Final Project

Xilong Li (3467966)

2022-06-09

Contents

Introduction

The purpose of this project is to generate a model that will predict the gross profits of a commodity type: "cookies" and "pastry", based on the commodity's certain properties.
I obtained this data from a large local grocery retail company when I was working as an intern for six months. During my internship, I was authorized to download limited amount of selling data. However, due to the confidential requirement of the company, the data used in this project is still intentionally selected in a very limited amount, and the data itself is also mutated.
Despite this, this project still serves a function of exploring the potential ways of using machine learning skills to interpret past sales data, in order to better predict the performance of each commodity.
Such prediction is important because this can help the retail company to evaluate and analyze the selling performance of each commodity, helping the manager to come out of a better and more comprehensive decision on whether or not to exclude this commodity from the store shell.
Therefore, although the result and method in this project might be far from the standard of being used commercially, it is still a good chance of personal practice and exploration on the issue of predicting the product's gross profit, which, according to my internship experience, had not been widely used by my company.

Loading Data and Packages

```
library(tidymodels)
library(tidyverse)
library(MASS)
library(glmnet)
library(janitor)
```

```
library(discrim)
library(poissonreg)
library(klaR)
library(rpart.plot)
library(vip)
library(randomForest)
library(xgboost)
library(corrplot)
library(ranger)
```

The codebook is explained below:

- gross_profit: The gross profit that this specific commodity has created in the selected period of time;

- category: Four specific categories of the the commodity, includes Cookies, Pastry, Biscuit, and Wafer;

- volume: The volume of the commodity, calculated in grams;

- volume_range: The ranged volume of the commodity;
- package: The package of the commodity, includes Can, Bag, and Box;
- flavor: The flavor of the commodity;
- price: The selling price of the commodity;

- StorePerOrder: The number of store per order of this commodity;
- AmountPerOrder: The number of sold amount per order of this commodity;
- Sale_Store: The number of stores that is selling this commodity;
- initial_days: The number of days since this commodity was being sold;

It is also important to mention that, since the original data was extracted from the data base in a Chinese version, I had to manually convert all the Chinese vocabularies into English, in order for R to accurately load the data.

```
# Reading the original data;
original_data <- read.csv("Translated_data.csv")
head(original_data)
```

gross_profit	category	volume	volume_range	package	flavor	price	StorePerOrder	AmountPerOrder	Sale_Store	initial_days	X
15.059149	Biscuit	115g	100-299g/ml	Bag	Original	3.5	7.2631	1.8930085	198	2841	NA
0.000000	Cookies	44g	0-149g/ml	Bag	Chocolate	4.9	0.0000	0.0000000	0	1220	NA
4.763544	Pastry	60g	0-149g/ml	Box	Vanilla	5.0	10.5572	2.2596862	80	2312	NA
103.966349	Pastry	84g	0-149g/ml	Bag	Original	5.5	1.9736	0.2974709	215	2841	NA
41.006804	Biscuit	168g	150-199g/ml	Bag	Original	6.5	5.4526	0.8248753	172	2841	NA
9.234831	Biscuit	60g	0-149g/ml	Box	Strawberry	6.9	15.9553	2.5869891	32	420	NA

```r
set.seed(2216)
```

Data Cleansing

```r
# use clean_names() function to clean the column names of the data
original_data <- original_data %>%
  clean_names()

head(original_data)
```

gross_profit	category	volume	volume_range	package	flavor	price	store_per_order	amount_per_order	sale_store	initial_days	x
15.059149	Biscuit	115g	100-299g/ml	Bag	Original	3.5	7.2631	1.8930085	198	2841	NA
0.000000	Cookies	44g	0-149g/ml	Bag	Chocolate	4.9	0.0000	0.0000000	0	1220	NA
4.763544	Pastry	60g	0-149g/ml	Box	Vanilla	5.0	10.5572	2.2596862	80	2312	NA
103.966349	Pastry	84g	0-149g/ml	Bag	Original	5.5	1.9736	0.2974709	215	2841	NA
41.006804	Biscuit	168g	150-199g/ml	Bag	Original	6.5	5.4526	0.8248753	172	2841	NA
9.234831	Biscuit	60g	0-149g/ml	Box	Strawberry	6.9	15.9553	2.5869891	32	420	NA

The code below shows the selections process of the data that is going to be used to train the model: 1. turns the predictors "category", "flavor", "package" and "volume_range" into factor; 2. Since the original data about "volume" included the unit "grams", I deleted all the "g" from each data point and turn it into numeric; 3. Selected "category, volume, volume_range, package, flavor, price, store_per_order, amount_per_order, sale_store, initial_days" as predictors for "gross_profit" 4. deleted data that has zero value;

```r
selected_data <- original_data %>%
  mutate(category = factor(category),
         volume_range = factor(volume_range),
         package = factor(package),
         flavor = factor(flavor),
         volume = unlist(strsplit(volume, split='g', fixed=TRUE)),
         volume = as.numeric(volume)) %>%
  dplyr::select(
    gross_profit,
    category,
    volume,
    volume_range,
    package,
    flavor,
    price,
    store_per_order,
    amount_per_order,
    sale_store,
    initial_days) %>%
```

```
  filter(gross_profit != 0)

head(selected_data)
```

gross_profit	category	volume	volume_range	package	flavor	price	store_per_order	amount_per_sale	order_store	initial_days
15.059149	Biscuit	115	100-299g/ml	Bag	Original	3.5	7.2631	1.8930085	198	2841
4.763544	Pastry	60	0-149g/ml	Box	Vanilla	5.0	10.5572	2.2596862	80	2312
103.966349	Pastry	84	0-149g/ml	Bag	Original	5.5	1.9736	0.2974709	215	2841
41.006804	Biscuit	168	150-199g/ml	Bag	Original	6.5	5.4526	0.8248753	172	2841
9.234831	Biscuit	60	0-149g/ml	Box	Strawberry	6.9	15.9553	2.5869891	32	420
48.648763	Pastry	40	0-149g/ml	Bag	Chocolate	6.9	8.8013	1.1966070	31	198

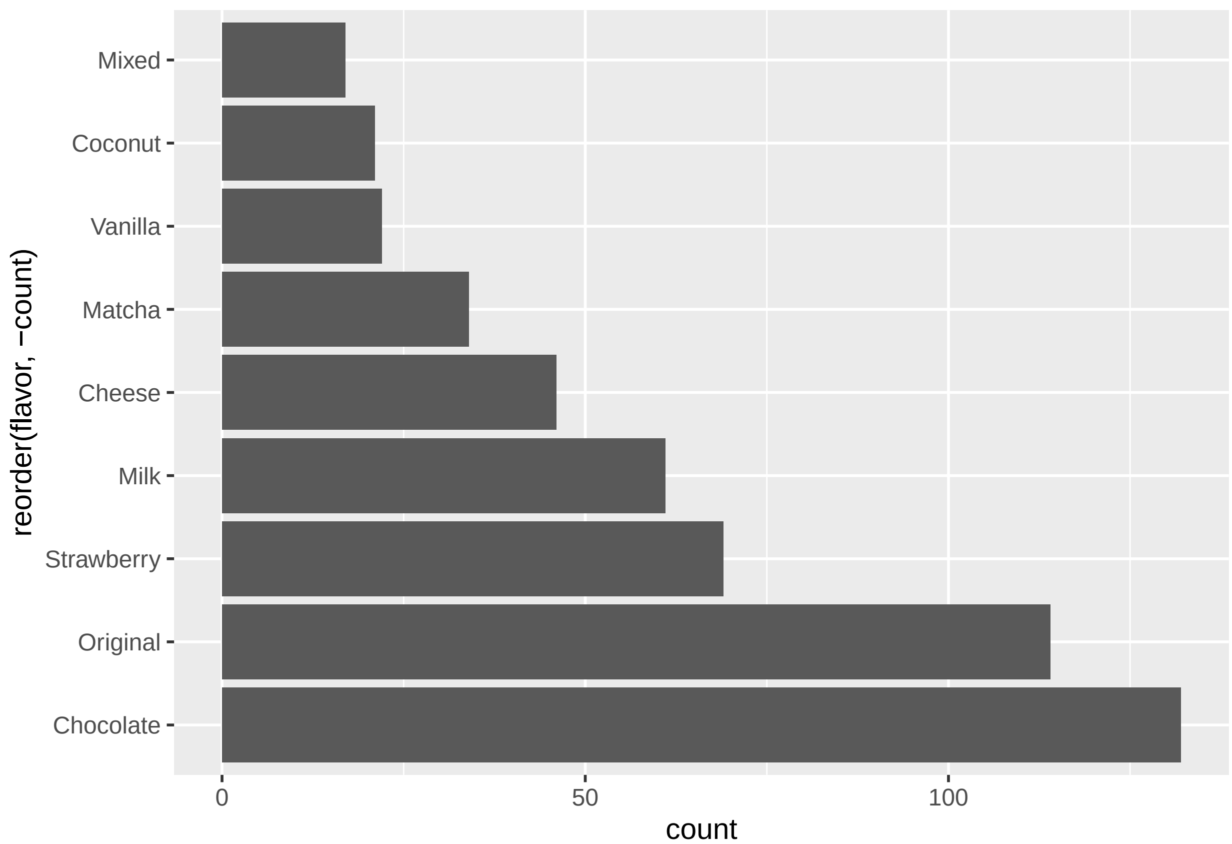
Data Splitting and Cross-Validation

```
sales_split <- initial_split(selected_data, prop = 0.80) # set the probability as 80%

sales_train <- training(sales_split)
sales_test <- testing(sales_split)

train_folds <- vfold_cv(sales_train, v = 5) # establish the fold level as 5;
```

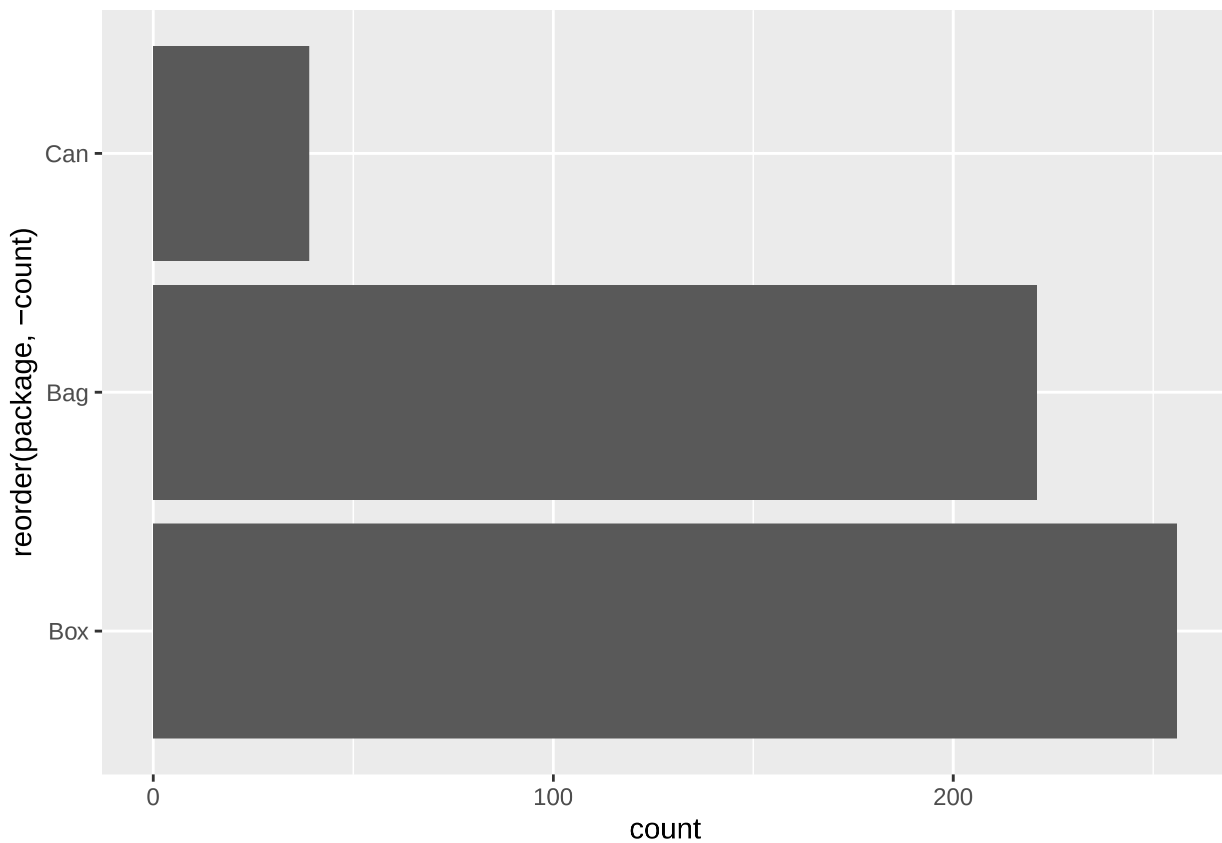
Exploratory Data Analysis

- In order to analyze my data set, I first realized that there all many categorical predictors that might affect the prediction results;
- Therefore, I counted the numbers of three predictors: "flavor", "package", "category" to understand the specific distributions in these three predictors;
- And the results below shows that some factors in each predictors are outnumbered by the others, such as "Mixed" in the "flavor" predictor and "can" in the "package" predictor;
- As a result, such uneven distribution of interior factors in these predictors might affect the accuracy of the model training and prediction;

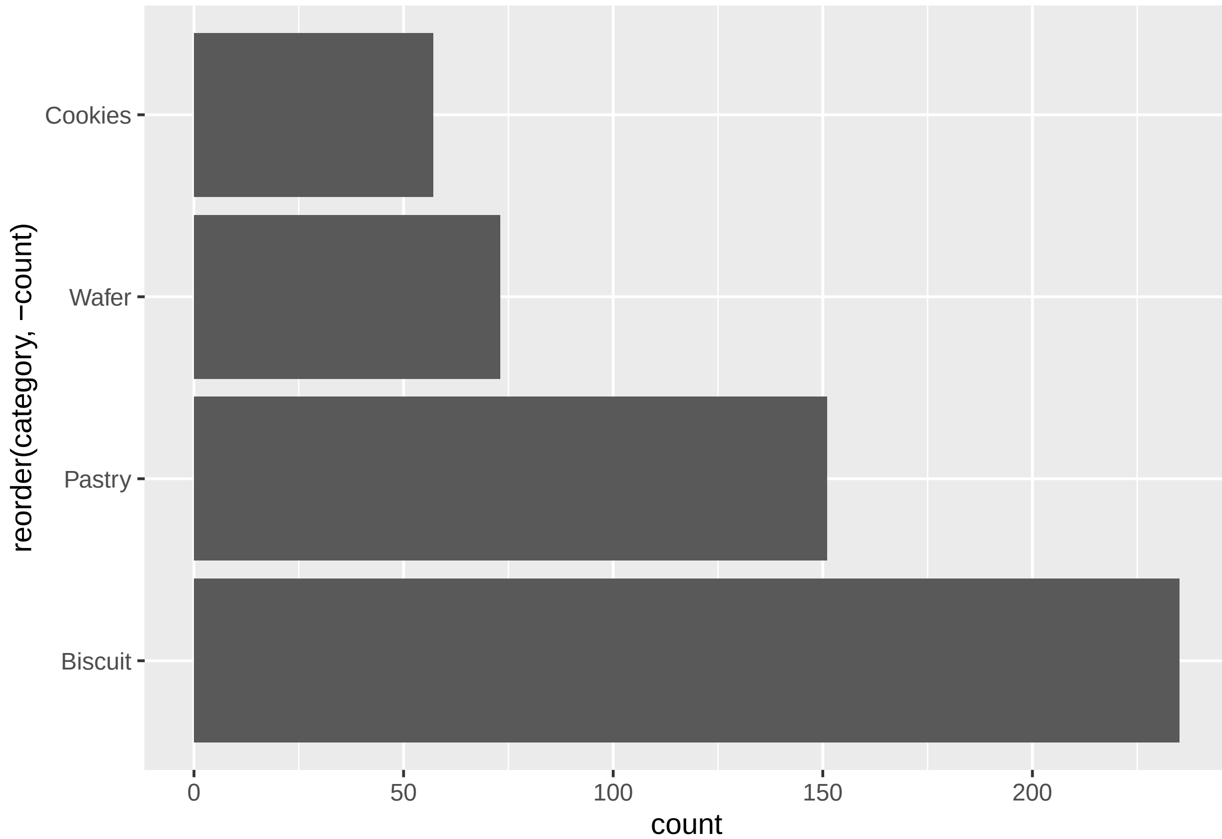
```
ordered_data_1 <- selected_data %>%
  group_by(flavor) %>%
  summarise(count = n()) %>%
  arrange(count)
ggplot(ordered_data_1, aes(x = count, y = reorder(flavor, -count))) + geom_bar(stat = "identity")
```




```
ordered_data_2 <- selected_data %>%
  group_by(package) %>%
  summarise(count = n()) %>%
  arrange(count)
ggplot(ordered_data_2, aes(x = count, y = reorder(package, -count))) + geom_bar(stat = "identity")
```

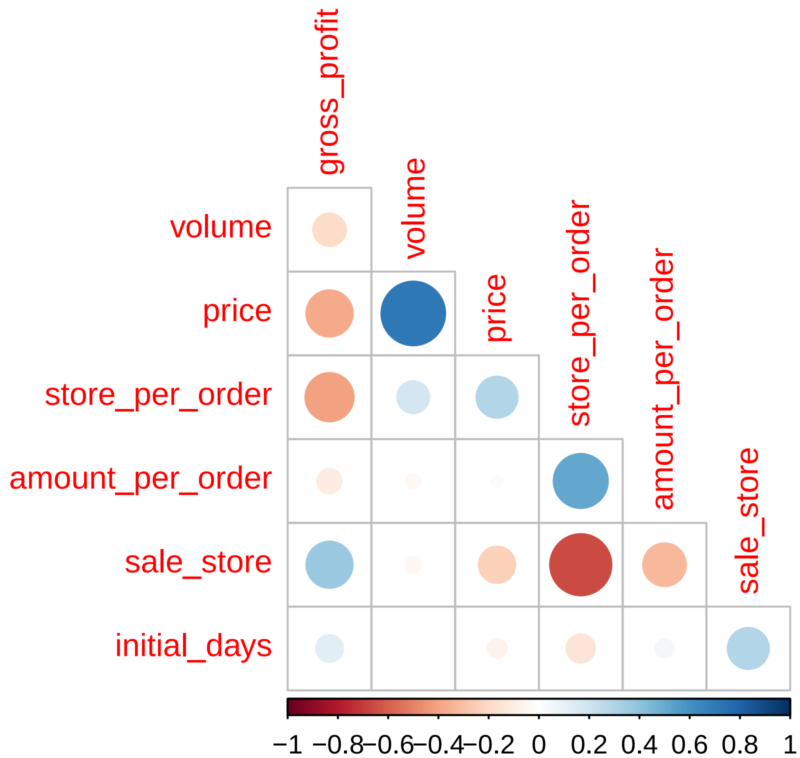



```
ordered_data_3 <- selected_data %>%
  group_by(category) %>%
  summarise(count = n()) %>%
  arrange(count)
ggplot(ordered_data_3, aes(x = count, y = reorder(category, -count))) + geom_bar(stat = "identity")
```



- I also used the correlation graph to evaluate the interactions between each numeric predictor;
- And it shows that price has a very strong positive correlation with volume; It makes sense since that the larger the volume of the package, the higher the price would be.
- And sale_store has very high negative correlation with store_per_order, which also makes sense since store_per_order is directly calculated based on the number of stores;

```
cor_data <- sales_train %>%
  dplyr::select(where(is.numeric))
corrplot(cor(cor_data), type = 'lower',diag = FALSE)
```



Model Fitting

Setting Up Recipe

- Then, we establish the recipe that can be used in later modeling;
- I excluded the predictor "volume_range" since it keeps showing errors when I was trying to fit. So I used "volume" instead since they express the same information;
- Center and scale the predictors and establish dummy variables (category, package, flavor);
- I also added the interaction between volume and package, since it might be possible that different volume requires different package.

```
sales_recipe <- recipe(gross_profit ~
                         category+
                         volume+
                         package+
                         flavor+
                         price+
                         store_per_order+
                         amount_per_order+
                         sale_store+
                         initial_days,
                       data = sales_train) %>%
  step_dummy(category, package, flavor) %>%
  step_interact(~ volume:starts_with("package")) %>%
  step_center(all_predictors()) %>%
```

```
  step_scale(all_predictors())

sales_recipe
```

```
## Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor          9
##
## Operations:
##
## Dummy variables from category, package, flavor
## Interactions with volume:starts_with("package")
## Centering for all_predictors()
## Scaling for all_predictors()
```

1. Linear Regression Model

```
lm_model <- linear_reg() %>%
  set_engine("lm")

lm_wflow <- workflow() %>%
  add_model(lm_model) %>%
  add_recipe(sales_recipe)
```

- We create the fit function and save the result:

```
lm_fit <- fit(lm_wflow, sales_train)
lm_fit %>%
  # This returns the parsnip object:
  extract_fit_parsnip() %>%
  # Now tidy the linear model object:
  tidy()

# Write Out Results & Workflow:
save(lm_fit, lm_wflow, file = "Data/model_fitting/lm_fit.rda")
```

2. Ridge Regression Model

Set up the model specification.

```
ridge_spec <-
  linear_reg(penalty = tune(), mixture = 0) %>%
  set_mode("regression") %>%
  set_engine("glmnet")
```

Create a workflow object.

```r
ridge_workflow <- workflow() %>%
  add_recipe(sales_recipe) %>%
  add_model(ridge_spec)
```

Creates a grid of evenly spaced parameter values.

```r
penalty_grid <- grid_regular(penalty(range = c(-5, 5)), levels = 50)
penalty_grid
```

penalty
1.000000e-05
1.600000e-05
2.560000e-05
4.090000e-05
6.550000e-05
1.048000e-04
1.677000e-04
2.683000e-04
4.292000e-04
6.866000e-04
1.098500e-03
1.757500e-03
2.811800e-03
4.498400e-03
7.196900e-03
1.151400e-02
1.842070e-02
2.947050e-02
4.714870e-02
7.543120e-02
1.206793e-01
1.930698e-01
3.088844e-01
4.941713e-01
7.906043e-01
1.264855e+00
2.023590e+00
3.237458e+00
5.179475e+00
8.286428e+00
1.325711e+01
2.120951e+01
3.393222e+01
5.428675e+01
8.685114e+01
1.389495e+02
2.222996e+02
3.556480e+02
5.689866e+02
9.102982e+02
1.456348e+03

penalty
2.329952e+03
3.727594e+03
5.963623e+03
9.540955e+03
1.526418e+04
2.442053e+04
3.906940e+04
6.250552e+04
1.000000e+05

Tune the model and save the result for later use.

```
ridge_tune <- tune_grid(
  ridge_workflow,
  resamples = train_folds,
  grid = penalty_grid
)

# Write Out Results & Workflow:
save(ridge_tune, ridge_workflow, file = "Data/model_fitting/ridge_tune.rda")
```

3. Tree Decision Model

Set up the model specification.

```
reg_tree_spec <- decision_tree() %>%
  set_engine("rpart") %>%
  set_mode("regression")
```

Create a workflow object.

```
reg_tree_wf <- workflow() %>%
  add_model(reg_tree_spec %>% set_args(cost_complexity = tune())) %>%
  add_recipe(sales_recipe)
```

Creates a grid of evenly spaced parameter values.

```
param_grid <- grid_regular(cost_complexity(range = c(-4, -1)), levels = 10)
```

Tune the model and save the result for later use.

```
tree_tune <- tune_grid(
  reg_tree_wf,
  resamples = train_folds,
  grid = param_grid
)

# Write Out Results & Workflow:
save(tree_tune, reg_tree_wf, file = "Data/model_fitting/tree_tune.rda")
```

4. Random Forest Model

Set up the model specification.

```
rf_spec <- rand_forest(mtry = tune(), trees = tune(), min_n = tune()) %>%
  set_engine("ranger", importance = "impurity") %>%
  set_mode("regression")
```

Create a workflow object.

```
rf_wf <- workflow() %>%
  add_recipe(sales_recipe) %>%
  add_model(rf_spec)
```

Creates a grid of evenly spaced parameter values.

```
rf_grid <- grid_regular(mtry(range = c(3,8)), trees(range = c(5,2000)),
                        min_n(range = c(3,8)), levels = 5)
```

Tune the model and save the result for later use.

```
rf_tune <- tune_grid(
  rf_wf,
  resamples = train_folds,
  grid = rf_grid
)

# Write Out Results & Workflow:
save(rf_tune, rf_wf, file = "Data/model_fitting/rf_tune.rda")
```

5. Boosted Tree Model

Set up the model specification.

```
boost_spec <- boost_tree(trees = tune()) %>%
  set_engine("xgboost") %>%
  set_mode("regression")
```

Create a workflow object.

```
boost_wf <- workflow() %>%
  add_model(boost_spec) %>%
  add_recipe(sales_recipe)
```

Creates a grid of evenly spaced parameter values.

```
boost_grid <- grid_regular(trees(range = c(10,2000)), levels = 10)
```

Tune the model and save the result for later use.

```r
boost_tune <- tune_grid(
  boost_wf,
  resamples = train_folds,
  grid = boost_grid
)

# Write Out Results & Workflow:

save(boost_tune, boost_wf, file = "Data/model_fitting/boost_tune.rda")
```

Model Selection and Performance

Load Previous Saved Data

```r
load("data/model_fitting/lm_fit.rda")
load("data/model_fitting/ridge_tune.rda")
load("data/model_fitting/tree_tune.rda")
load("data/model_fitting/rf_tune.rda")
load("data/model_fitting/boost_tune.rda")
```

1. Linear Regression Model

- After fitting the Linear Regression Model, we now compare the fitted value of gross profit to the actual observed value.

```r
sales_train_res <- predict(lm_fit, new_data = sales_train %>%
                               dplyr::select(-gross_profit))
sales_train_res <- bind_cols(sales_train_res, sales_train %>%
                               dplyr::select(gross_profit))
```

- Thus a generated datafame is shown below:

```r
sales_train_res %>%
  head()
```

.pred	gross_profit
-81.355342	-33.520424
-34.205174	11.762315
24.775899	3.082706
24.692187	5.254099
-3.752874	29.757399
19.656663	5.973426

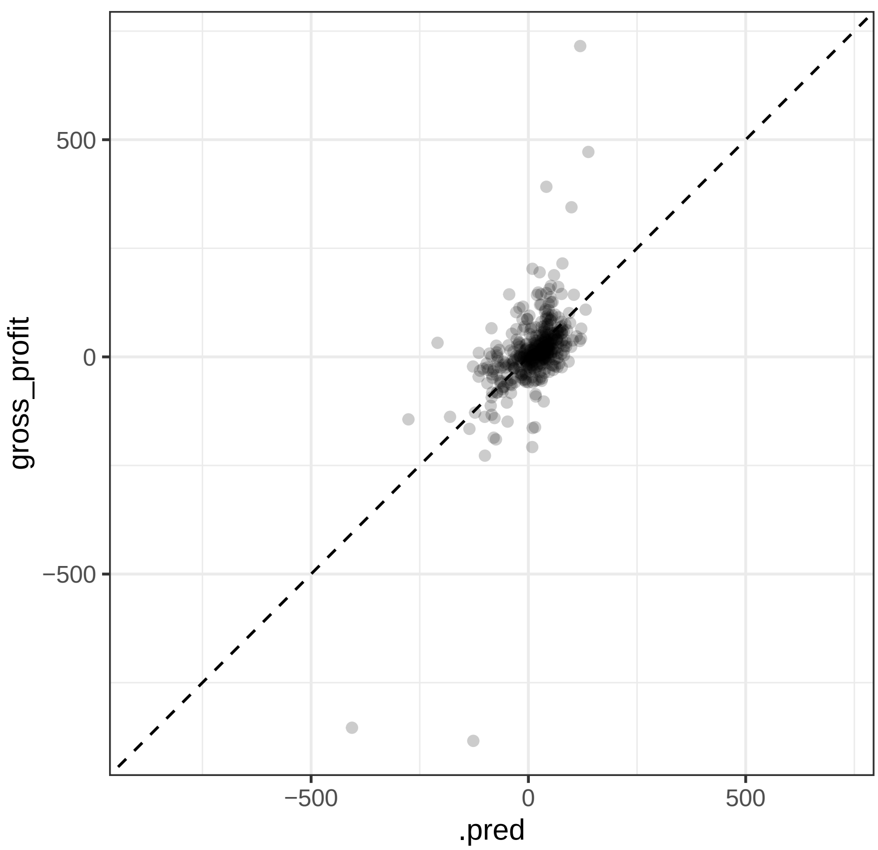
- Then, we decide to calcultae “rmse”, “rsq”, “mae” values to evaluate the performance of this model:

```r
sales_metrics <- metric_set(rmse, rsq, mae)
sales_metrics(sales_train_res, truth = gross_profit,
                estimate = .pred)
```

.metric	.estimator	.estimate
rmse	standard	80.7683158
rsq	standard	0.3348636
mae	standard	47.2504689

- As we can see, the rmse value is larger than 80, which is really high, and thus this result indicates that the linear regression model might not be suitable to achieve our goal.
- We can also plot and visualize the result:

```r
sales_train_res %>%
  ggplot(aes(x = .pred, y = gross_profit)) +
  geom_point(alpha = 0.2) +
  geom_abline(lty = 2) +
  theme_bw() +
  coord_obs_pred()
```

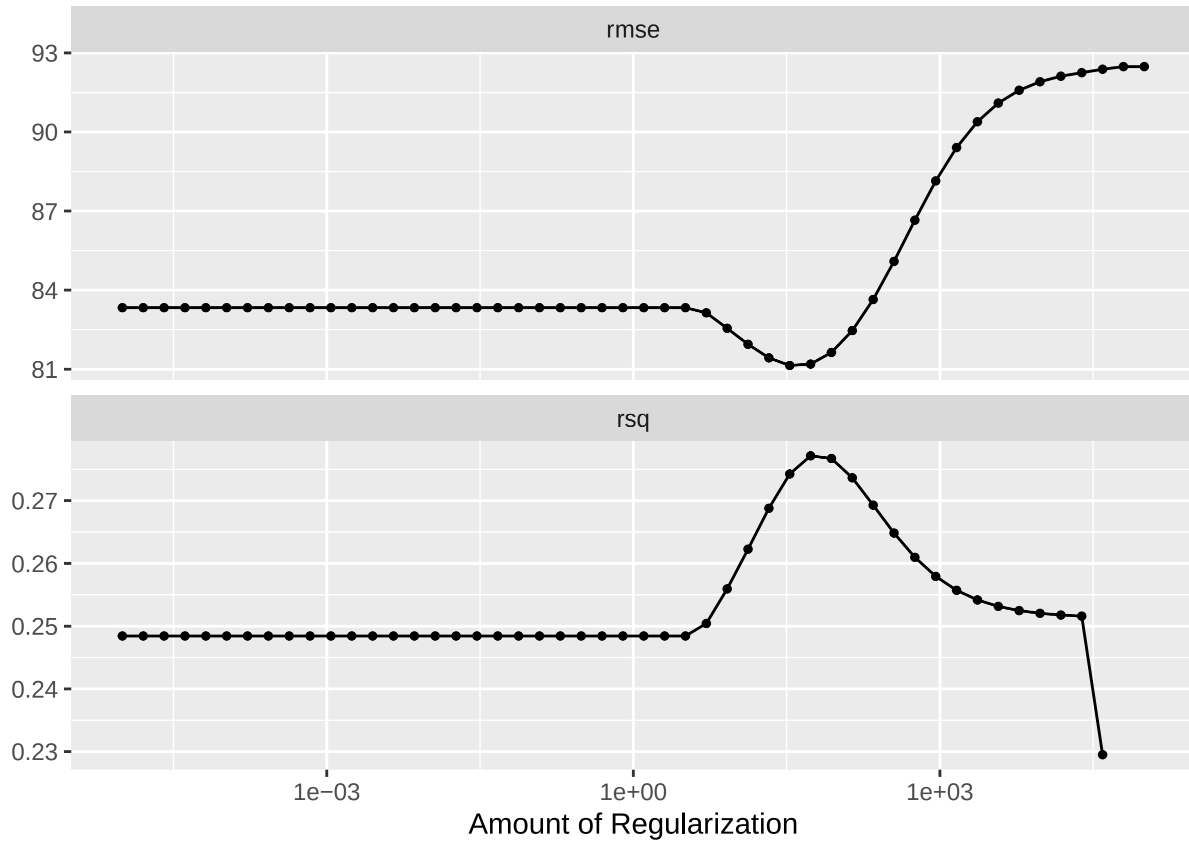


- As it is shown above, all the observed values of gross profit concentrate in a certain area, and there are also many outliers that can influence the prediction.
- Therefore, this linear regression model might NOT be a good way to predict the gross profit.

2. Ridge Regression Model

- After obtaining the output of tune_grid(),we first use autoplot() to create a visualization of the result:

```
autoplot(ridge_tune)
```



- As we can see in the graph, there is a certain point that rmse reaches the lowest level;
- Thus we use show_best() to find the lowest rmse of the best turning ridge model:

```
show_best(ridge_tune, metric = "rmse") %>% dplyr::select(-.estimator, -.config)
```

penalty	.metric	mean	n	std_err
33.93222	rmse	81.13850	5	16.70875
54.28675	rmse	81.19072	5	17.18792
21.20951	rmse	81.42716	5	16.21721
86.85114	rmse	81.63386	5	17.60988
13.25711	rmse	81.94270	5	15.76210

- As it is shown above, the lowest rmse is 77.36626 for the ridge regression model, which is lower than the linear regression model;
- It is also shown that the best penalty is at the level of 33.93222;

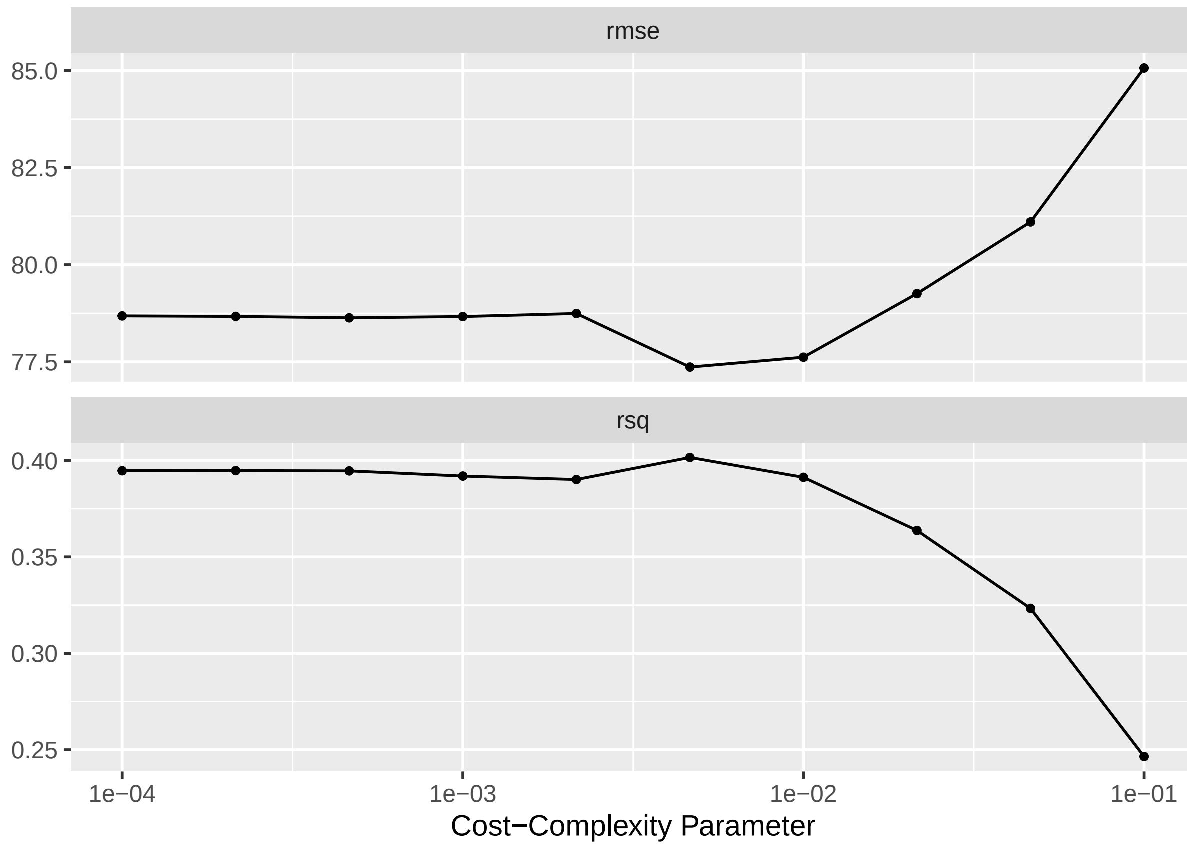
```
best_penalty <- select_best(ridge_tune, metric = "rmse")
best_penalty
```

penalty	.config
33.93222	Preprocessor1_Model33

3. Tree Decision Model

- For the Tree Decision Model, we use the same analysis procedure;
- We first use the autoplot() function to visulize the tuning result:

```
autoplot(tree_tune)
```



- The graph above shows that rmse reaches the lowest level when the cost-complexity parameter is in between 1e-03 and 1e-02;
- Then we use show_best() and notice that the best rmse of the Tree Decision model is very close to the lowest rmse of Ridge Regression model;

```
show_best(tree_tune)
```

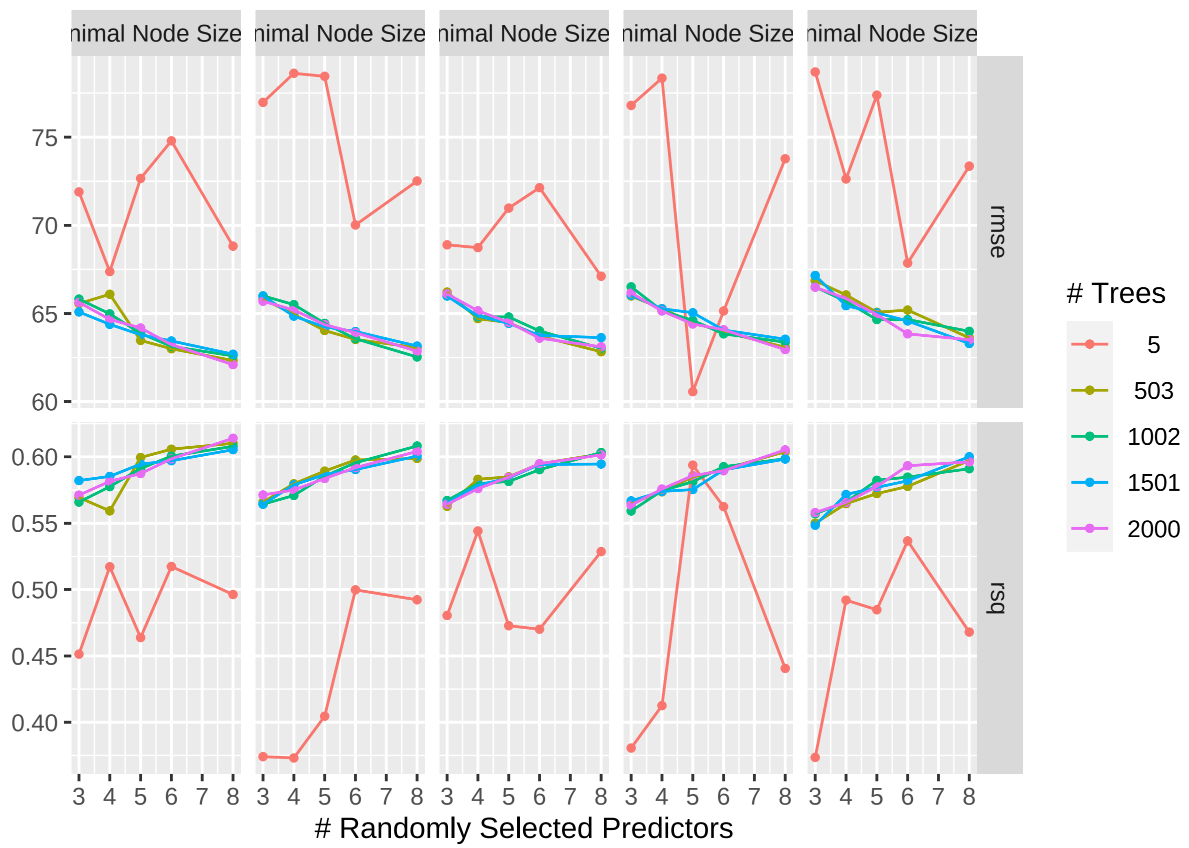
```
## Warning: No value of ‘metric‘ was given; metric ’rmse’ will be used.
```

cost_complexity	.metric	.estimator	mean	n	std_err	.config
0.0046416	rmse	standard	77.36626	5	11.38199	Preprocessor1_Model06
0.0100000	rmse	standard	77.61994	5	11.31526	Preprocessor1_Model07
0.0004642	rmse	standard	78.63397	5	11.89005	Preprocessor1_Model03
0.0010000	rmse	standard	78.66672	5	11.89160	Preprocessor1_Model04
0.0002154	rmse	standard	78.67046	5	11.89987	Preprocessor1_Model02

4. Random Forest Model

- We first visualize the tuning result of Random Forest Model:

```
autoplot(rf_tune)
```



- As it is shown above, when the tree number is only five, the results are highly variable, but when the number of trees gets really high, the graph shows that a larger node size results a smaller value of rmse;
- Then we select the specific best tuning model:

```
show_best(rf_tune)
```

```
## Warning: No value of `metric` was given; metric 'rmse' will be used.
```

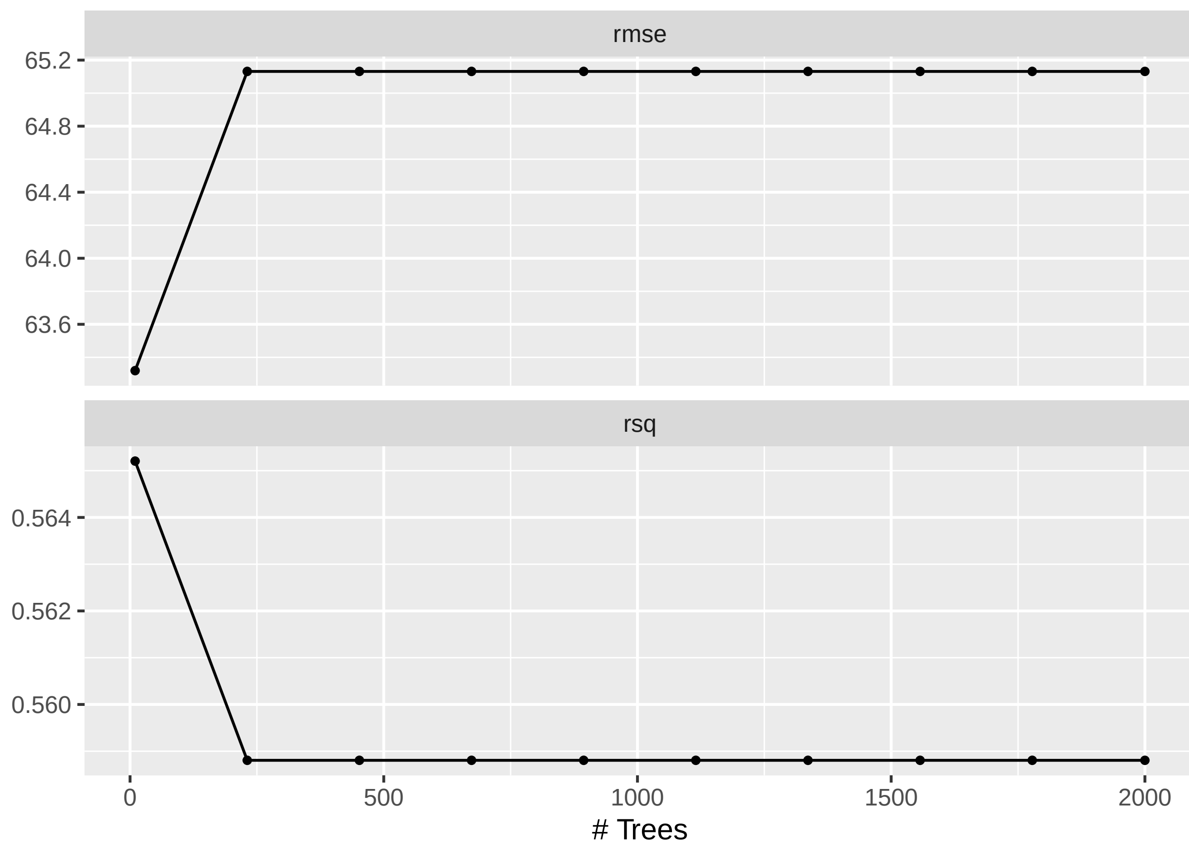
mtry	trees	min_n	.metric	.estimator	mean	n	std_err	.config
5	5	6	rmse	standard	60.56217	5	12.81039	Preprocessor1_Model078
8	2000	3	rmse	standard	62.09721	5	15.13394	Preprocessor1_Model025
8	503	3	rmse	standard	62.32769	5	14.98286	Preprocessor1_Model010
8	1002	4	rmse	standard	62.53059	5	15.09913	Preprocessor1_Model040
8	1002	3	rmse	standard	62.59266	5	15.19247	Preprocessor1_Model015

- It can be noticed above that the overall rmse is significantly smaller than three previous models, which indicates that random forest might be a better model;
- However, it also shows that the influence of the number of trees on the rmse is highly variable.

5. Boosted Tree Model

- Again, we first visualize the tuning result:
- what's interesting is that the graph below shows a contradicting result to the random forest model:
- it shows that when the smaller the number of trees is, the smaller the rmse is.

```
autoplot(boost_tune)
```



- We then select the best tuning model:

```
show_best(boost_tune)
```

```
## Warning: No value of `metric` was given; metric 'rmse' will be used.
```

trees	.metric	.estimator	mean	n	std_err	.config
10	rmse	standard	63.31936	5	15.31624	Preprocessor1_Model01
231	rmse	standard	65.13161	5	14.71023	Preprocessor1_Model02
452	rmse	standard	65.13161	5	14.71023	Preprocessor1_Model03
673	rmse	standard	65.13161	5	14.71023	Preprocessor1_Model04
894	rmse	standard	65.13161	5	14.71023	Preprocessor1_Model05

- Therefore, it shows that the rmse is lowest when the number of trees is 10, which is the lowest level as well;
- And it also contains a lower rmse than the first three models, while it is still higher than the rmse of the random forest model;

Final Model Selection and Testing

- According to the analysis above, we find that the Random forest model has the lowest average rmse level, and thus I decide to use the Random Forest model as the final model to do the testing.
- Thus, we first select the best Random Forest Model, which is the model when mtry = 5, trees = 5, min_n = 5;

```
best_model <- select_best(rf_tune, metric = "rmse")
best_model
```

mtry	trees	min_n	.config
5	5	6	Preprocessor1_Model078

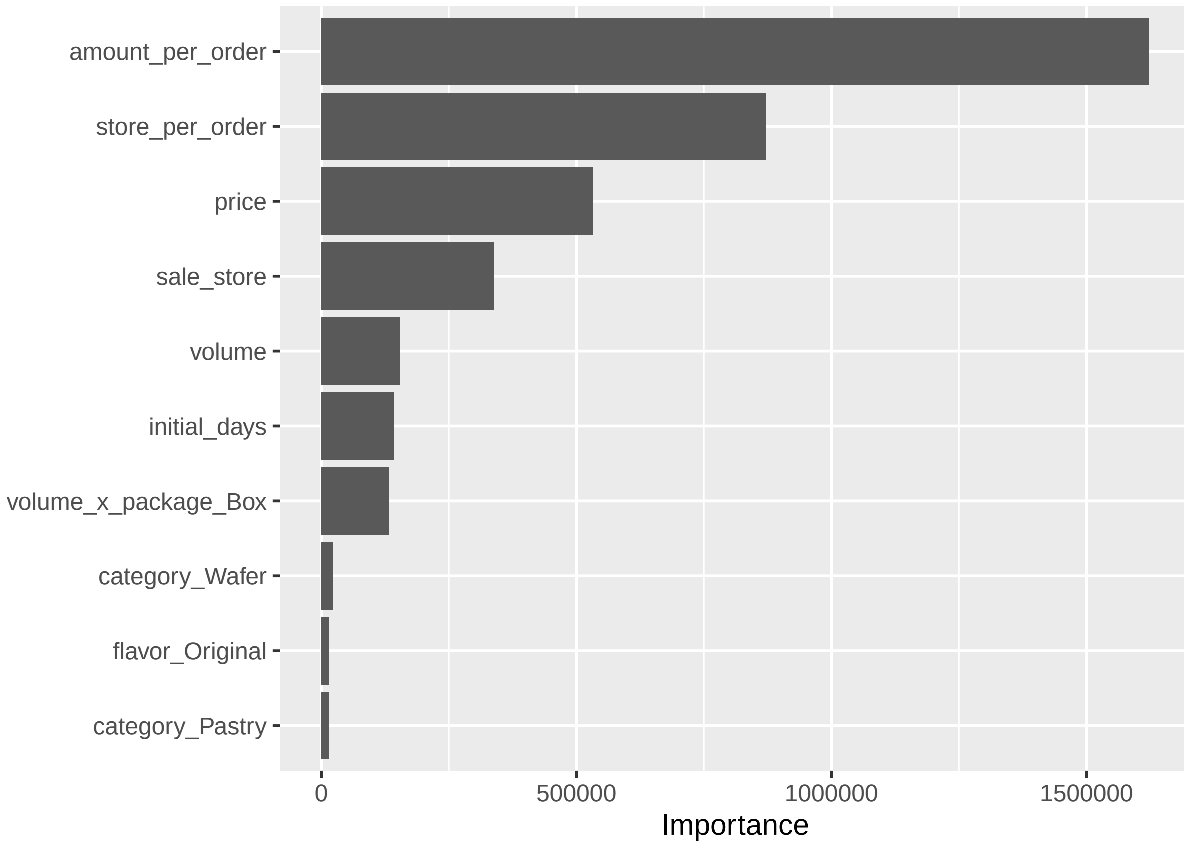
- Then, we fit the model with

```
final_rf <- finalize_workflow(rf_wf, best_model)
final_fit <- fit(final_rf, data = sales_train)

augmented_result <- augment(final_fit, new_data = sales_test)

augment(final_fit, new_data = sales_test) %>%
  rmse(truth = gross_profit, estimate = .pred)
```

.metric	.estimator	.estimate
rmse	standard	48.71332

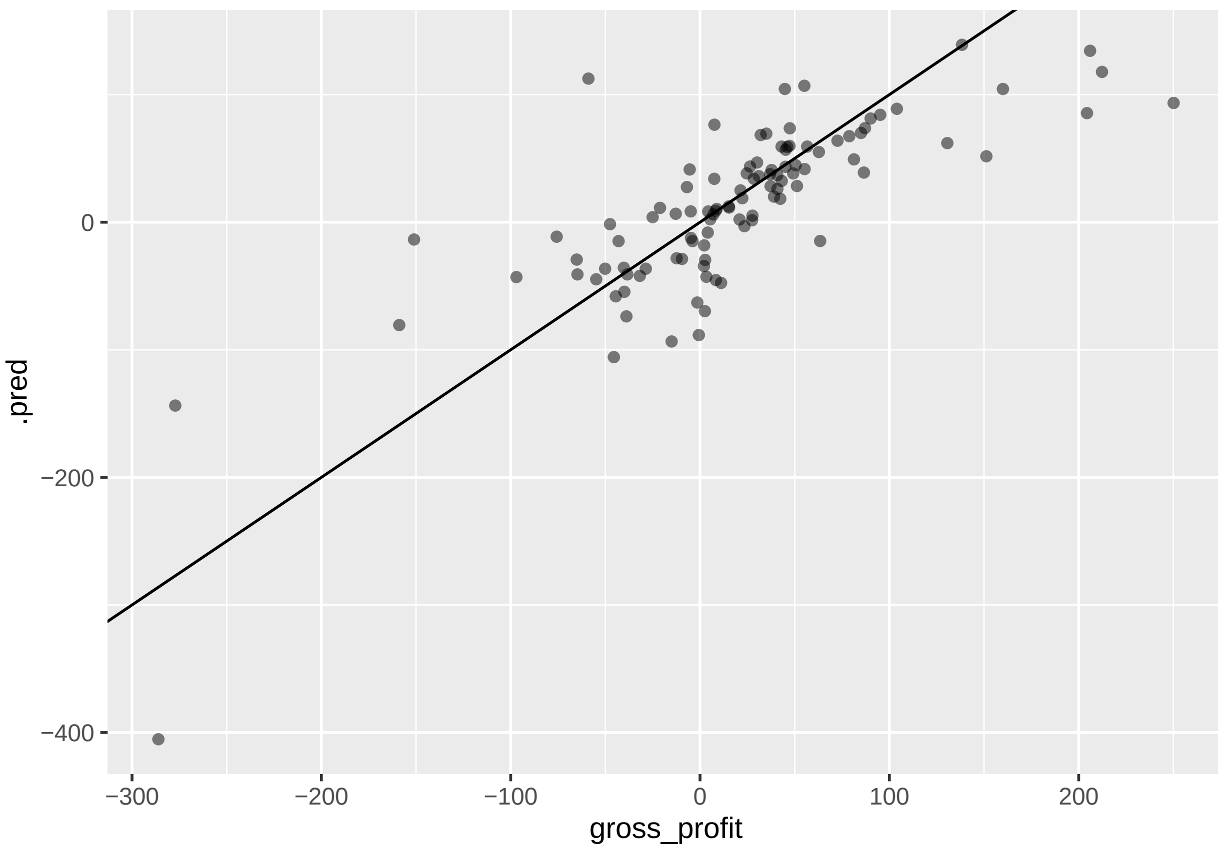
- As we can see, the tested fit of rmse is 62.58235, which is similar to the rmse level in the training set;
- We can also use the vip() function to evaluate the importance of each predictors in this model:

```r
final_fit %>%
  extract_fit_engine() %>%
  vip()
```



- Finally, we use ggplot() to visualize our prediction:

```r
augment(final_fit, new_data = sales_test) %>%
  ggplot(aes(gross_profit, .pred)) +
  geom_abline() +
  geom_point(alpha = 0.5)
```



- As it is shown in the graph above, this prediction looks better than the linear regression model, since the observed points are no longer highly concentrated.
- However, there are still many outliers that can reduce the accuracy of prediction

Conclusion

- In conclusion, this project strives to use five different models in order to predict the possible gross profits of the retail commodities about cakes.
- Among the five models, the Random Forest Model has the best performance and is thus selected to test.
- However, even though the Random Forest model does a relatively better job, its resulting rmse is still much higher than expected, which means that it is still not a good model to predict the gross profit.
- There are many possible explanations of this:

  1. It might be because that the training and testing data is highly insufficient, and thus the model cannot be trained well to make good predictions;
  2. It might be because that there are insufficient number of predictors in this model, which result in an inaccurate trained model;
  3. It is also worth noticing that two predictors (store_per_order & amount_per_order) were incorrectly selected, because these two predictors are originally calculated by the number of sold orders, which directly contribute the gross profit.

- Therefore, this model can definitely have more and better improvements in the future developments.