Pstat231 Project

Zihao Yang

2022-05-11

Introduction

Figure 1: Image Source: https://ethereum.org/en/

The purpose of this project is to predict the value of Ether with the historical price of top 10 Cryptocurrencies excluding itself (Binance coin, Bitcoin, Cardano, Dogecoin, Litecoin, Monero, Stellar, Tether, USD coin, and XRP).

What Is Ethereum?

Ethereum is a decentralized, open-source blockchain with smart contract functionality. Ether (ETH) is the native cryptocurrency of the platform. Among cryptocurrencies, Ether is second only to Bitcoin in market capitalization.

Ethereum allows anyone to deploy permanent and immutable decentralized applications onto it, with which users can interact. It means that the market is not influenced by the government, and the currency cannot be inflated. In simple words, Ether belongs to one of the virtual encrypted digital currencies. People can use it to trade online and it is recognized by many institutions. Its security can be guaranteed, and transactions can be conducted anytime and anywhere in the network environment, which is why it has become very popular in recent years.

Here is a short video about the Ethereum and Ether for detailed instruction:

```r
embed_youtube("IsXvoYeJxKA")
```

Why Ether?

We chose Ether because it is second only to Bitcoin in market share. However, the mechanics of Ether and Bitcoin are not quite the same. Through long-term research, it was found that Ether has great market potential, and it will not become less and less like the undiscovered Bitcoin over time. In addition, Bitcoin was born earlier, and there has been a lot of research on Bitcoin. So we decided to choose Ether which has the greatest potential and a moderate birth time as the research object.

Goal of this project

We only keep focus on the trading function of Ether as a cryptocurrency. People would make stock-like investments by holding and buying and selling Ether. As an investment asset, people always hope to maximize the benefits. To achieve this goal, our usual strategy is to buy low and sell high. As the market fluctuates and holdings change, so does the price of Ether. Generally, the price of a product is affected by the price of similar products in the market. Our goal is to use top 10 Cryptocurrencies to predict the price of Ether to see how it will be affected. This will provide some reference for people during making investments.

Loading Data and Packages

Loading packages:

The packages have been loaded at the beginning of the Rmarkdown file. In these packages, some of them are used, while some are not. We only use the packages like tidyverse, tidymodels, ggplot2, randomForest, etc. It is my personal preference to library all the packages for such related study.

Introduction of Data:

The data set is acquired from Kaggle called Cryptocurrency Historical Prices (link here). It is a dataset that includes the historical prices information of 23 Cryptocurrencies by market capitalization. We choose 11 (Ether, Binance coin, Bitcoin, Cardano, Dogecoin, Litecoin, Monero, Stellar, Tether, USD coin, and XRP) of 23 Cryptocurrencies in csv files. Each of the observation is the trading history on a daily basis, starting from their birth time to July 6, 2021.

Now, we need to process these raw files into one useful file for our project. Each *original* csv file provides 10 variables:

1.`SNo` : Index of each observation

2.`Name` : Name of each kind of Cryptocurrency

3.`Symbol` : Symbol represented each kind of Cryptocurrency

4.`Date` : Date of observation

5.`Open` : Opening price on the given day

6.`High` : Highest price on the given day

7.`Low` : Lowest price on the given day

8.`Close` : Closing price on the given day

9.`Volume` : Volume of transactions on the given day

10.`Market Cap` : Market capitalization in USD

Notice: The variables here are from the original data, not what we will use in our model.

Loading data

We will use symbols to represent each cryptocurrency.

```
ETH_raw <- read.csv("data/unprocessed/coin_Ethereum.csv")
BTC_raw <- read.csv("data/unprocessed/coin_Bitcoin.csv")
ADA_raw <- read.csv("data/unprocessed/coin_Cardano.csv")
DOGE_raw <- read.csv("data/unprocessed/coin_Dogecoin.csv")
LTC_raw <- read.csv("data/unprocessed/coin_Litecoin.csv")
XLM_raw <- read.csv("data/unprocessed/coin_Stellar.csv")
BNB_raw <- read.csv("data/unprocessed/coin_BinanceCoin.csv")
USDT_raw <- read.csv("data/unprocessed/coin_Tether.csv")
XMR_raw <- read.csv("data/unprocessed/coin_Monero.csv")
USDC_raw <- read.csv("data/unprocessed/coin_USDCoin.csv")
XRP_raw <- read.csv("data/unprocessed/coin_XRP.csv")
```

Since different cryptocurrencies have different birth times, in order to avoid missing values, we choose the time period when these eleven currencies exist. Because their data is as of July 6, 2021, and is based on a daily basis. So we just need to find the cryptocurrency with the smallest number of observations and choose the appropriate time period based on it.

Find Minimum observations

Min observations:

```
min(c(nrow(ETH_raw ), #Find Minimum observations
      nrow(BTC_raw ),
      nrow(ADA_raw ),
      nrow(DOGE_raw),
      nrow(LTC_raw ),
      nrow(XLM_raw ),
      nrow(BNB_raw ),
      nrow(USDT_raw),
      nrow(XMR_raw ),
      nrow(USDC_raw),
      nrow(XRP_raw )))
```

```
## [1] 1002
```

```
#Extract the last 1002 observations from each data set
ETH_raw <- tail(ETH_raw ,n=1002)
BTC_raw <- tail(BTC_raw ,n=1002)
ADA_raw <- tail(ADA_raw ,n=1002)
DOGE_raw <-tail(DOGE_raw,n=1002)
LTC_raw <- tail(LTC_raw ,n=1002)
XLM_raw <- tail(XLM_raw ,n=1002)
BNB_raw <- tail(BNB_raw ,n=1002)
USDT_raw <-tail(USDT_raw,n=1002)
XMR_raw <- tail(XMR_raw ,n=1002)
USDC_raw <-tail(USDC_raw,n=1002)
XRP_raw <- tail(XRP_raw ,n=1002)
```

To make the price more feasible and representative, we choose to use the mean price of a day calculated by `High` and `Low` of each cryptocurrency.

```
ETH_raw <- ETH_raw  %>% mutate(price = (High+Low)/2)
BTC_raw <- BTC_raw  %>% mutate(price = (High+Low)/2)
ADA_raw <- ADA_raw  %>% mutate(price = (High+Low)/2)
DOGE_raw <-DOGE_raw %>% mutate(price = (High+Low)/2)
LTC_raw <- LTC_raw  %>% mutate(price = (High+Low)/2)
XLM_raw <- XLM_raw  %>% mutate(price = (High+Low)/2)
BNB_raw <- BNB_raw  %>% mutate(price = (High+Low)/2)
USDT_raw <-USDT_raw %>% mutate(price = (High+Low)/2)
XMR_raw <- XMR_raw  %>% mutate(price = (High+Low)/2)
USDC_raw <-USDC_raw %>% mutate(price = (High+Low)/2)
XRP_raw <- XRP_raw  %>% mutate(price = (High+Low)/2)
```

Now, we extract the date and mean price column from each data frame and binds them into a new data set `coins`.

```
coins <- cbind(ETH_raw$Date  ,
               ETH_raw$price ,
               BTC_raw$price ,
               ADA_raw$price ,
               DOGE_raw$price,
               LTC_raw$price ,
               XLM_raw$price ,
               BNB_raw$price ,
               USDT_raw$price,
               XMR_raw$price ,
               USDC_raw$price,
               XRP_raw$price )
coins <- as.data.frame(coins)
colnames(coins) <- c("Date",
                     "ETH",
                     "BTC",
                     "ADA",
                     "DOGE",
                     "LTC",
                     "XLM",
                     "BNB",
                     "USDT",
                     "XMR",
                     "USDC",
                     "XRP")
coins[2:12] <- lapply(coins[2:12], FUN = function(y){as.numeric(y)})
coins[1] <- lapply(coins[1], FUN = function(y){as.character(y)})
head(coins)
```

```
##                  Date      ETH      BTC        ADA        DOGE      LTC
## 1 2018-10-09 23:59:59 228.2810 6634.175 0.08598600 0.005606450 58.87045
## 2 2018-10-10 23:59:59 226.1010 6589.625 0.08498715 0.005589395 58.04490
## 3 2018-10-11 23:59:59 207.4465 6415.240 0.07728895 0.005224085 54.48380
## 4 2018-10-12 23:59:59 194.0540 6282.485 0.07285930 0.005120310 52.34805
## 5 2018-10-13 23:59:59 198.8215 6284.160 0.07368025 0.005217365 53.74590
## 6 2018-10-14 23:59:59 198.5350 6321.680 0.07224475 0.005183515 52.93545
##         XLM       BNB      USDT       XMR     USDC       XRP
## 1 0.2465645 10.400250 0.9958625 113.88650 1.004180 0.4811245
## 2 0.2420040 10.303550 0.9963165 112.34950 1.011525 0.4690965
## 3 0.2237595  9.781660 1.0024010 107.27650 1.016480 0.4215555
## 4 0.2122380  9.449160 0.9911870 101.65905 1.009925 0.4092290
## 5 0.2169750  9.552435 0.9901975 101.83300 1.011920 0.4200695
## 6 0.2140520  9.553700 0.9900570 102.08300 1.013715 0.4132260
```

Finally, we have the data set `coins` for our project.

There is a full copy of the codebook available in the zipped files. Here is a list of key variables in `coins` that are helpful for understanding this project:

There are 12 variables:

1.`Date(date)` : Date of observation

2.`ETH(eth)` : Mean price of Ethereum of each day

3.`BTC(btc)` : Mean price of Bitcoin of each day

4.`ADA(ada)` : Mean price of Cardano of each day

5.`DOGE(doge)` : Mean price of Doge Coin of each day

6.`LTC(ltc)` : Mean price of Litecoin of each day

7.`XLM(xlm)` : Mean price of Stellar of each day

8.`BNB(bnb)` : Mean price of Binance Coin of each day

9.`USDT(usdt)` : Mean price of Tether of each day

10.`XMR(xmr)` : Mean price of Monero of each day

11.`USDC(usdc)` : Mean price of USD Coin of each day

12.`XRP(xrp)` : Mean price of XRP of each day

Check the dimension

```
dim(coins)
```

```
## [1] 1002   12
```

Check missing value

```
sum(is.na(coins))
```

```
## [1] 0
```

According to the output above, this data set includes 12 columns and 1002 observations. There are 1 character variable (Date) and 11 numeric variables (different cryptocurrencies). There is no missing value in our data set which is good for processing next steps.

Data Cleaning

Since our data is relatively tidy, We just need to apply a few more steps to make it more fittable for the project.

- Clean names

```
coins <- coins %>%
  clean_names()
```

- Parse date

```
coins <- coins %>%
  mutate(
    date = parse_date(date, "%Y-%m-%d %H:%M:%S")
  )
```

Now the data has been cleaned up and ready to process next steps.

Exploratory Data Analysis

The entire exploratory data analysis will be based on the whole set. There are 1002 observations and each of them represents a single-day mean price history.

In order to get more descriptive characters of the data set, we summary the data.

```
summary(coins)
```

```
##       date                 eth                btc               ada
##  Min.   :2018-10-09   Min.   :  84.086   Min.   : 3233.3   Min.   :0.024179
##  1st Qu.:2019-06-16   1st Qu.: 166.871   1st Qu.: 7012.0   1st Qu.:0.043970
##  Median :2020-02-21   Median : 221.375   Median : 9409.0   Median :0.074607
##  Mean   :2020-02-21   Mean   : 575.911   Mean   :15627.4   Mean   :0.266469
##  3rd Qu.:2020-10-28   3rd Qu.: 423.090   3rd Qu.:13391.6   3rd Qu.:0.133240
##  Max.   :2021-07-06   Max.   :4074.100   Max.   :63208.9   Max.   :2.237526
##       doge                ltc               xlm                bnb
##  Min.   :0.0015231   Min.   : 23.290   Min.   :0.034801   Min.   :  4.5591
##  1st Qu.:0.0023649   1st Qu.: 45.794   1st Qu.:0.069783   1st Qu.: 15.5080
##  Median :0.0027059   Median : 59.418   Median :0.095949   Median : 21.1720
##  Mean   :0.0354744   Mean   : 85.901   Mean   :0.153783   Mean   : 71.0326
##  3rd Qu.:0.0035356   3rd Qu.:101.351   3rd Qu.:0.168046   3rd Qu.: 31.2835
##  Max.   :0.6728672   Max.   :376.050   Max.   :0.729991   Max.   :654.8469
##       usdt              xmr               usdc              xrp
##  Min.   :0.95918   Min.   : 33.322   Min.   :0.97766   Min.   :0.13993
##  1st Qu.:1.00025   1st Qu.: 58.390   1st Qu.:1.00012   1st Qu.:0.24086
##  Median :1.00152   Median : 79.222   Median :1.00163   Median :0.30106
##  Mean   :1.00269   Mean   :108.041   Mean   :1.00441   Mean   :0.38351
##  3rd Qu.:1.00531   3rd Qu.:120.561   3rd Qu.:1.00722   3rd Qu.:0.41969
##  Max.   :1.03231   Max.   :470.101   Max.   :1.05605   Max.   :1.77204
```

From the result above, we can see the most valuable is `btc` whose maximum reach $63208.9, and `eth` whose maximum is $4074.100. All others are relatively low, and some of them are always around $1. It is really interesting that in about 3 years, the price of `eth` has increased by about 50 times, and `btc` has increased about 20 times. No wonder so many people are investing in them.

Response Variable: The Price of Ether

First, we can calculate the mean and variance of Ether price.

Mean:

```
mean(coins$eth)
```

```
## [1] 575.9111
```

Variance:

```
var(coins$eth)
```

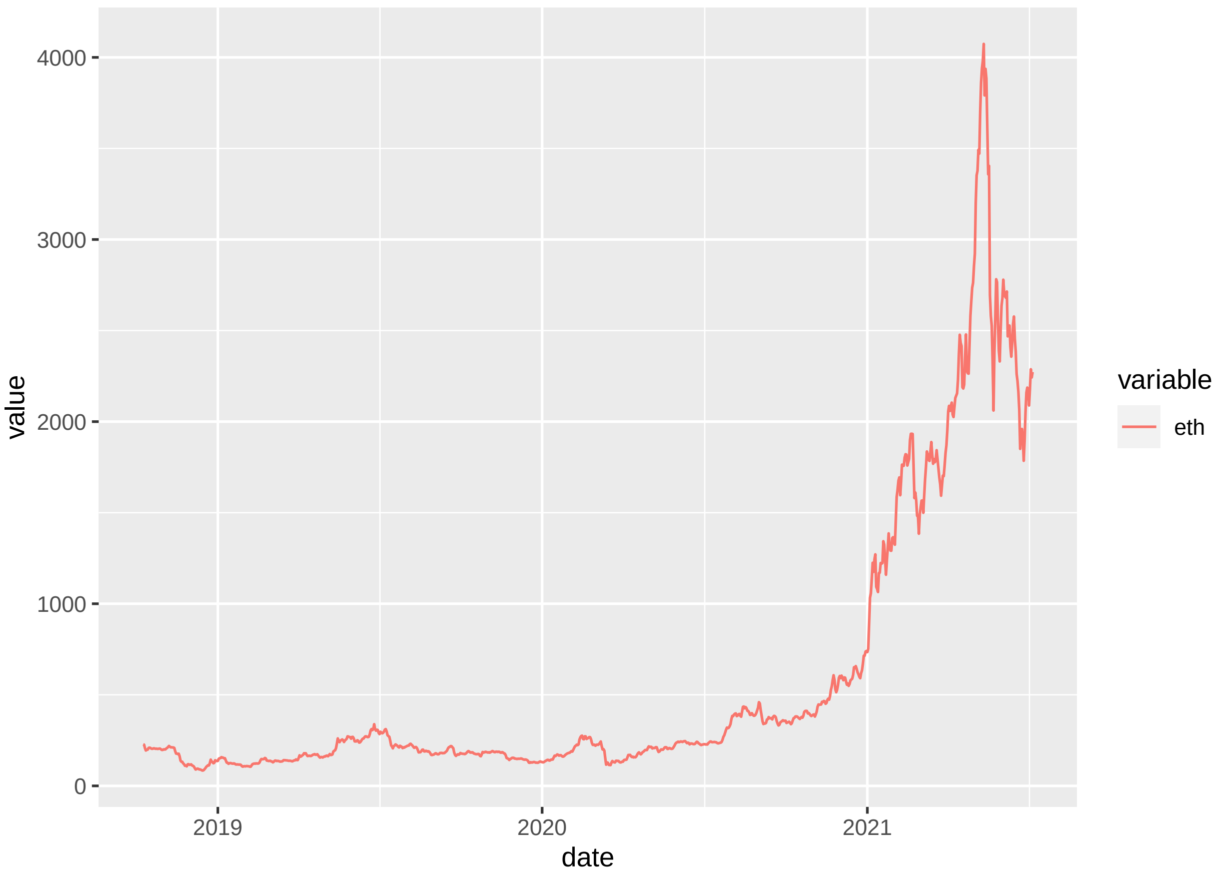
```
## [1] 605712.88
```

We can see the variance is quite large, which indicates that the data points are very spread out from the mean.

Now, we can plot the `eth` to explore the distribution.

First we plot `eth` with time to see the overall trend.

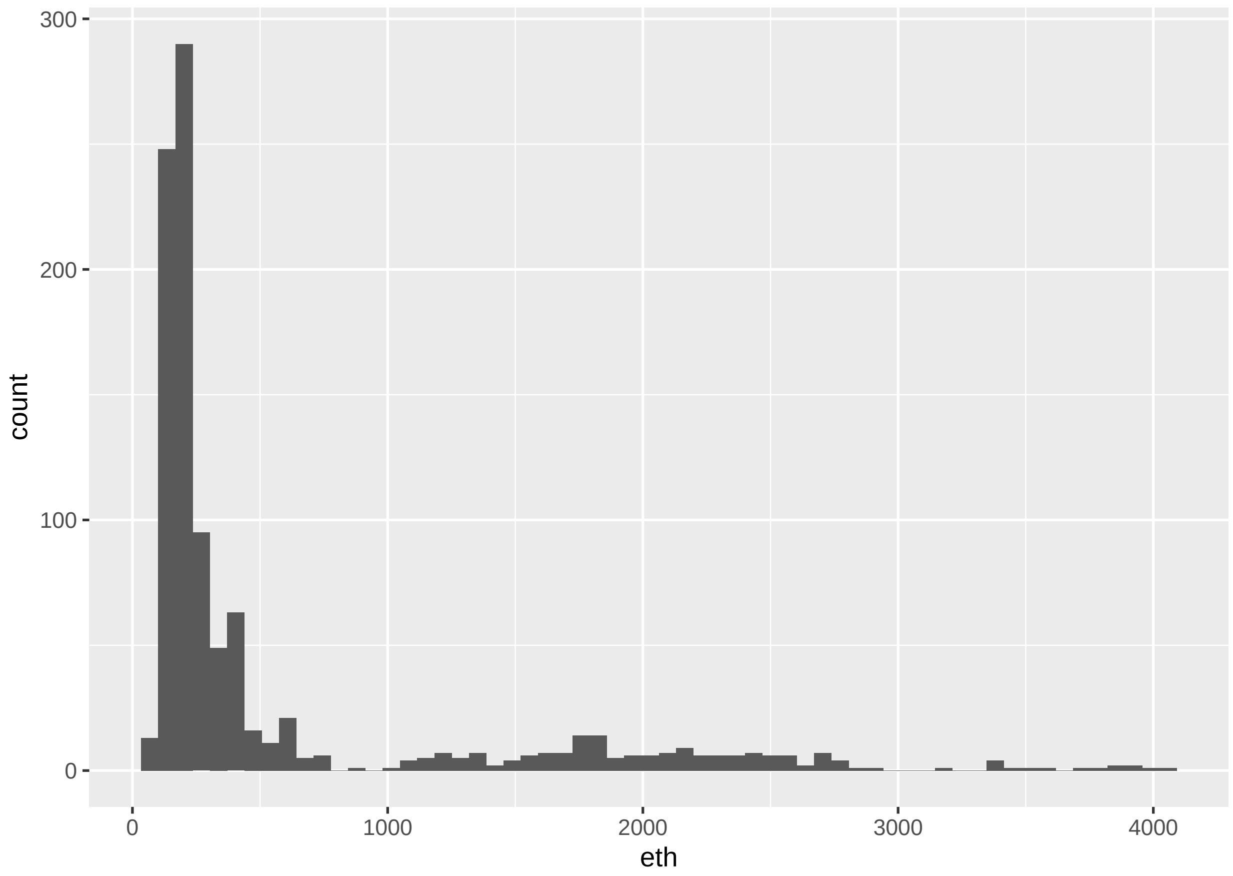
```
data_short <- melt(coins %>% select(date,eth), id = "date")
data_short %>%
  ggplot(aes(x = date,
             y = value,
             color = variable))+
  geom_line()
```



We can find that over time, the price is trending upwards in general. The closing price gradually increased from 2020 and then increased rapidly in 2021, reaching an overall high in the middle of the year, before beginning to fall back.

Then we will assess the distribution of our response variable `etc`, so let's plot a histogram for it.

```
coins %>%
  ggplot(aes(eth)) +
  geom_histogram(bins = 60)
```



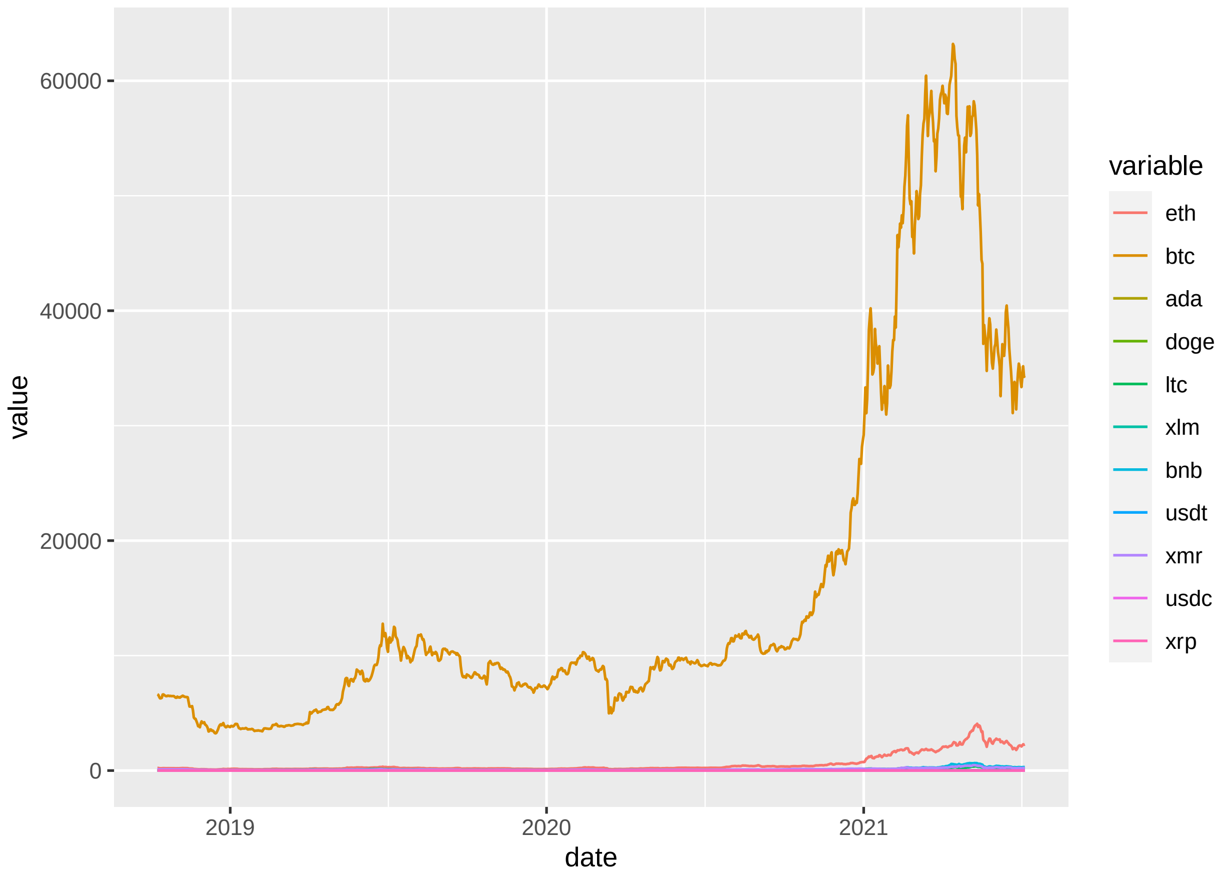
From the histogram, we notice `etc` is positively skewed, meaning that much of the mass of its distribution is at the lower end, with a long tail to the right. Most observations in the data set are less than $1000.

Predicted Variables

Now we keep focus on our predicted variables.

First we plot all the variables with time to check if there any difference or similarity in trend with `eth`.

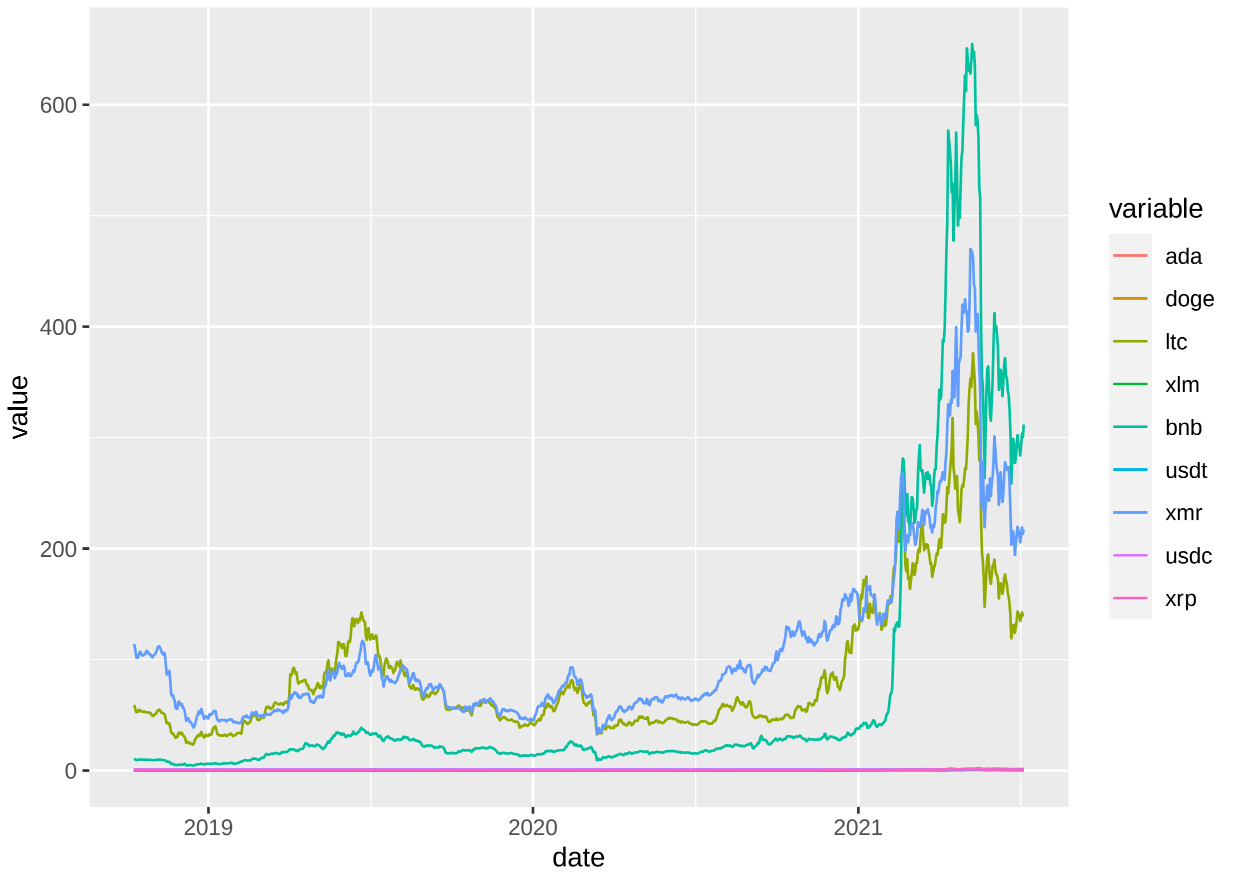
```
data_long <- melt(coins, id = "date")
data_long %>%
  ggplot(aes(x = date,
             y = value,
             color = variable))+
  geom_line()
```



From the plot, we notice that the trend of btc is similar to the trend of eth. However, the value of btc and eth are too large comparing with other variables. We cannot see the relationship between other variables clearly.

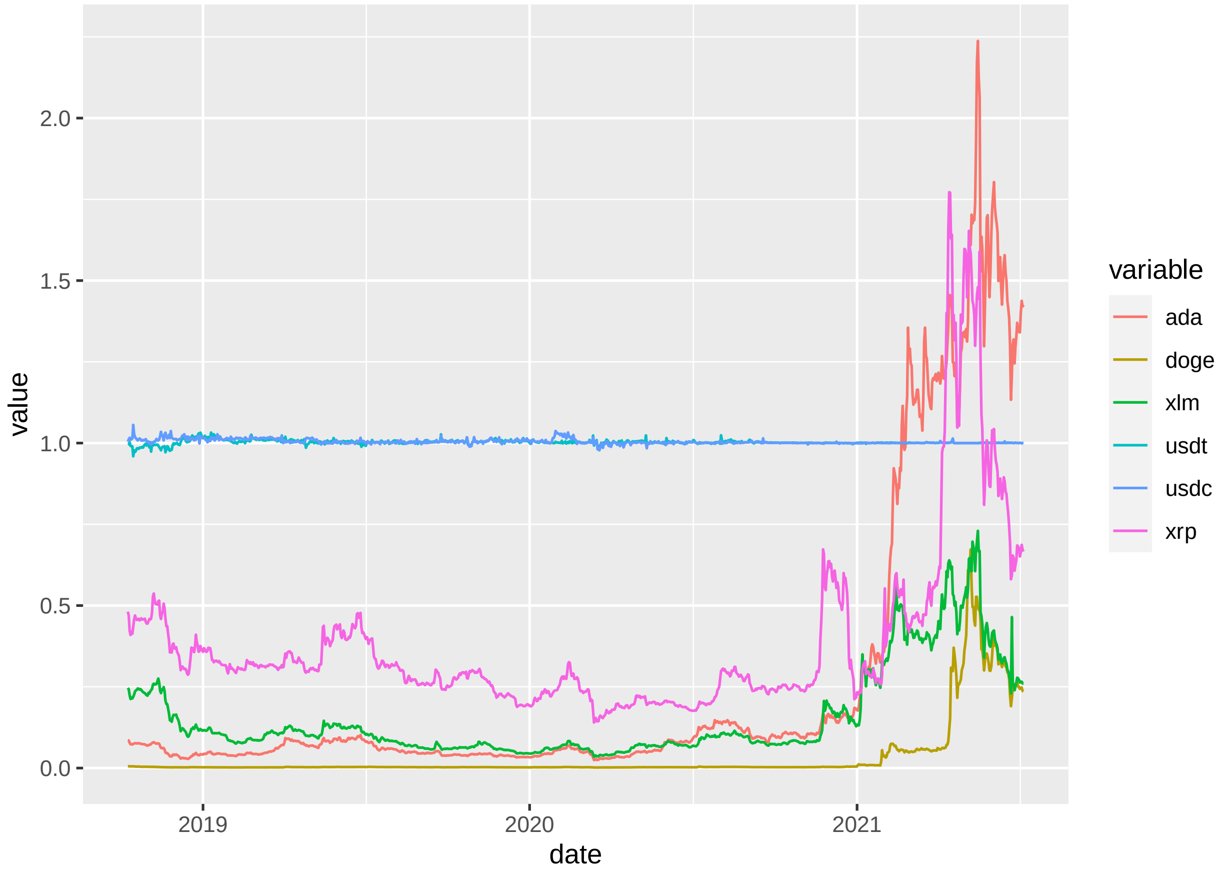
After removing variables btc and eth, we plot it again.

```
data_long2 <- melt(coins %>% select(-btc,-eth), id = "date")

data_long2 %>%
  ggplot(aes(x = date,
             y = value,
             color = variable))+
  geom_line()
```



We found the `ltc`, `xmr`, and `bnb` also have the same trend as `eth` in general. They have a small peak at the middle of 2019 like `btc`. But these three variables are still too large for others. Then we remove them and replot others.

```
data_long3 <- melt(coins %>% select(-btc,-eth, -ltc, -xmr, -bnb), id = "date")

data_long3 %>%
  ggplot(aes(x = date,
             y = value,
             color = variable))+
  geom_line()
```

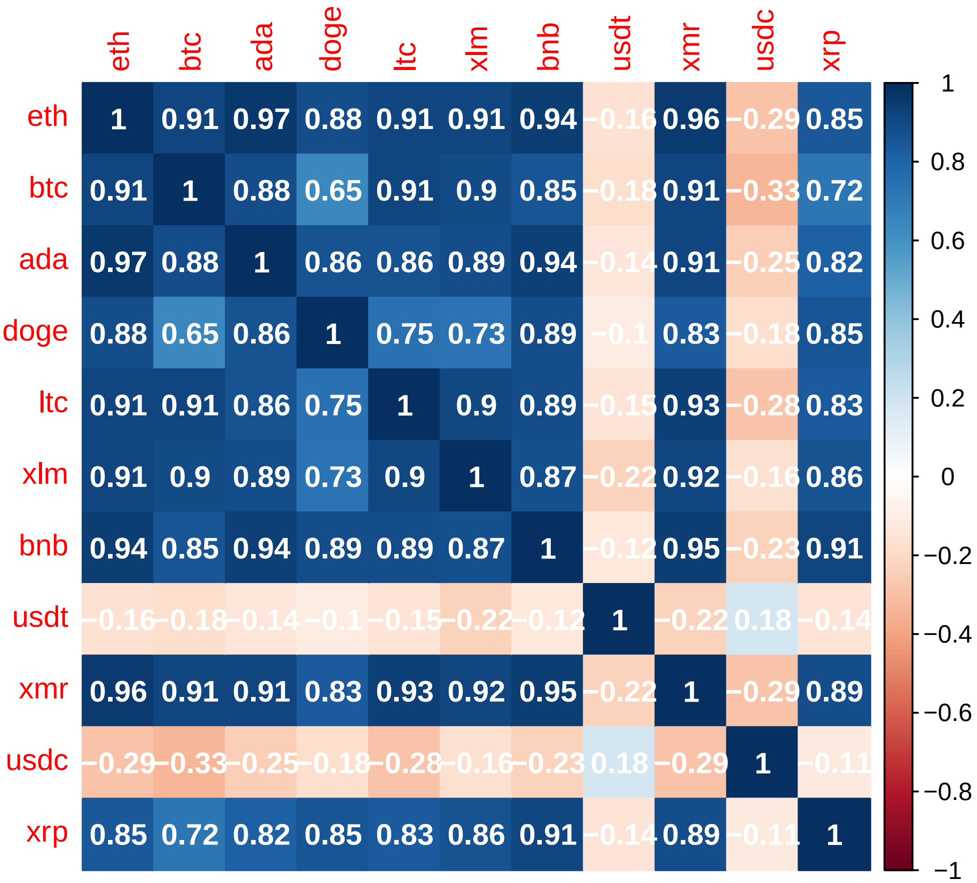


Here are some interesting patterns. The `usdt` and `usdc` are very stable, keeping around $1. But it makes sense to me, because those are famous of being stable and nearly 1:1 to USD. People are always trading other cryptocurrencies with them instead of USD. Other 4 cryptocurrcies are generally the same trend as `eth`, but they have slight difference between the `eth`.

Now we can see the trend of all predicted variables are really similar to the variable `eth`, which means they may have strong correlation with each other.

To explore more on the correlation, we can construct a correlation plot.

```
coins %>%
  select(where(is.numeric)) %>%
  cor() %>%
  corrplot(type = "full", diag = TRUE,
           method = "color",addCoef.col = 'white')
```



By plotting the correlation matrix, we can have a better understand on the structure of the data set. The color blue indicates that the variables are positively correlated, while the color red indicates the variables are negatively correlated. And white means they are not correlated to each other.

From the plot, we notice that `usdt` and `usdc` are weakly negatively correlated to other variables. That can be explained as above that they are stable and keep 1:1 to USD. People use them to trade other cryptocurrencies. So when their price drop, other price increase.

Except `usdt` and `usdc`, all other variables are strongly correlated. Among these correlations, `ada` is strongly correlated to `eth`. So are the `xmr` and `eth`. Also, `xmr` is strongly correlated to `bnb`.

Data Splitting

Since we only have one data set, we want to estimate the generalization performance after we finish modeling part. If we use the same data when we train and estimate the performance, then we will have a much more optimistic result when estimate. Then, there will be a lack of accuracy.

The data was split into 80% for training and 20% for testing. And we use stratified sampling on the `eth` variable.

```
coins_split <- coins %>%
  initial_split(prop = 0.80, strata = "eth")

coins_train <- training(coins_split)
coins_test <- testing(coins_split)
```

Check the dimension of training and testing set

```
dim(coins_train)
```

```
## [1] 800  12
```

```
dim(coins_test)
```

```
## [1] 202  12
```

```
800/(800+202)
```

```
## [1] 0.79840319
```

The training data set has 800 observations and the testing data set has 202 observations. And the proportion is 0.79840319, which indicates the number of observation is correct.

Model Fitting

Fold and Recipe

Fold the training data into 10 folds with 5 repeats

```
coins_fold <- vfold_cv(coins_train, v = 10, repeats = 5)
```

Create a recipe

```
coins_recipe <- recipe(formula = eth ~ btc + ada +
                         doge + ltc + xlm + bnb +
                         usdt + xmr + usdc + xrp,
                       data = coins) %>%
  step_center(all_predictors()) %>%
  step_scale(all_predictors())
# Since we don't have any factor variables, we don't need
# to use step_dummy() and others. All we need to do is
# center and scale all the predictors to normalized them.
```

Ridge Regression

First, we try to use Ridge regression as our model.

In Ridge regression, there is an essential parameter `penalty` which will affect our model. We can fit a Ridge model with different values of `penalty`, but it would be nice if we can find the best value of `penalty`.

Here we use a method called hyperparameter tuning to help us decide the best `penalty`. It is a way of fitting many models with different sets of hyperparameters trying to find one best. The complexity in hyperparameter tuning can come from the number of different models tried. Here we only look at grid search, only looking at evenly spaced parameter values.

Model specification

```
ridge_spec <- linear_reg(penalty = tune(), mixture = 0) %>%
  set_mode("regression") %>%
  set_engine("glmnet")
  # Tells tune_grid() that penalty parameter should be tuned
  # mixture set to 0, only use Ridge
  # Set the model to regression and use glmnet engine
```

Model workflow

Now we combine to create a `workflow` object.

```
ridge_wkflow <- workflow() %>%
  add_recipe(coins_recipe) %>%
  add_model(ridge_spec)
```

Penalty range

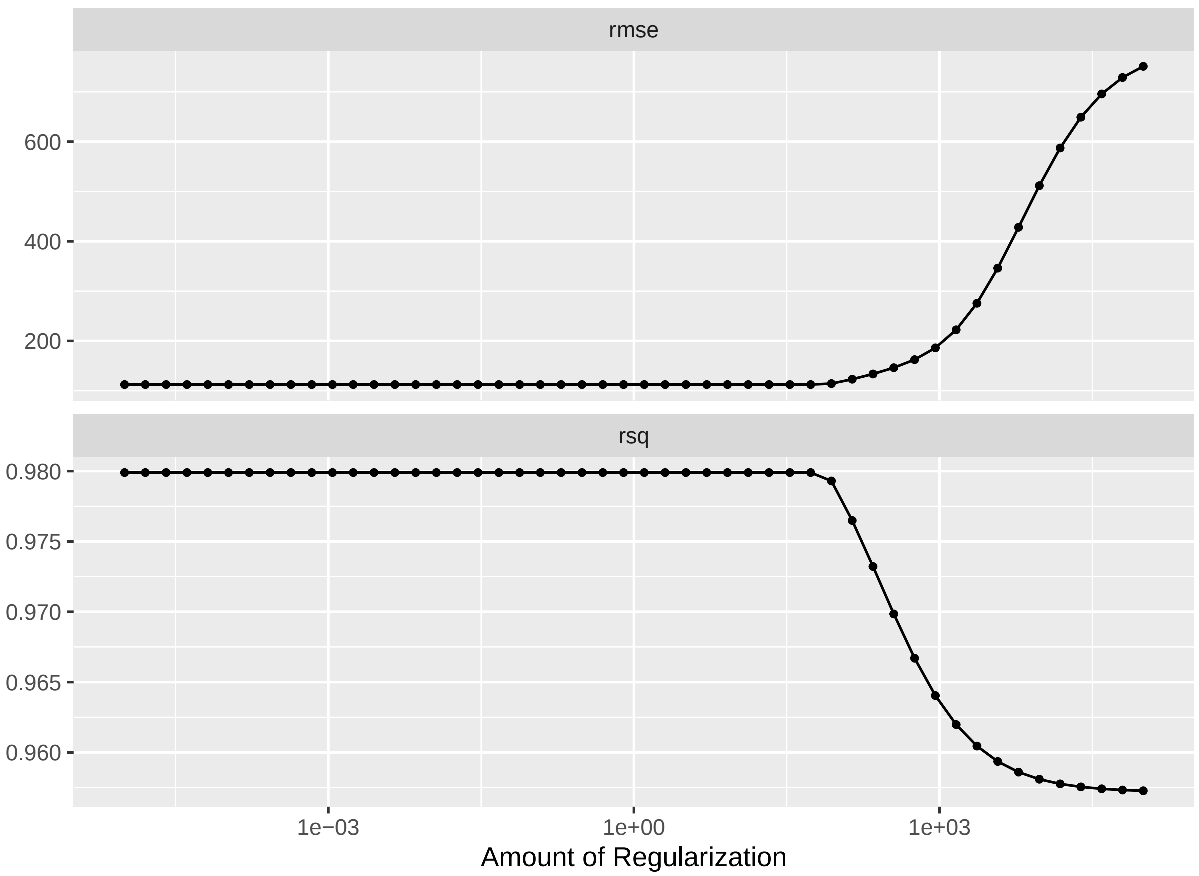
The last thing is to set an appropriate range for our `penalty`. We can use `grid_regular()` which creates a grid of evenly spaced parameter values. We choose the range from -5 to 5 and divided in to 50 levels.

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

Ridge regression model fitting

We have enerything we need for Ridge regression and now we can fit all the models.

```
ridge_res <- tune_grid(
  ridge_wkflow,
  resamples = coins_fold,
  grid = penalty_grid
)

autoplot(ridge_res)
```



From plot, we can see that the amount of regularization affects the performance metrics differently.

We can also check the raw metrics by `collect_metrics()`.

```
head(collect_metrics(ridge_res))
```

```
## # A tibble: 6 x 7
##     penalty .metric .estimator    mean     n std_err .config
##       <dbl> <chr>   <chr>        <dbl> <int>   <dbl> <chr>
## 1 0.00001   rmse    standard   112.       50 2.85    Preprocessor1_Model01
## 2 0.00001   rsq     standard     0.980    50 0.00141 Preprocessor1_Model01
## 3 0.0000160 rmse    standard   112.       50 2.85    Preprocessor1_Model02
## 4 0.0000160 rsq     standard     0.980    50 0.00141 Preprocessor1_Model02
## 5 0.0000256 rmse    standard   112.       50 2.85    Preprocessor1_Model03
## 6 0.0000256 rsq     standard     0.980    50 0.00141 Preprocessor1_Model03
```

Now, we can use `select_best()` to select the best value.

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

```
## # A tibble: 1 x 2
##   penalty .config
##     <dbl> <chr>
## 1 0.00001 Preprocessor1_Model01
```

Then we can finalize our model with this value of penalty. The best model should be fit again with the whole training data set.

```
ridge_final <- finalize_workflow(ridge_wkflow, best_penalty)

ridge_final_fit <- fit(ridge_final, data = coins_train)
```

Apply the final model on the whole **training** set and validate its performance.

```
final_ridge <- augment(ridge_final_fit, new_data = coins_train)
final_ridge %>% rmse(truth = eth, estimate = .pred)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard        112.
```

The RMSE (root mean squared error) tells us how far apart the predicted values are from the observed values in a dataset, on average. The lower the RMSE, the better a model fits a dataset. We find that the RMSE for our Ridge model is **112.09605**. Next we can explore other models to check their RMSE value and find the model with the lowest value of RMSE as our final model.

K-Nearest Neighbor Method

Our second model is the KNN model. The process is similar to the above. For nearest neighbor, we only tuned `neighbors` as the model's other defaults are fine. Then set the workflow and add the recipe again.

Model specification and workflow

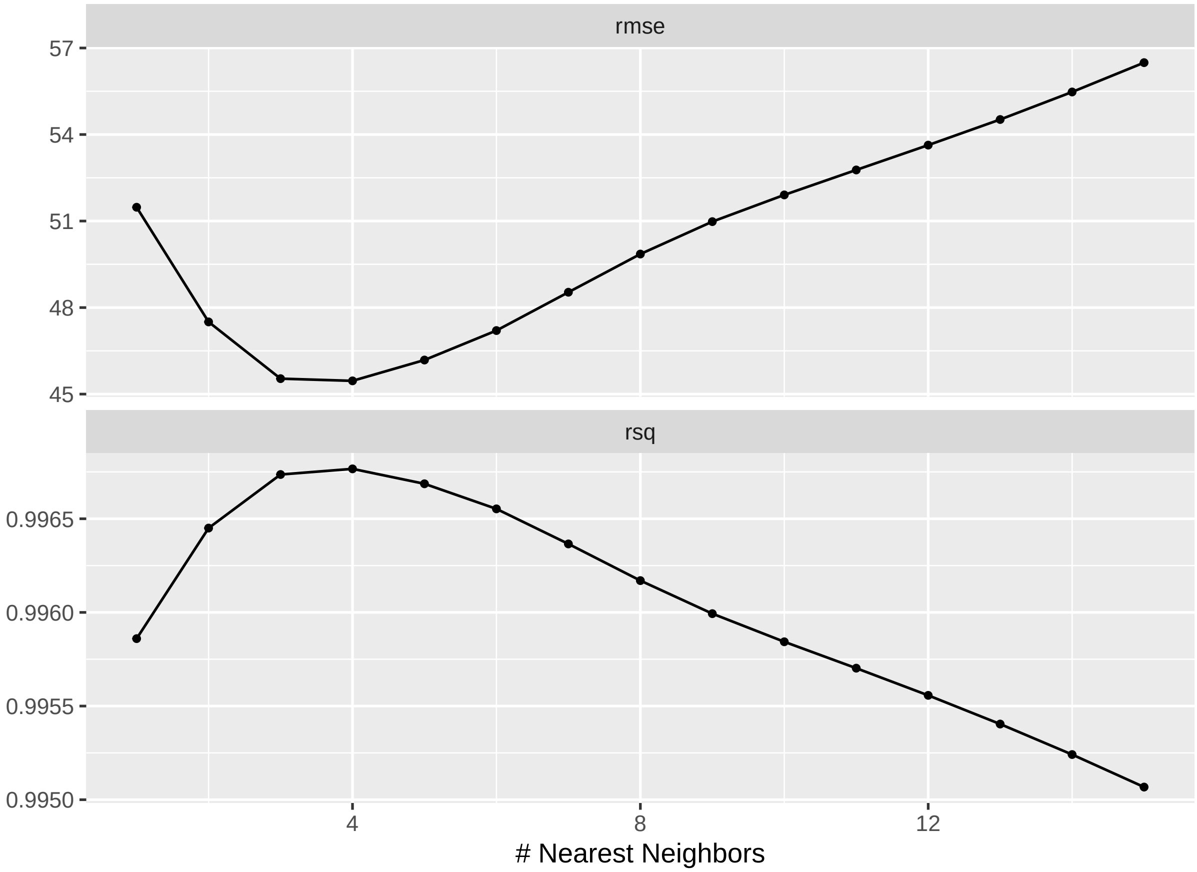
```
knn_model <- nearest_neighbor(
  neighbors = tune(),
  mode = "regression",
  engine = "kknn")

knn_wkflow <- workflow() %>%
  add_model(knn_model) %>%
  add_recipe(coins_recipe)
```

Tuning grid

```
# set-up tuning grid
knn_params <- extract_parameter_set_dials(knn_model)


# define grid
knn_grid <- grid_regular(knn_params, levels = 50)
```

KNN Model fitting

```
knn_tune <- knn_wkflow %>%
  tune_grid(
    resamples = coins_fold,
    grid = knn_grid
  )

autoplot(knn_tune)
```



Now check the raw metrics.

```
head(collect_metrics(knn_tune))
```

```
## # A tibble: 6 x 7
##   neighbors .metric .estimator   mean     n  std_err .config
##       <int> <chr>   <chr>       <dbl> <int>    <dbl> <chr>
## 1         1 rmse    standard   51.5      50 1.58     Preprocessor1_Model01
## 2         1 rsq     standard    0.996    50 0.000251 Preprocessor1_Model01
## 3         2 rmse    standard   47.5      50 1.39     Preprocessor1_Model02
## 4         2 rsq     standard    0.996    50 0.000205 Preprocessor1_Model02
## 5         3 rmse    standard   45.5      50 1.23     Preprocessor1_Model03
## 6         3 rsq     standard    0.997    50 0.000183 Preprocessor1_Model03
```

Select the best neighbors.

```
best_neighbors <- select_best(knn_tune, metric = "rmse")
best_neighbors
```

```
## # A tibble: 1 x 2
##   neighbors .config
##       <int> <chr>
## 1         4 Preprocessor1_Model04
```

Then finalize our KNN model. The best model should be fit again with the whole training data set.

```
knn_final <- finalize_workflow(knn_wkflow, best_neighbors)

knn_final_fit <- fit(knn_final, data = coins_train)
```

Apply the final model on the whole **training** set and validate its performance.

```
final_knn <- augment(knn_final_fit, new_data = coins_train)
final_knn %>% rmse(truth = eth, estimate = .pred)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard        20.7
```

The RMSE of our KNN model is only **20.697688**. It means our model fits pretty well, and it is quite smaller than the Ridge regression model which is **112.09605**.

Random Forest Model

The third model we use is random forest model. Same as before, we need to set up model and workflow.

Model specification and workflow

In random forest model, we need to tune `min_n`, `tres`, and `mtry`. Set `mode` to `"regression"` and use `randomForest` engine.

```
rf_model <- rand_forest(min_n = tune(),
                        mtry = tune(),
                        trees = tune(),
                        mode = "regression") %>%
  set_engine("randomForest", importance = TRUE)

rf_wkflow <- workflow() %>%
  add_model(rf_model) %>%
  add_recipe(coins_recipe)
```

Tuning grid

Here we set `mtry` in the range from 1 to 10, because it represents the number of predictors and we only have 10 predictors. So if `mtry = 10`, then the model uses all 8 predictors.

```
rf_grid <- grid_regular(
  mtry(range = c(1,10)),
  trees(range = c(10,300)),
  min_n(range = c(1,5)),
  levels = 8
)
```

Random Forest Model fitting

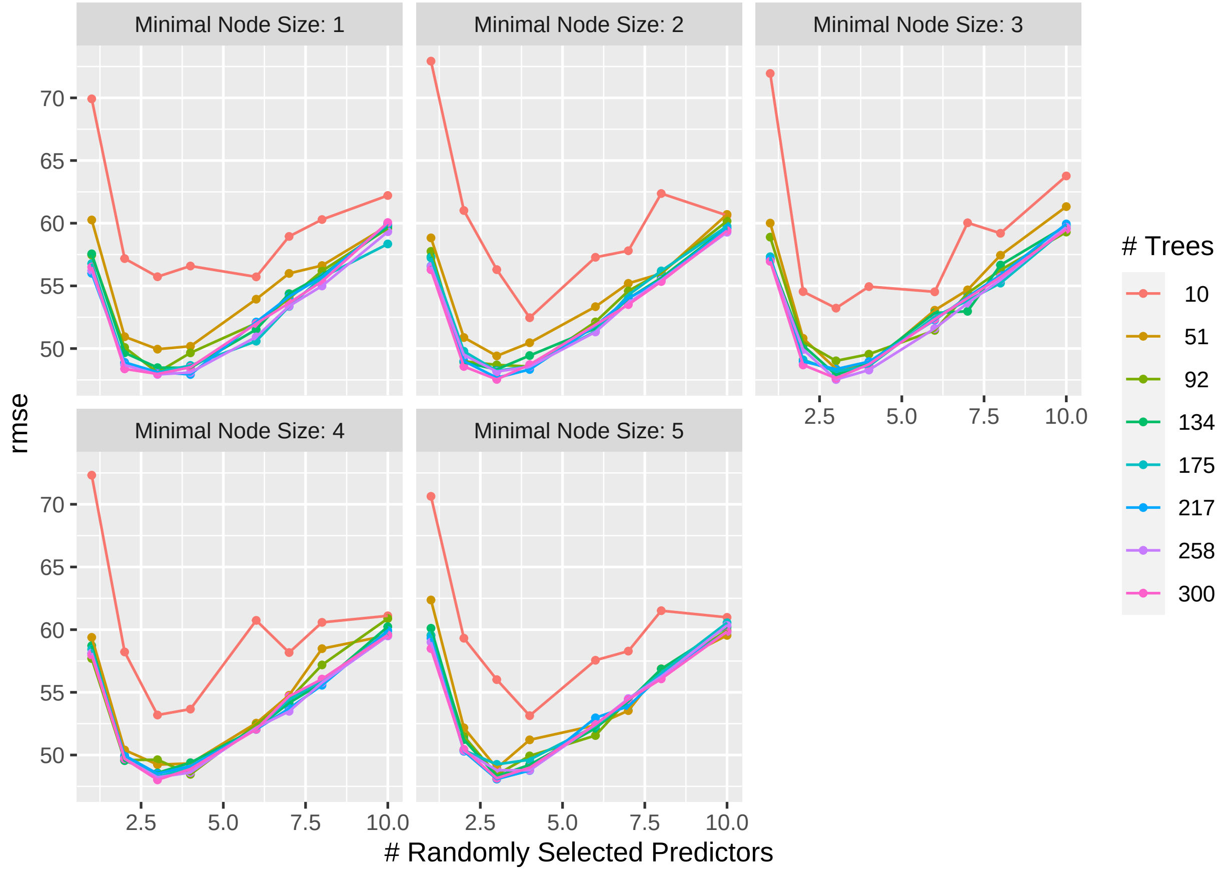
```
rf_tune <- tune_grid(
  rf_wkflow,
  resamples = coins_fold,
  grid = rf_grid,
  metrics = metric_set(rmse)
)

autoplot(rf_tune)
```

Here we save the `rf_tune` into `rda` file and load it, so that it will not take too much time to knit the Rmarkdown.

```
#save(rf_tune, file = "temp_rda/rf_tune.rda")

load(file = "temp_rda/rf_tune.rda")
```

Plot the `rf_tune`.

```
autoplot(rf_tune)
```



Now check the lowest RMSE.

```
head(collect_metrics(rf_tune) %>% arrange(mean))
```

```
## # A tibble: 6 x 9
##    mtry trees min_n .metric .estimator  mean     n std_err .config
##   <int> <int> <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1     3   258     3 rmse    standard    47.5    50    1.98 Preprocessor1_Model1~
## 2     3   300     2 rmse    standard    47.5    50    1.97 Preprocessor1_Model1~
## 3     3   300     3 rmse    standard    47.6    50    2.00 Preprocessor1_Model1~
## 4     3   217     2 rmse    standard    47.7    50    2.04 Preprocessor1_Model1~
## 5     3   134     3 rmse    standard    47.9    50    2.00 Preprocessor1_Model1~
## 6     4   217     1 rmse    standard    47.9    50    2.08 Preprocessor1_Model0~
```

The lowest `RMSE` is the model with `mtry = 3`, `trees = 258`, and `min_n = 3`. The `RMSE` value is 47.530856.

Select the best parameters.

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

```
## # A tibble: 1 x 4
##    mtry trees min_n .config
##   <int> <int> <int> <chr>
## 1     3   258     3 Preprocessor1_Model179
```

Then finalize our random forest model. The best model should be fit again with the whole training data set.

```
rf_final <- finalize_workflow(rf_wkflow, best_rf)

rf_final_fit <- fit(rf_final, data = coins_train)
```

Apply the final model on the whole **training** set and validate its performance.

```
final_rf <- augment(rf_final_fit, new_data = coins_train)
final_rf %>% rmse(truth = eth, estimate = .pred)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard        18.9
```

It is surprised that the value of RMSE of random forest model is only **18.93514**, which is slightly better than the KNN model.

Boosted Trees Model

In a similar process, we need to set up the specification and the workflow.

Model specification and workflow

In boosted tree model, we choose to tune `trees` and `learning rate`. Set `mode` to `"regression"` and use `xgboost` engine.

```
bt_model <- boost_tree(mode = "regression",
                       trees = tune(),
                       learn_rate = tune()) %>%
  set_engine("xgboost")

bt_wkflow <- workflow() %>%
  add_model(bt_model) %>%
  add_recipe(coins_recipe)
```

Tuning grid

Here we set `trees` in the range from 10 to 1000 and `learning rate` between 0.1 and 0.3, with using 10 levels.

```
bt_grid <- grid_regular(trees(c(10,1000)),learn_rate(c(0.1,0.3)), levels = 10)
```

Boosted Tree Model fitting

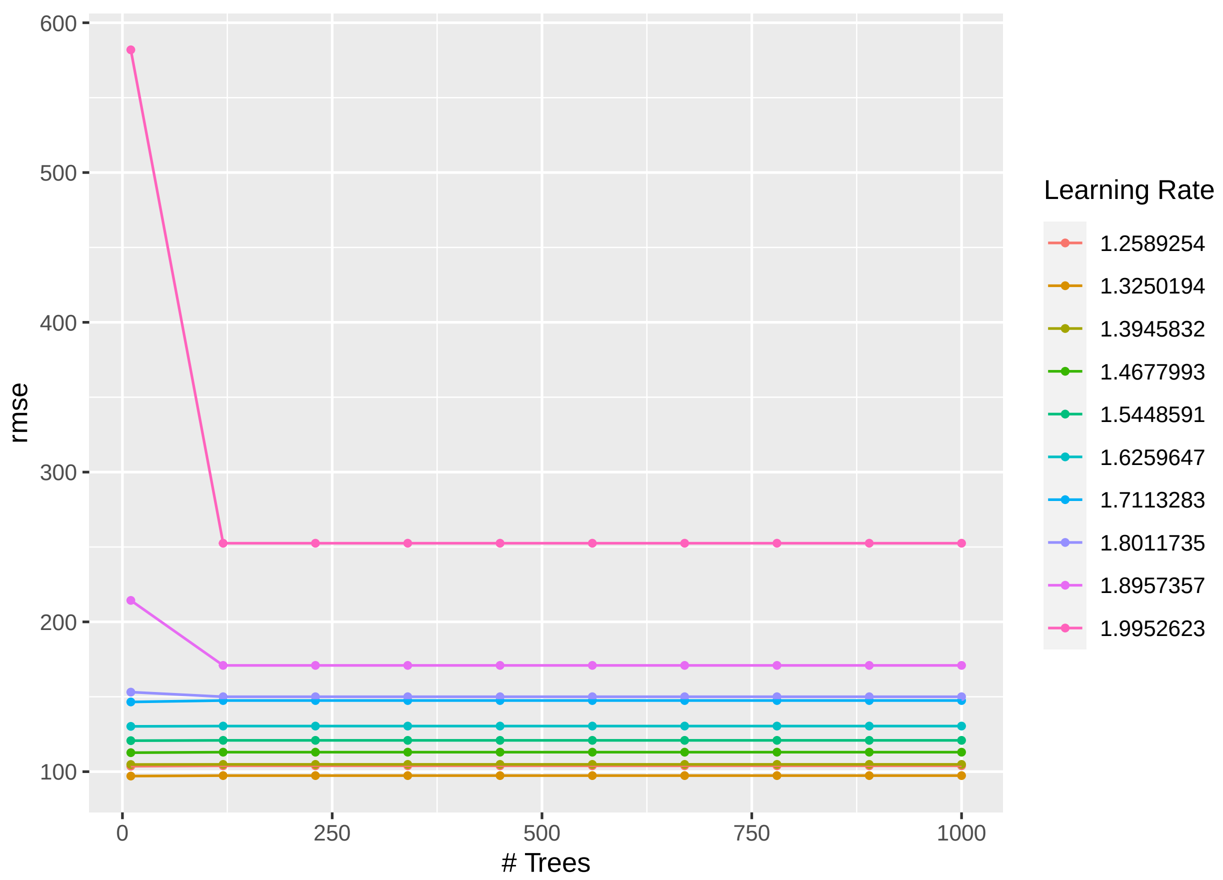
```
bt_tune <- tune_grid(
  bt_wkflow,
  resamples = coins_fold,
  grid = bt_grid,
  metrics = metric_set(rmse)
)
```

Here we save the bt_tune into rda file and load it, so that it will not take too much time to knit the Rmarkdown.

```
#save(bt_tune, file = "temp_rda/bt_tune.rda")

load(file = "temp_rda/bt_tune.rda")
```

Plot the bt_tune.

```
autoplot(bt_tune)
```



Now check the lowest RMSE.

```
head(collect_metrics(bt_tune) %>% arrange(mean))
```

```
## # A tibble: 6 x 8
##   trees learn_rate .metric .estimator  mean     n std_err .config
##   <int>      <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1    10       1.33 rmse    standard    97.1    50    5.26 Preprocessor1_Model011
## 2   120       1.33 rmse    standard    97.4    50    5.28 Preprocessor1_Model012
## 3   230       1.33 rmse    standard    97.4    50    5.28 Preprocessor1_Model013
## 4   340       1.33 rmse    standard    97.4    50    5.28 Preprocessor1_Model014
## 5   450       1.33 rmse    standard    97.4    50    5.28 Preprocessor1_Model015
## 6   560       1.33 rmse    standard    97.4    50    5.28 Preprocessor1_Model016
```

Select the best parameters.

```
best_bt <- select_best(bt_tune, metric = "rmse")
best_bt
```

```
## # A tibble: 1 x 3
##   trees learn_rate .config
##   <int>      <dbl> <chr>
## 1    10       1.33 Preprocessor1_Model011
```

Then finalize our random forest model. The best model should be fit again with the whole training data set.

```
bt_final <- finalize_workflow(bt_wkflow, best_bt)

bt_final_fit <- fit(bt_final, data = coins_train)
```

Apply the final model on the whole **training** set and validate its performance.

```
final_bt <- augment(bt_final_fit, new_data = coins_train)

final_bt %>% rmse(truth = eth, estimate = .pred)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard        7.44
```

The RMSE of our boosted tree model is only **7.4416747**, which is smaller than all three models before. So we should use boosted tree model as our final model.

Now let's continue with the **Boosted Tree Model** being the model that performed best.

Final Model Fitting

Now we can fit our boosted tree model to our testing data set.

```
final_bt_test <- augment(bt_final_fit, new_data = coins_test)

final_bt_test %>% rmse(truth = eth, estimate = .pred)
```

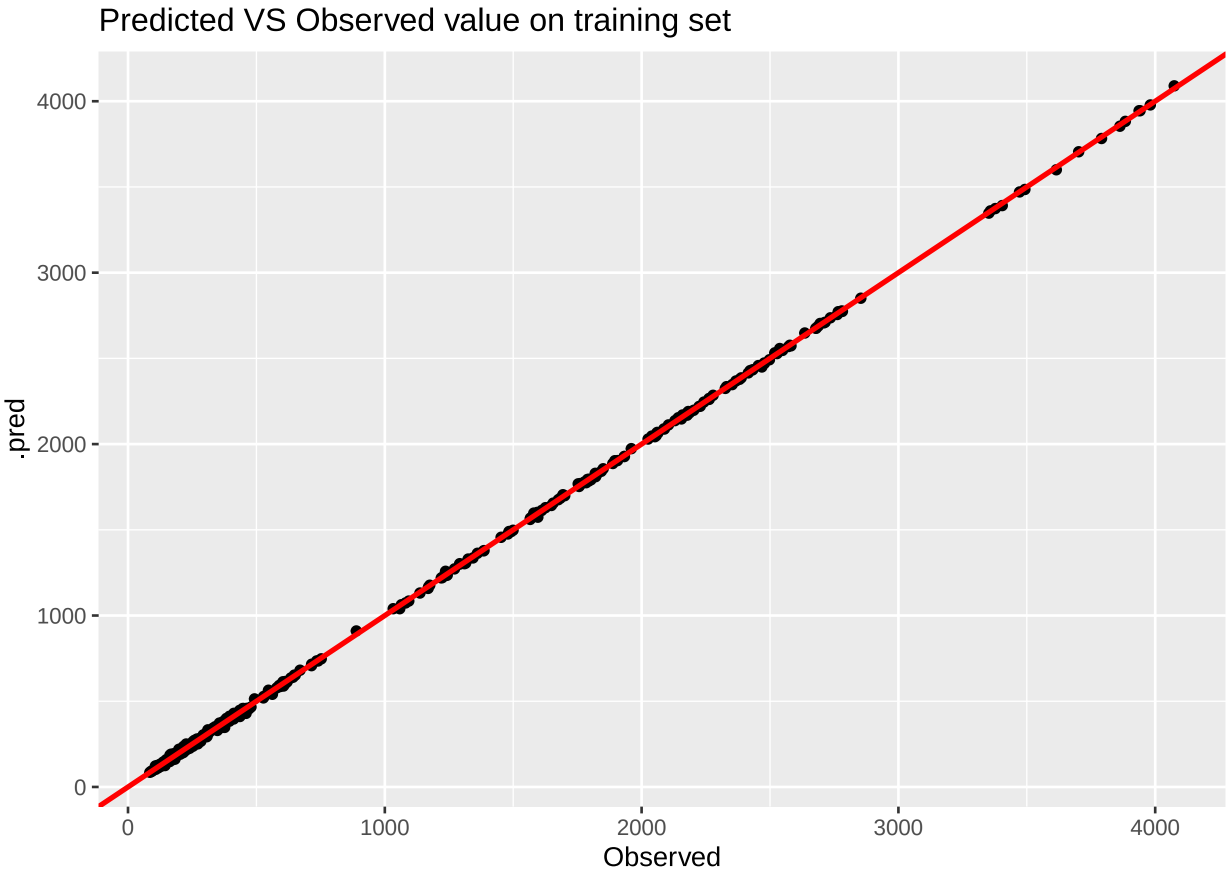
```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard        60.6
```

We find that the RMSE value = **60.641016** on the testing data set is much larger than the one from the training set which is **7.4416747**. It usually indicates that our model is overfitting to the data.

To see it clearly, let's take a close look at our predicted value and true value.

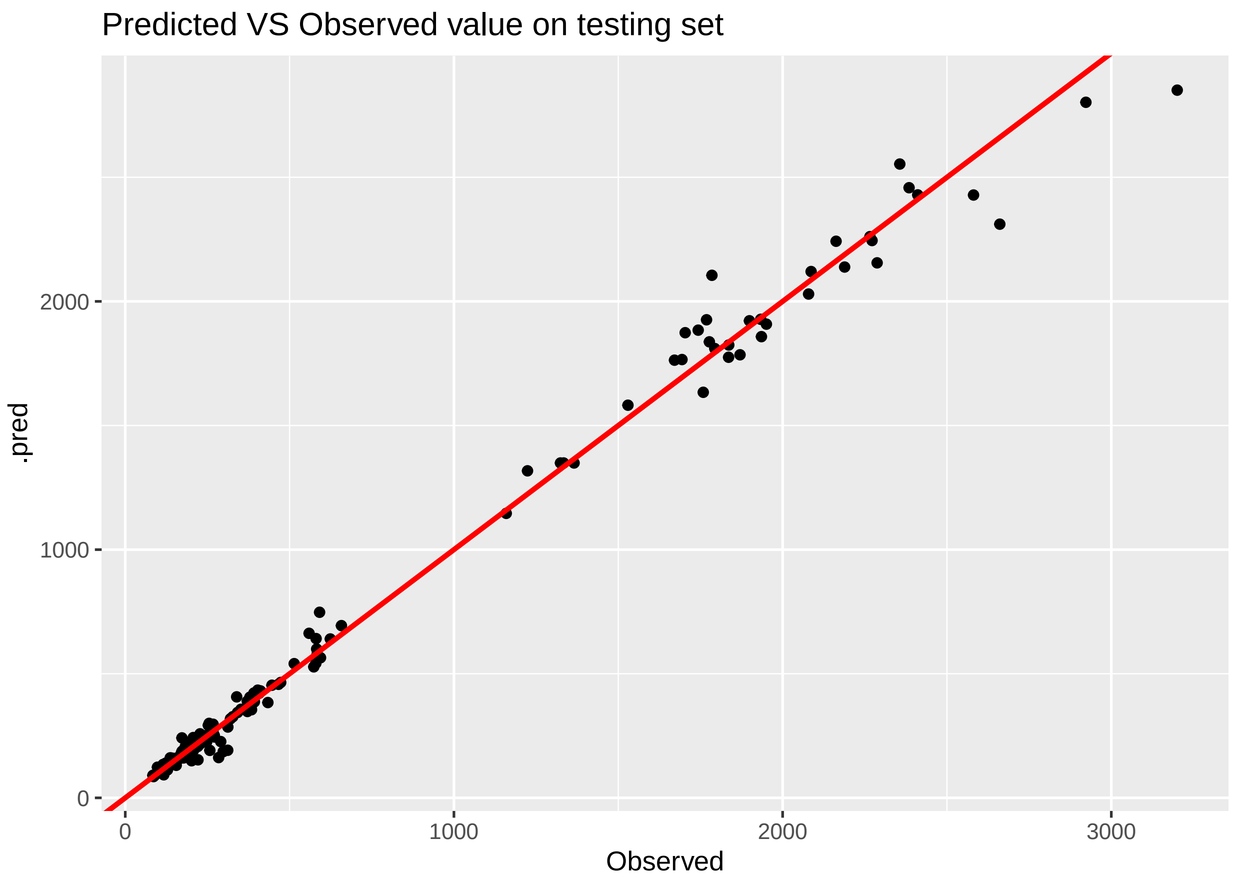
First, we plot the predicted value versus observed value on the **training** data set using our boosted tree model.

```
data_mod <- data.frame(predicted = predict(bt_final_fit,coins_train),
                       Observed = coins_train$eth)
data_mod %>% ggplot(aes(x = Observed,
                        y = .pred))+
  geom_point()+
  geom_abline(intercept = 0,
              slope = 1,
              color = "red",
              size = 1)+
  ggtitle("Predicted VS Observed value on training set")
```



Next, we plot the predicted value versus observed value on the **testing** data set using our boosted tree model.

```
data_mod2 <- data.frame(predicted = predict(bt_final_fit,coins_test),
                        Observed = coins_test$eth)
data_mod2 %>% ggplot(aes(x = Observed,
                         y = .pred))+
  geom_point()+
  geom_abline(intercept = 0,
              slope = 1,
              color = "red",
              size = 1)+
  ggtitle("Predicted VS Observed value on testing set")
```



From both plot, we can see our boosted tree model as the best model fits the training data set very well. It is almost perfect. However, the second plot shows that comparing to the training set, the model is not fitting that well on the testing set. But we can see the predicted value are not scattering too far from the observed value. It means although our model is overfitting, it still do a good job on the testing data.

Conclusion

Summary

This study use the historical mean price data of top 10 Cryptocurrencies to predict the mean price of Ether. And we are trying to find out if the price of Ether would be affected by other Cryptocurrencies. In the EDA part, we see that most of them have the same trend with time. What is more, some of them are strongly correlated to each other.In model fitting, we used k-fold Cross-Validation as our resampling method and 4 different models to fit the training data set. The 4 models are Ridge Regression model, K-Nearest Neighbors model, Random Forest model, and Boosted Tree model. Since we are doing regression project, we choose to use RMSE as our indicator to find the best model. The lower the value of RMSE, the better a model fits a data set, or the better the model in general.

Best Model

Our best model is the boosted tree model with RMSE of 7.4416747, which is the smallest among our models. The poorest model is the Ridge regression model with the highest RMSE of 112.09605. And the RMSE of the rest two model are really close to each other. The RMSE of KNN model is 20.697688, and the RMSE of the random forest model is 18.93514.

Although the boosted tree model has the lowest RMSE and seems to be perfect. When we fit it to the testing data set, the RMSE becomes quite larger than before with 60.641016. That means our model is overfitting to the testing data set. I think it may be caused by that there are few observations in the training data set. If we could have more observations in our data or split more for testing, then maybe the model would perform better.

Overall Conclusion

In conclusion, the study on the price data of Cryptocurrencies provides a way to predict the price of Ether using the price of other Cryptocurrencies, resulting in an interesting outcome for people to supervise the price of Ether by using other Cryptocurrencies' price. Also, we found that most of Cryptocurrencies have the same trend in price, even though their price are not on a same scale. Many kinds of Cryptocurrencies have strong correlations to each other. In other words, one's price can be greatly affected by the price of another or many kinds together.

Interesting Findings and Futher Study

Here are some interesting findings during our process. One is, since Cryptocurrencies have strong correlations to each other, I would like to know how much proportions of influence each has on the market. Recently, the news of the collapse of NFT and LUNA is very popular. I wonder if a price crash of one or more of them will be devastating to the entire market. Second, I would also like to know the proportion of the price changes of other cryptocurrencies in the overall factor of price changing.

For our models, we have mentioned about the high RMSE value on the testing data set and its potential reason. Next, I want to increase the sample observations in the original data set and increase the split scale of the testing set. Then repeat the model fitting to see if the boosted tree model is still the best model and if it still has larger RMSE on the testing set. Besides, I am interested in exploring the KNN model and Random Forest model as they have moderate RMSE value except the boosted tree, and they are close to each other. What if we fit them to the testing data. Would they be over fitting as well or doing a good job on it?

Reference

1. Image is from https://ethereum.org/en/
2. Video is from https://youtu.be/IsXvoYeJxKA
3. Data set is from Kaggle https://www.kaggle.com/datasets/sudalairajkumar/cryptocurrencypricehistory?resource=download
4. Lecture slides and labs.
5. Image is from https://storage.googleapis.com/kaggle-datasets-images/1869/3241/da6c5b9172b03a761a927eda0fd7dd67/dataset-cover.jpg

Figure 2: Image Source: https://storage.googleapis.com/kaggle-datasets-images/1869/3241/da6c5b9172b03a761a927eda0fd7dd67/dataset-cover.jpg