry (number of variables considered at each split), trees (number of trees in the forest), and min_n (minimum number of data points in a node), and the engine used was ranger.

```
# Define the base random forest model with tuning parameters
rf_model <- rand_forest(
  mtry = tune(),  # Number of variables to possibly split at each node
  trees = tune(),  # Number of trees
  min_n = tune()  # Minimum number of data points in a node
) %>%
  set_mode("regression") %>%
  set_engine("ranger")
```

A workflow was created, combining the model and the formula for prediction. A grid of hyperparameters was defined for tuning, with mtry ranging from 2 to 5, trees ranging from 50 to 100, and min_n ranging from 5 to 15, and 3 levels. This grid allowed a systematic exploration of the parameter space.

```
# Define the workflow
rf_workflow <- workflow() %>%
  add_model(rf_model) %>%
  add_formula(log_total ~ .)

# Define a grid for hyperparameter tuning
rf_grid <- grid_regular(
  mtry(range = c(2, 5)),
  trees(range = c(50, 100)),
  min_n(range = c(5, 15)),
  levels = 3)
```

The model was then tuned using the defined grid and cross-validation folds. The tuning results were collected, and the best model was selected based on the RMSE metric. The final workflow was then finalized with the best hyperparameters: mtry = 5, trees = 100, and min_n = 10. The Random Forest model achieved an RMSE of 0.1765 and SE of 0.0076 on the training data.

```r
# Tune the model
rf_tune_results <- tune_grid(
  rf_workflow,
  resamples = cv_folds,
  grid = rf_grid,
  control = control_grid(save_pred = TRUE))

# Collect the tuning metrics
rf_metrics <- rf_tune_results %>%
  collect_metrics()

# Print the metrics to inspect the results
print(rf_metrics)

# Select the best model based on RMSE
best_rf <- rf_tune_results %>%
  select_best(metric = "rmse")

```

This performance, combined with a Kaggle score of 0.01765, placed our model in the top 5 of the Kaggle competition. Due to its strong performance, we focused our efforts on further improving this model.

After further consideration and analysis, we decided to modify some parameters for the grid, which includes various values for three hyperparameters: mtry (the number of variables randomly sampled as candidates at each split), splitrule (the criterion for splitting nodes, either "variance" or "extratrees"), and min.node.size (the minimum size of terminal nodes). By specifying a range of values for each parameter, the script sets up a comprehensive grid search to identify the best combination of parameters that yields the highest model performance during cross-validation.

```r
#Tuning Parameter Grid
param_grid <- expand.grid(
  mtry = seq(2, 10, by = 2),
  splitrule = c("variance", "extratrees"),
  min.node.size = c(1, 5, 10)
)
```

To ensure robust model evaluation, the script sets up a cross-validation control using the trainControl function from the caret package. It specifies 5-fold cross-validation (method = "cv", number = 5), which means the training data will

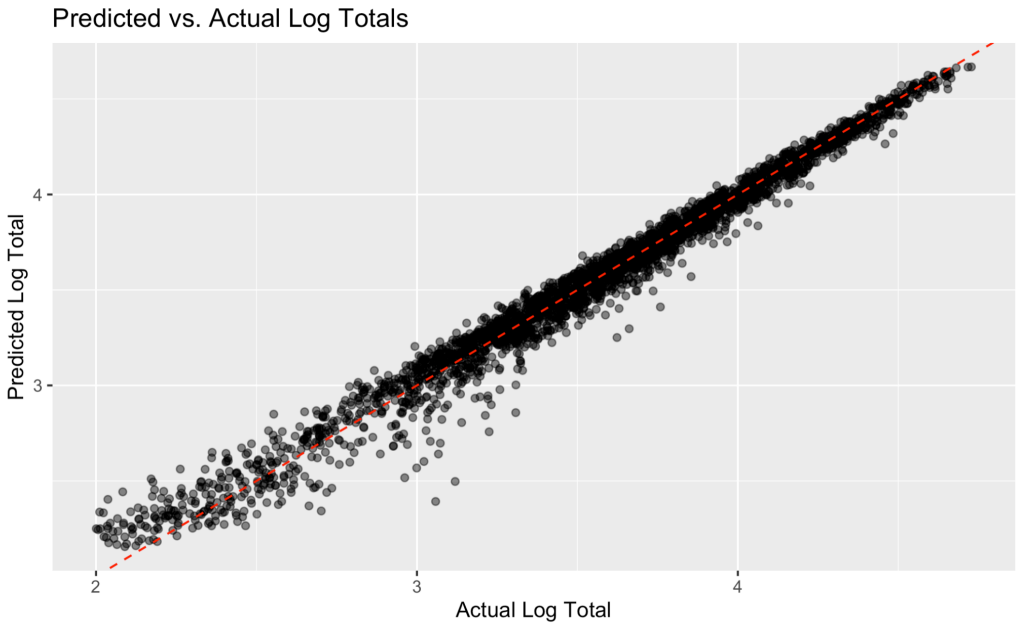
be split into five subsets. The model will be trained and validated five times, each time using a different subset for validation and the remaining subsets for training.

```
#Cross-Fold Validation
ctrl <- trainControl(
  method = "cv",
  number = 5,
  verboseIter = TRUE,
)
```

To further ensure better results for the Random Forest model, we used the train function from the caret package. Here, we specified method = "ranger" to use the ranger implementation of Random Forest. The trControl = ctrl argument applies the cross-validation settings defined earlier. The tuneGrid = param_grid argument uses the predefined parameter grid to tune the model. The importance = 'permutation' option indicates that variable importance will be assessed using permutation importance, and num.trees = 100. This comprehensive setup aims to find the best-performing model through rigorous training and validation. After tuning the model the best and final values used for the model were mtry = 10, splitrule = extratrees, and min.node.size = 10.

```
#Tuning Random Forest based on Parameter Grid
rf_tuned <- train(
  log_total ~ .,
  data = train,
  method = "ranger",
  trControl = ctrl,
  tuneGrid = param_grid,
  importance = 'permutation',
  num.trees = 100
)
```

To visualize the relationship between the actual and predicted log_total values in the training set, a scatter plot was created using ggplot2. In this plot, each point represents a training instance, with actual values on the x-axis and predicted values on the y-axis. This visualization aids in assessing the model's performance; accurate predictions are indicated by points that lie close to the reference line.



With all these changes, we were able to closely predict the unseen data and we achieved an RMSE of .0813 and an SE of .0015, securing 5th place in the Kaggle competition with a score of 0.01733 on the public leaderboard.

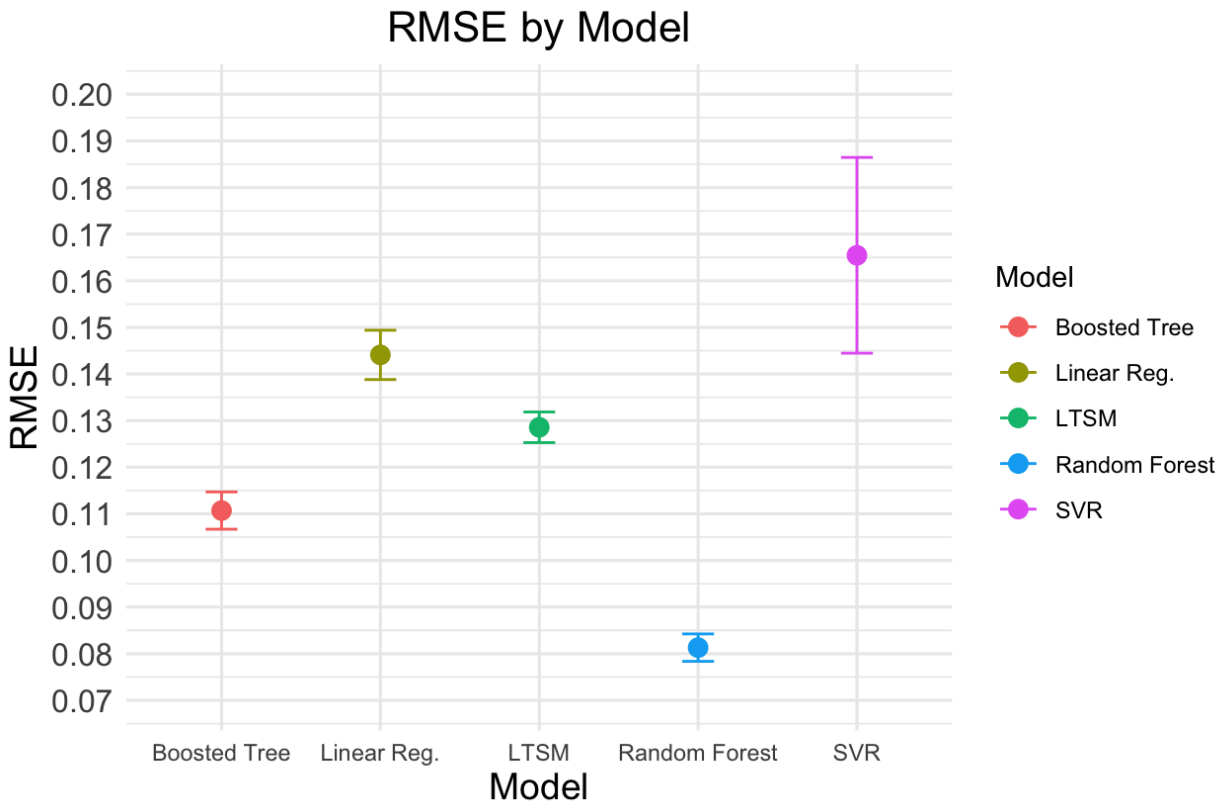
In conclusion, we evaluated five different models for this project: Boosting Trees, Neural Networks, Random Forests, Support Vector Machines, and Linear Regression. Among these models, the Random Forest demonstrated the best performance in terms of RMSE. Therefore, we selected the Random Forest as our final model for this project.

Table of finalized models:

Model identifier	***Type of Model***	***Engine***	***Recipe Used/Listed Variables***	***Hyperparameters***
decision_tree()	Linear Regression	“rpart”	34 variables used, Removed “order totals”	cost_complexity(range =c(0.001,0.1)), min_n(range= c(2, 40)), levels = 10
svm_rbf()	Support Vector Regression	“kernlab”	34 variables used, Removed “order totals”	Cost = 16 RBF Sigma = 0.03125
Tensorflow LTSM Model	Neural Network	“Keras”	34 variables used, Removed “order totals”	'model__dropout_1': 0.1, 'model__dropout_2': 0.1, 'model__learning_rate': 0.001, 'model__units_1': 128, 'model__units_2': 128
boost_tree()	Gradient Boosted Decision Trees	"xgboost"	34 variables used; Removed “order totals”	trees = 150, tree_depth = 1, learn_rate = 0.1
rf_model()	Random Forest	“Ranger”, “Caret”	34 variables used; Removed “order totals”	Mtry = 10, Splitrule = “extratrees”, Min.node.size = 10, Importance = “impurity”

Table of finalized metrics:

Model identifier	*Metric Score: RMSE*	*Metric Score: SE*
Linear	0.1518753	0.002800526
svm_rbf()	0.1654604	0.01071127
neural	0.1198763	0.001563042
boost_tree()	0.1107016	0.002040951
rf_model ()	0.0813287335691195	0.00149941784578548



Description:

The autopilot plot illustrates the Root Mean Square Error (RMSE) performance of the tested models. The plot clearly shows that the Random Forest model achieved the lowest RMSE, indicating good predictive accuracy compared to the other models. The standard error bars provide a visual representation of the variability in the RMSE estimates.. This comparison highlights the robustness and reliability of the Random Forest model in this analysis.

Final Model Discussion:

Based on the comprehensive analysis and evaluation of various models, the Random Forest model emerged as the final choice due to its superior performance in terms of RMSE. The model was meticulously tuned to optimize its parameters, including the number of variables considered at each split (mtry), the number of trees in the forest (trees), and the minimum number of data points in a node (min_n). The final selected hyperparameters were Mtry = 10, Splitrule = “extratrees”, Min.node.size = 10, Importance = “impurity”. This tuning process resulted in the Random Forest model achieving an RMSE of .0813 and an SE of .0015 on the training data, along with a competitive Kaggle score of 0.001596 on the private leaderboard, placing it in the top 5 of the competition.

The Random Forest model's strengths include its ability to handle complex datasets effectively, consistently produce accurate predictions, and its robustness against overfitting due to the ensemble nature of decision trees. However, it is computationally intensive to train and tune, especially with large datasets, and can be less interpretable compared to simpler models like linear regression. Despite these weaknesses, the Random Forest model's strengths in handling complexity and providing accurate predictions made it the optimal choice for this project. Future efforts may focus on further refining this model and exploring additional methods to enhance its performance.

The Random Forest model demonstrated strong predictive capabilities in analyzing the data, securing a fifth ranking in the Kaggle competition. Its approach effectively managed the complexities inherent in the dataset, mitigating the risks of overfitting and enhancing prediction accuracy. However, this model underscored several areas for potential enhancement, which could have been addressed to improve overall performance and efficiency. A potential improvement could be the Advanced Hyperparameter Tuning: The tuning process employed a conventional grid search method. While effective, more sophisticated approaches like Bayesian optimization or genetic algorithms could have been utilized. These methods can explore the parameter space more efficiently and identify optimal settings that might be missed by traditional methods. Such fine-tuning could lead to marginal but meaningful improvements in model performance. Furthermore, we could have explored the other provided dataset related to Amazon more thoroughly and incorporated additional information into our analysis. This expanded data could have provided deeper insights and been particularly helpful throughout our investigation.

Appendix: Final Annotated Script

```
library(ranger)
library(caret)
library(tidyverse)
set.seed(146)
train <- read_csv("train.csv")
test <- read_csv("test.csv")
train <- train %>%
  select(-order_totals)

#Tuning Parameter Grid
param_grid <- expand.grid(
  mtry = seq(2, 10, by = 2),
  splitrule = c("variance", "extratrees"),
  min.node.size = c(1, 5, 10)
)
#Cross-Fold Validation
ctrl <- trainControl(
  method = "cv",
  number = 5,
  verboseIter = TRUE,
```

```
)

#Tuning Random Forest based on Parameter Grid
rf_tuned <- train(
  log_total ~ .,
  data = train,
  method = "ranger",
  trControl = ctrl,
  tuneGrid = param_grid,
  importance = 'permutation',
  num.trees = 100
)

#Predictions
rf_predictions <- predict(rf_tuned, newdata = test)

#Formatting Data for Submissions
predictions <- test %>%
  select(id) %>%
  bind_cols(tibble(log_total = rf_predictions))

# Save the predictions to a CSV file
write_csv(predictions, "5_cv_predictions.csv")
```

Appendix: Team Member Contribution

Albert:

- Support vector regression model, report editing, writing.

Benedetta:

- Initial random forest coding for the candidate model, report writing, editing and organizing.

James:

- Final random forest coding for the candidate model, Neural Network candidate model coding, script for the random forest final model.

Kyle:

- Created Gradient Boosted candidate model code and report about boosting trees, and report.

Mikayla:

- Exploratory analysis graphs and initial descriptions, Linear regression model, report.