Ether Pricing Analysis

Ajay Dugar

December 10, 2019

Introduction and Background Knowledge

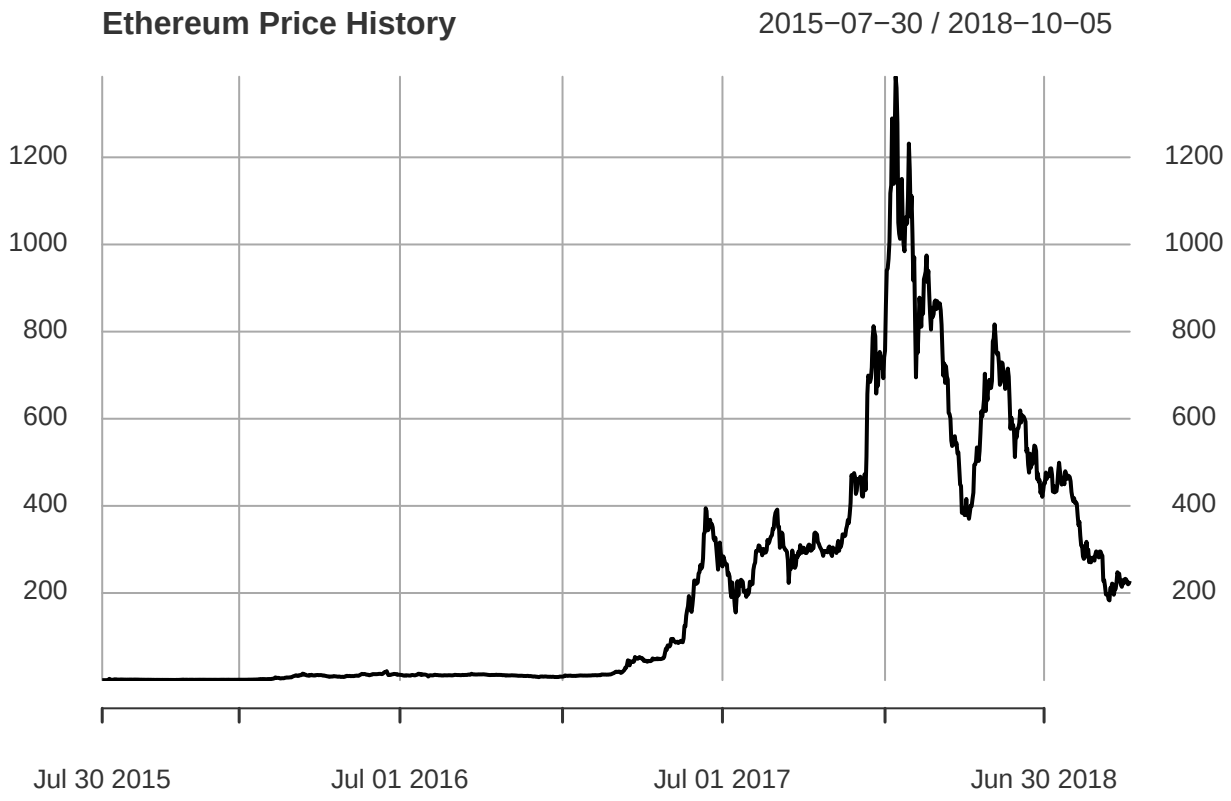
The purpose of this project is to determine the daily price of the cryptocurrency Ether (not to be confused with its blockchain platform Ethereum) using historical factor data (from July 30th, 2015 to October 5th, 2018), using a variety of techniques drawn from both classical statistics and machine learning. These models will be explored in the following sections. The factors that are going into this model are blocksize, hashrate, and transaction history

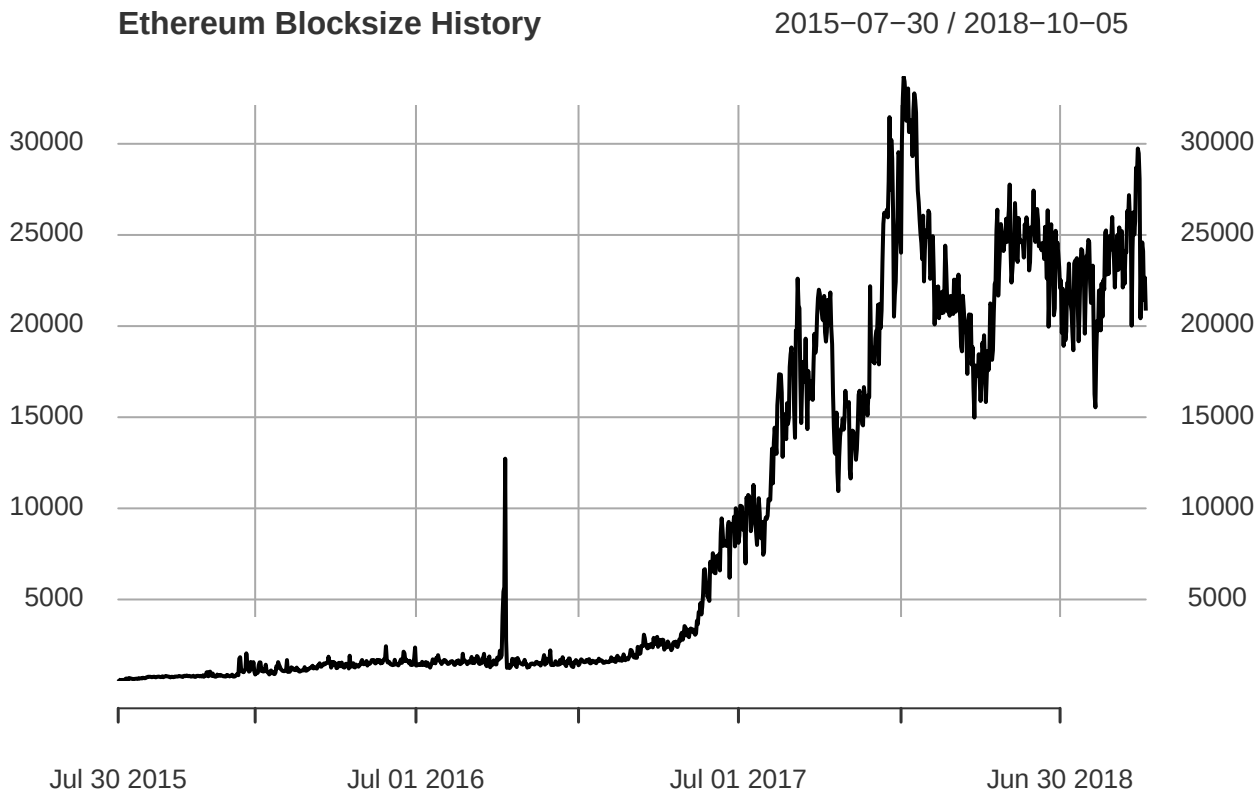
With respect to cryptocurrencies, a block refers to a bundle of transactions that take place over a network. Blocksize ('blocksize') refers to the amount of transactions that can be handled by a single block. This limit is mean to stop denial-of-service attacks on the network. Higher blocksize indicates increasing usage and transactions at a given time. Hashrate ('hashrate'), refers to the mining rate of a cryptocurrency, measured in hashes per second. This depends on the time to decrypt the cryptographic hash function unique to each cryptocurrency. Ethereum specifically uses the KECCAK-256 hash function (Bitcoin uses SHA-256). As more miners join a cryptocurrency's network, the difficulty of the hash function is increased, and the hash rate increases. For Ethereum, the hashrate adjusted such that the block time remains between 10 and 20 seconds (For Bitcoin, this is approximately 10 minutes).

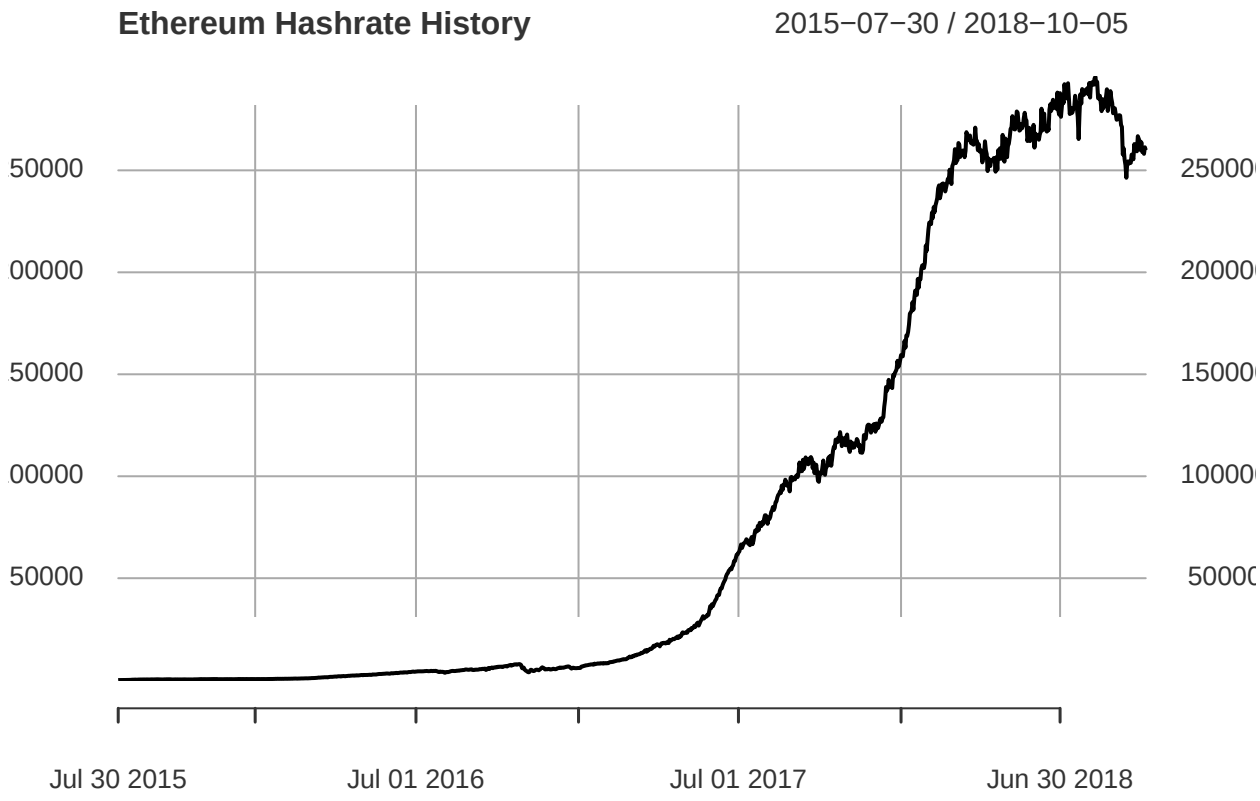
Transaction history ('trans_hist') is simply the number of transactions that take place on the Ethereum network. This is an indicator of the volume of activity of the users of Ether. An increasing number of users is an indication that the network is increasingly more trusted and an indication of the credibility of the network.

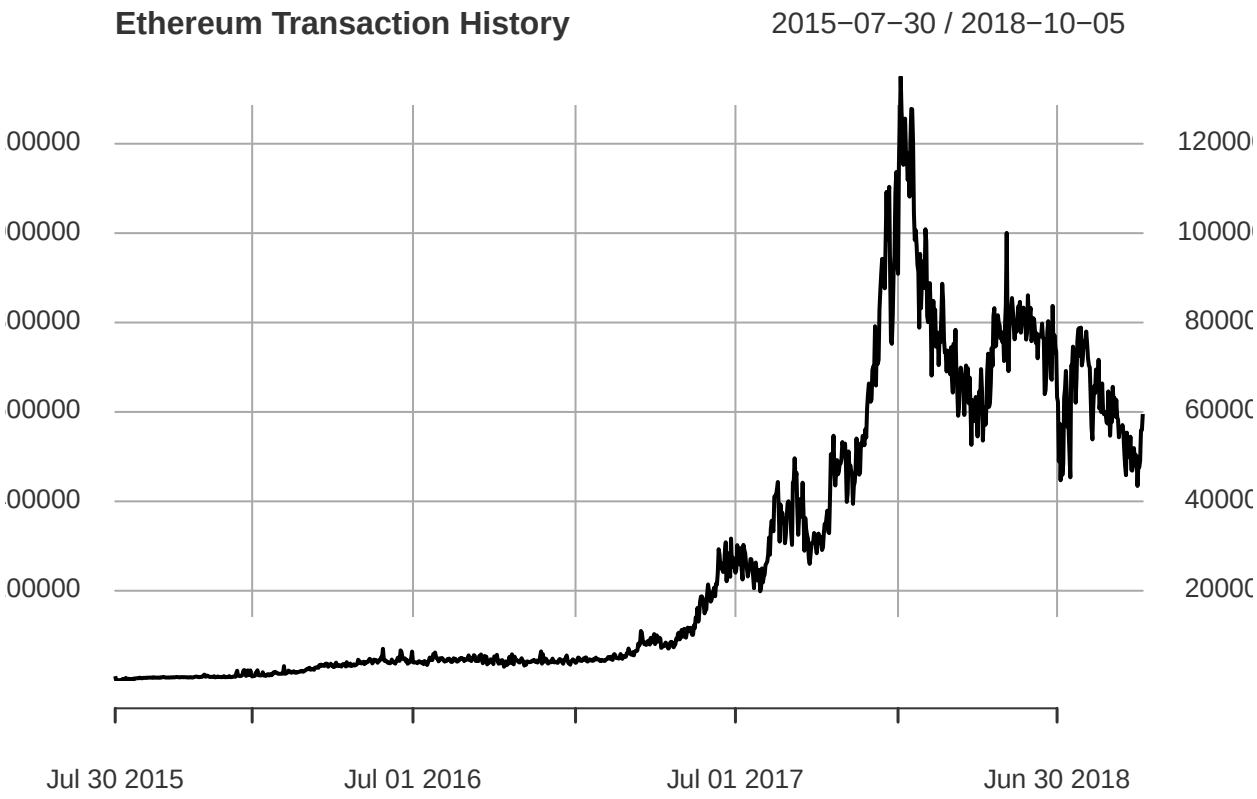
Summary Statistics

Below are the time series histories of Ether price, blocksize, hashrate, and transaction history.


Ethereum Price History
2015−07−30 / 2018−10−05
1200
1000
800
600
400
200
Jul 30 2015
Jul 01 2016
Jul 01 2017
Jun 30 2018

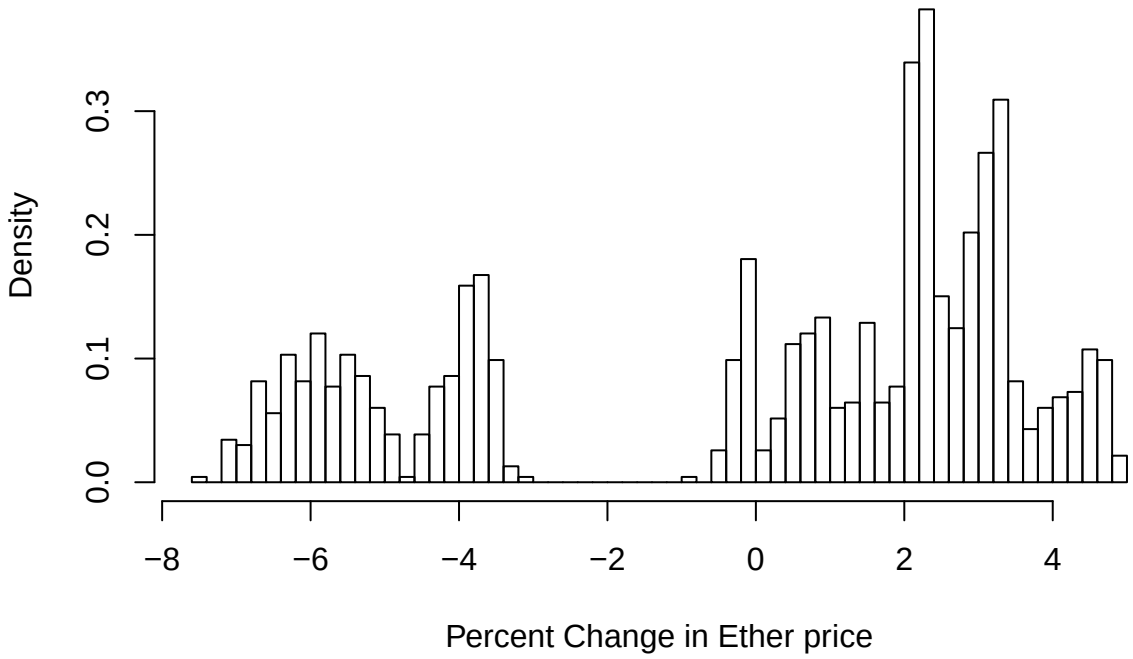

Ethereum Blocksize History
2015−07−30 / 2018−10−05
30000
25000
20000
15000
10000
5000
Jul 30 2015
Jul 01 2016
Jul 01 2017
Jun 30 2018


Ethereum Hashrate History
2015−07−30 / 2018−10−05
50000
00000
50000
00000
50000
25000
20000
15000
10000
50000
Jul 30 2015
Jul 01 2016
Jul 01 2017
Jun 30 2018

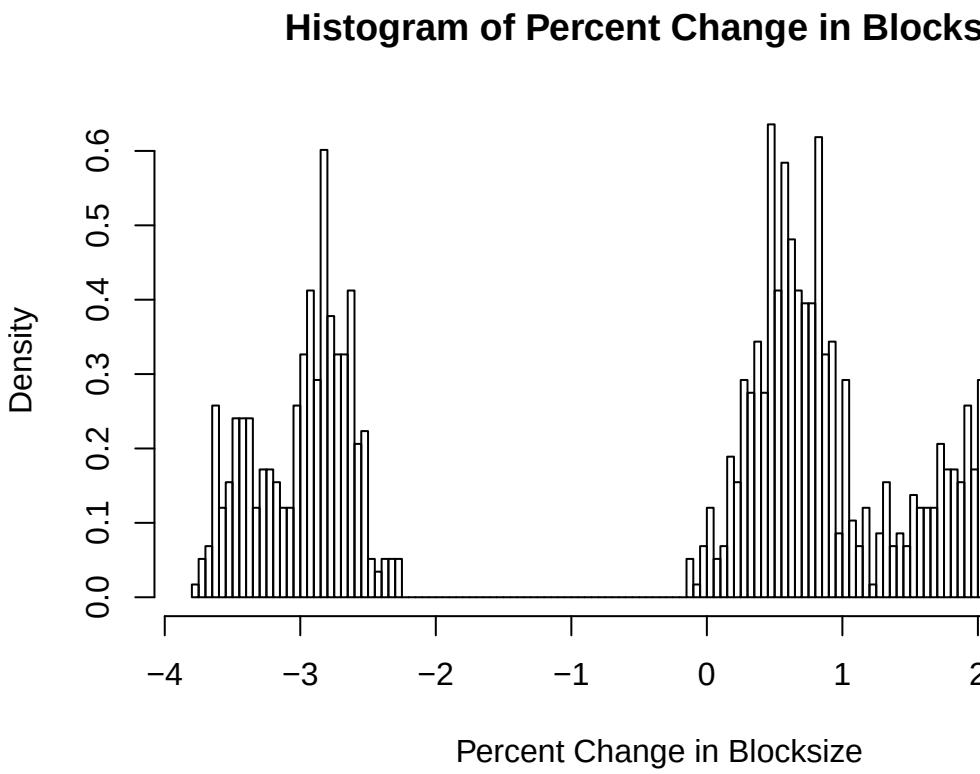


From the time-series data of both the independent and dependent variables, the classic conditional heteroskedasticity of financial instruments is observed, where the variance of the factor is correlated with the magnitude of the factor. Thus, we will use the log-difference of the numerical factors, $ln(Y(t))-ln(Y(t-1)) = ln(\frac{Y(t)}{Y(t-1)}) \approx \frac{Y(t)-Y(t-1)}{Y(t-1)}$, which is the percent change over a single time period.

The percentage change of Ether price:

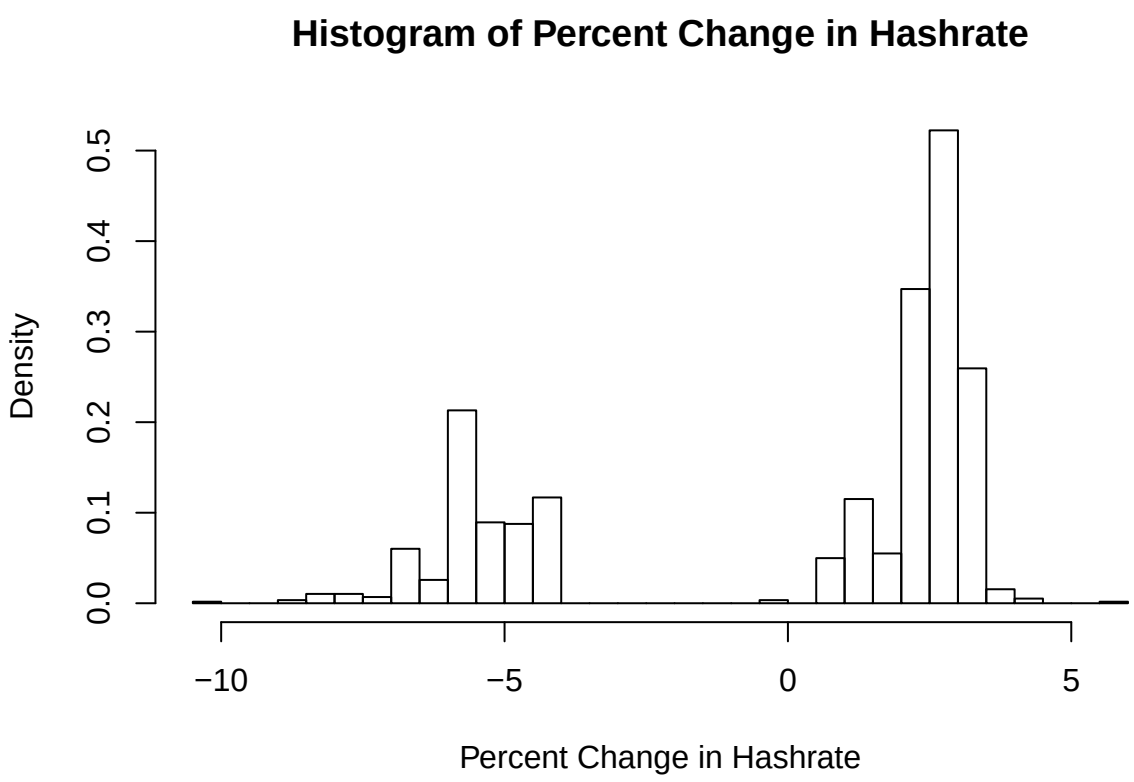


From this distribution, there is not immediately apparent statistical distribution of Ether price changes. A measure of centrality of this distribution is the median, for this distribution being 1.537.



The percentage change of the blocksize:

This a multimodal distribution and indicates that there are four regimes in the chain of percent changes. These appear to be four different normal distributions ($N(-3.356, 0.205), N(-2.743, 0.164), N(0.652, 0.321), N(2.262, 0.390)$). The changes demonstrate that there are four different levels of usage changes that occur on a day-to-day basis.




```
## [1] -5.470628
```

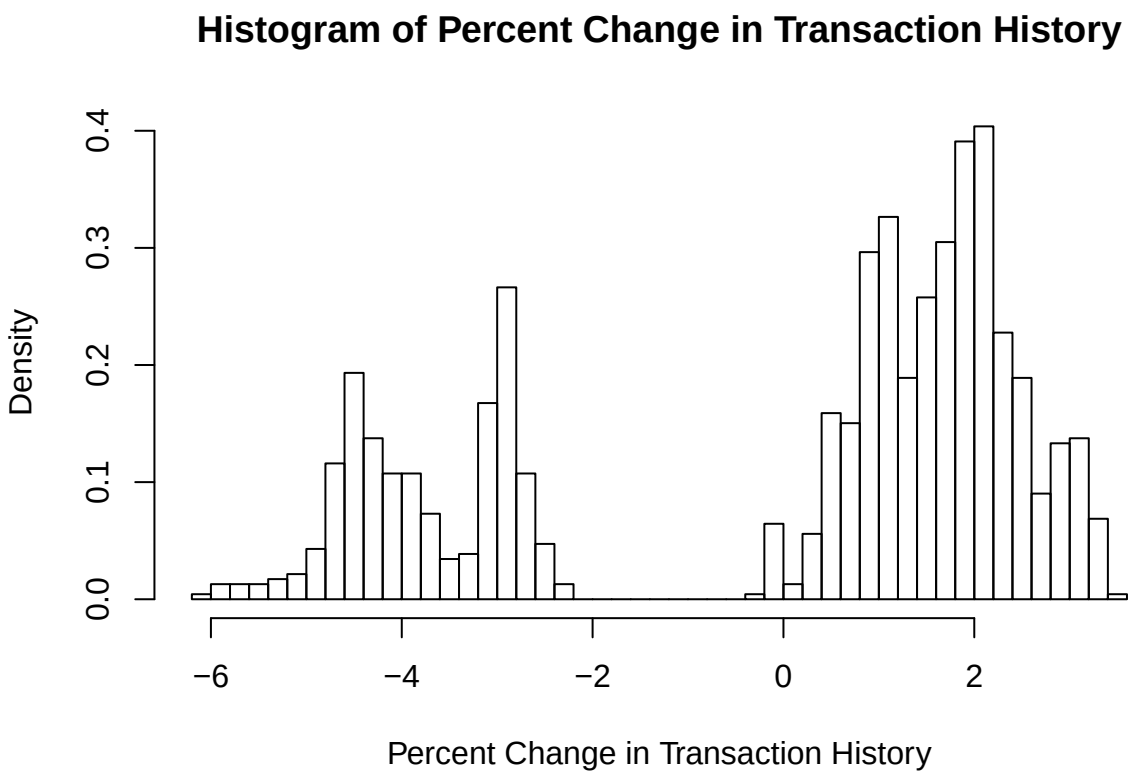
```
## [1] 0.9525354
```

```
## [1] 2.688662
```

```
## [1] 0.4260192
```

From the density distribution, it is clear that the hashrate percentage change distribution is bimodal. Assuming a normal distribution for each mode, the data gives $N(-5.470, 0.952)$ and $N(2.688, 0.426)$

```
hist(ether_data$ld_trans_hist, xlab = "Percent Change in Transaction History", main = "Histogram of Per
```



From the density distribution, it is clear that the transaction history percentage change distribution is trimodal. Assuming a normal distribution for each mode, the data gives $N(-4.451, 0.460)$, $N(-2.981, 0.301)$ $N(1.693, 0.777)$

Linear Regression (OLS)

From previous analysis (further explained in the Appendix), the model chosen was the following:

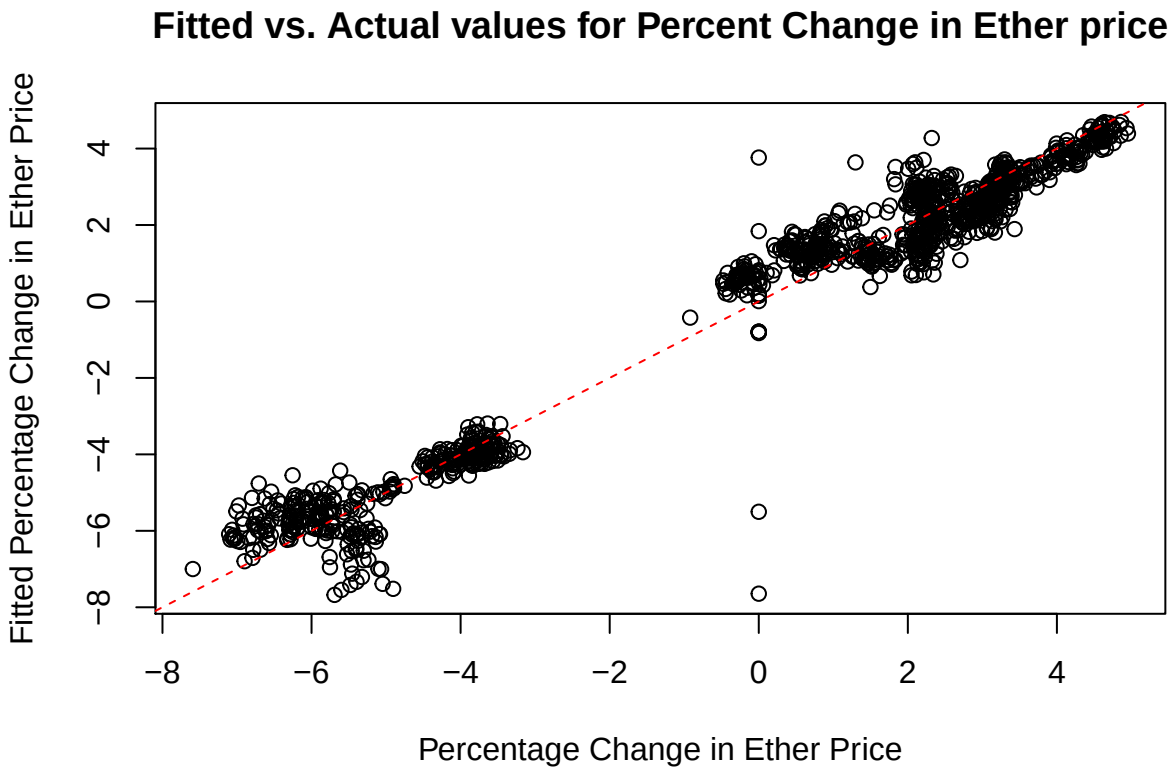
```
ols_reg_model <- lm(ld_price ~ ld_blocksize + ld_hashrate + ld_trans_hist, data = ether_data)
summary(ols_reg_model)
```

```
##
## Call:
## lm(formula = ld_price ~ ld_blocksize + ld_hashrate + ld_trans_hist,
##     data = ether_data)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.7637 -0.4860  0.0619  0.4207  7.6444
##
## Coefficients:
##                Estimate Std. Error t value Pr(>|t|)
## (Intercept)    0.008902   0.020861   0.427    0.670
## ld_blocksize   0.300950   0.038778   7.761 1.84e-14 ***
```

```
## ld_hashrate   -0.035029   0.023392  -1.498    0.135
## ld_trans_hist  1.152521   0.036256  31.788  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.7115 on 1160 degrees of freedom
## Multiple R-squared:  0.9613, Adjusted R-squared:  0.9612
## F-statistic:  9615 on 3 and 1160 DF,  p-value: < 2.2e-16
```

The idea behind choosing this model is that these were the variables most closely correlated with the Ether price and would make the most sense as drivers in the change of price. Including the seasonalities with respect to the month and day of the price changes appeared to overfit the model and were thus excluded, which makes sense as it shouldn't show any marked seasonal behavior as it is a cryptocurrency and not a commodity or equity. This model makes sense given that there should be a positive relationship between transaction history changes and price changes, as more trading tends to mean that price volatility is higher. This is similar to blocksize and inversely proportional to hashrate, which make their relationships make sense.

```
ether_data2 = data.frame(ether_data, ld_price_hat = fitted(ols_reg_model), e = residuals(ols_reg_model))
plot(ether_data2$ld_price, ether_data2$ld_price_hat, main = "Fitted vs. Actual values for Percent Change
abline(0,1, col = "red", lty = 2)
```




```
mean(residuals(ols_reg_model)^2)
```

```
## [1] 0.5044388
```

Looking at the plot, it appears that $\hat{ld\_price}$ is approximately equal to ld_price. The log difference appears to mitigate the heterskedasticity with the errors that were present when simply using raw price. The dashed red line across the plot is the line $\hat{ld\_price} = ld\_price$. The mean squared error of this model is 0.5044388.

Lasso Regression

From the overall data, a subset will be selected for training and testing. 80% of the data will be used to train the model, and the other 20% will be used to test the data. The same training and testing data will be the same for both the ridge and lasso regression to allow comparison.

A k-fold (k = 10) cross-validation was done to find the minimization of the objection function of the lasso regression.

```
cv.out.lasso = cv.glmnet(x_train, y_train, alpha = 1)
bestlam.lasso = cv.out.lasso$lambda.min
bestlam.lasso
```

```
## [1] 0.022892
```

From this analysis, $\lambda = 0.022892$ minimizes the cross-validated mean squared error. The plotted value of of the regressor coefficients with respect to lambda are given in the Appenix. The more significant the regressor, the more resistant it is to shrinkage as lambda increases. Using this minimal value, the test mean squared error can be calculated from this model.

```
lasso_mod = glmnet(x_train, y_train, alpha = 1, lambda = bestlam.lasso)
lasso_pred = predict(lasso_mod, s = bestlam.lasso, newx = x_test) # Use best lambda to predict test dat
mean((lasso_pred - y_test)^2)
```

```
## [1] 0.435132
```

The mean squared error for the lasso regression is 0.435132.

Ridge Regression

The same methods used to train and test the lasso regression will be utilized for the rdige regression, given that the only difference between the two models is the objective function.

A k-fold (k = 10) cross-validation was done to find the minimization of the objection function of the ridge regression.

```
cv.out.ridge = cv.glmnet(x_train, y_train, alpha = 0)
bestlam.ridge = cv.out.ridge$lambda.min
bestlam.ridge
```

```
## [1] 0.347938
```

From this analysis, $\lambda = 0.347938$ minimizes the cross-validated mean squared error. The plotted value of of the regressor coefficients with respect to lambda are given in the Appenix. The more significant the regressor, the more resistant it is to shrinkage as lambda increases. Using this minimal value, the test mean squared error can be calculated from this model.

```
ridge_mod = glmnet(x_train, y_train, alpha = 0, lambda = bestlam.ridge)
ridge_pred = predict(ridge_mod, s = bestlam.ridge, newx = x_test) # Use best lambda to predict test dat
mean((ridge_pred - y_test)^2)
```

```
## [1] 0.5916838
```

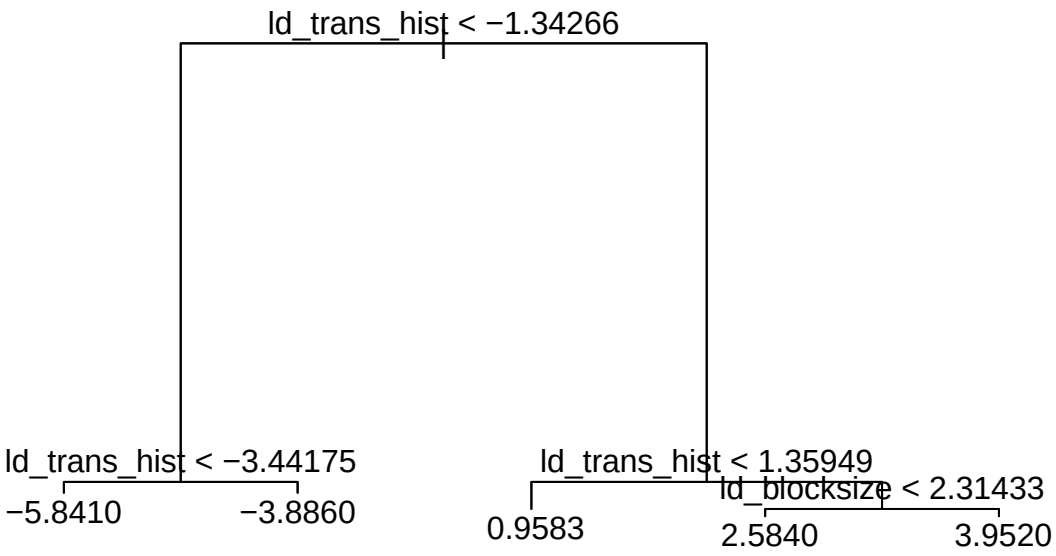
The mean squared error for the ridge regression is 0.5916838. Comparing between the mean squared error between the ridge and lasso regressions, the mean squared error of the lasso regression is smaller than the ridge regression ($0.435132 < 0.5916838$).

Decision Tree

```
tree_ether = tree(ld_price ~ ld_blocksize+ ld_hashrate + ld_trans_hist, train)
summary(tree_ether)
```

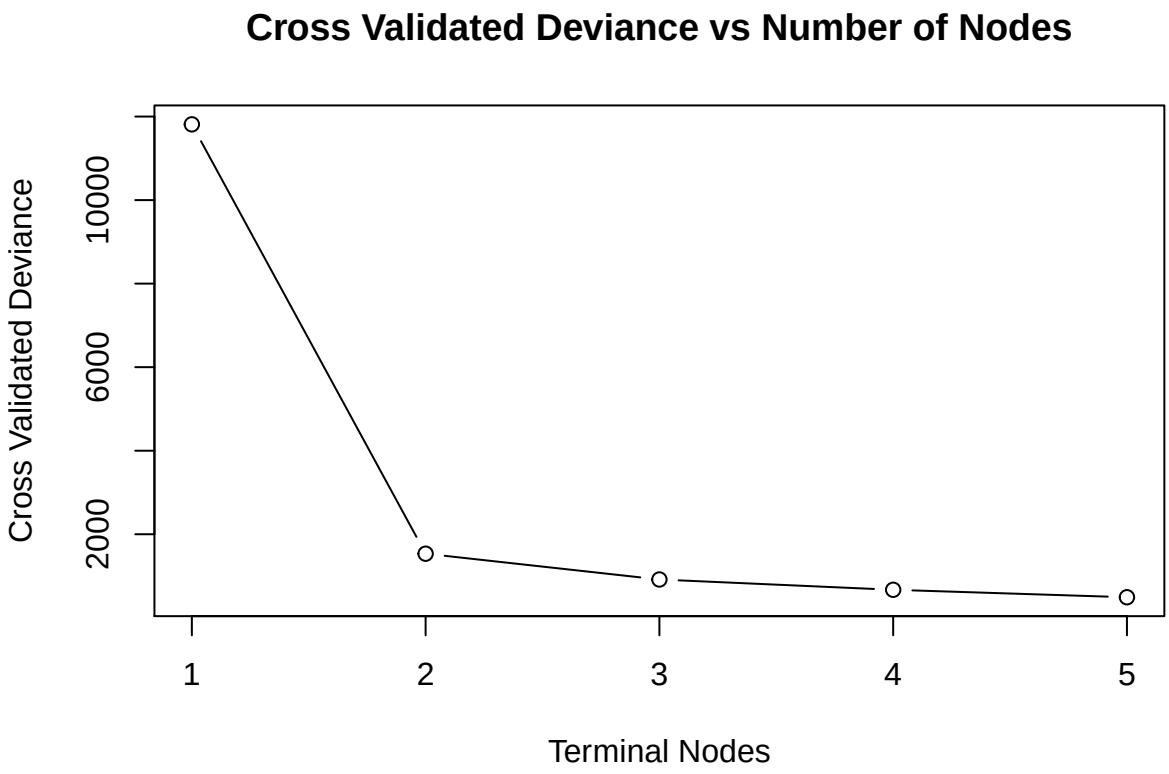
```
##
## Regression tree:
## tree(formula = ld_price ~ ld_blocksize + ld_hashrate + ld_trans_hist,
##     data = train)
## Variables actually used in tree construction:
## [1] "ld_trans_hist" "ld_blocksize"
## Number of terminal nodes:  5
## Residual mean deviance:  0.5063 = 468.9 / 926
## Distribution of residuals:
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -2.584000 -0.441000  0.001277  0.000000  0.474100  5.841000
```

```
plot(tree_ether)
text(tree_ether, pretty = 0)
```



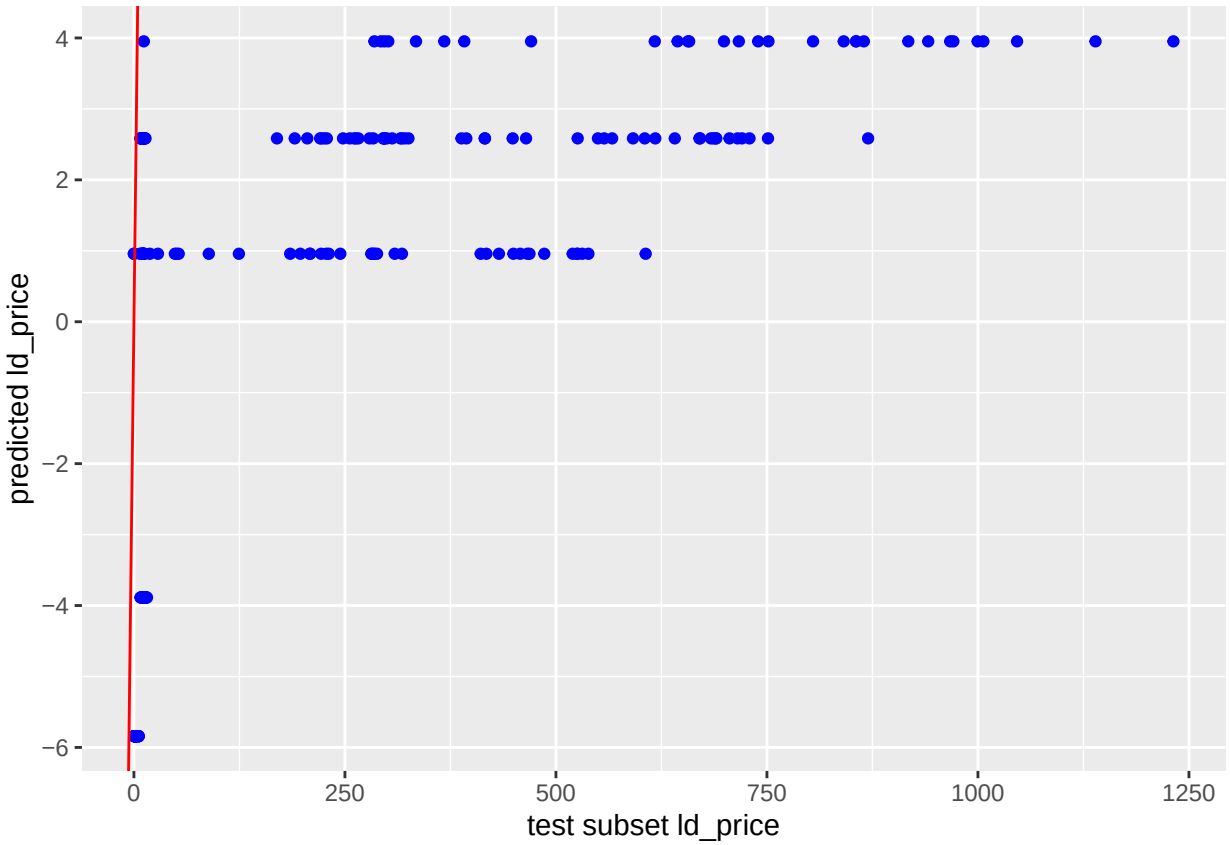
From the summary statistics and image of the unpruned decision tree, observe that there are four splits and five terminal nodes. percent chage of transaction history appears to be the primary driver of the percent change in Ether price percent change, given that it is the root and first two nodes.

```
set.seed(490)
cv.ether = cv.tree(tree_ether, FUN = prune.tree)
plot(cv.ether$size, cv.ether$dev, type = "b", main = "Cross Validated Deviance vs Number of Nodes", xlab
```



Given that the unpruned tree is relatively simple with only 5 terminal nodes, it is not surprising that the cross-validated deviance is lowest for the unpruned tree, as pruning the tree would give uncessary variance to the model. Usually, as the size of the tree increases, the issue of overfitting arises, but since this is not the case here, the unpruned tree will suffice.

```
single_tree_estimate = predict(tree_ether, newdata = test)

ggplot() +
    geom_point(aes(x = test$price, y = single_tree_estimate), color = 'blue') +
    geom_abline(color = 'red')+
    labs(x='test subset ld_price' , y= 'predicted ld_price')
```




```
mean((single_tree_estimate - test$price)^2)
```

```
## [1] 144329.1
```

From the predicted percent change in Ether price vs actual test percent changes in Ether price, there is a significant difference between the values. This is futher exemplified by the test mean squared error of 144329.1 which is order of magnitudes higher than other models. This is to be expected, given that this is time series data.

Bagging

Given that the single tree model did extremely poorly, perhaps bagging may be useful a useful ensemble method to improve upon the weak learning single trees.

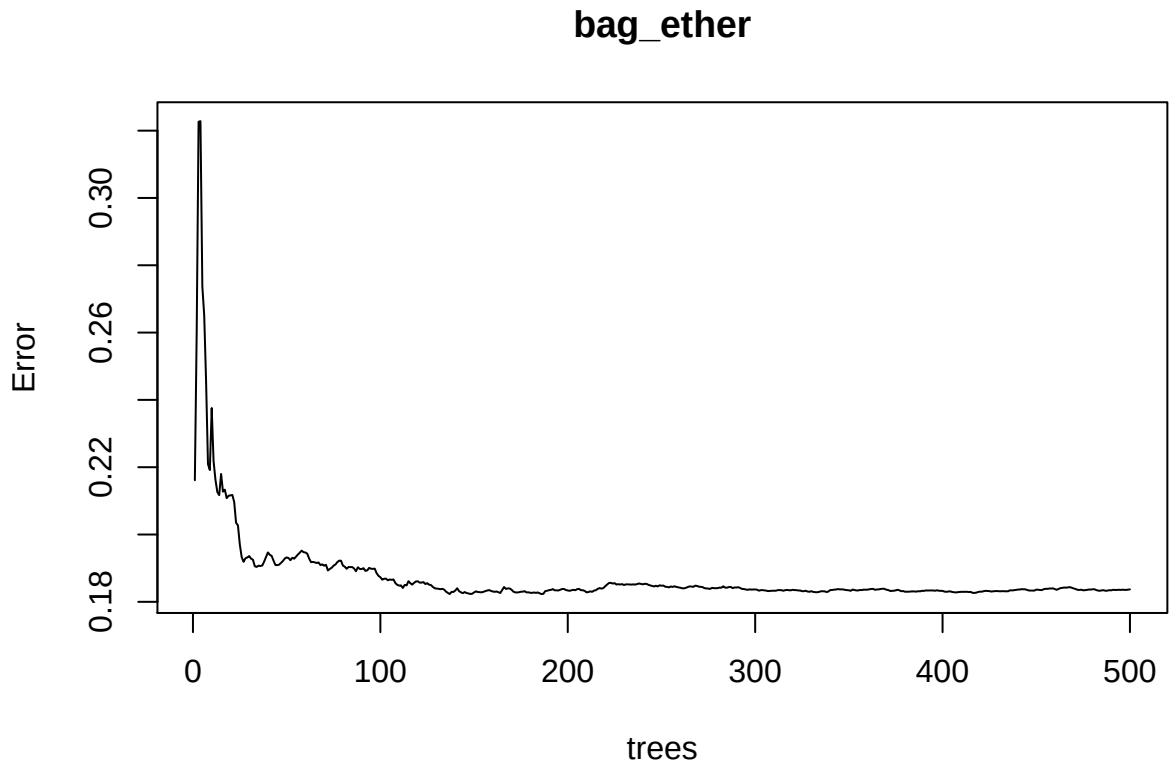
```
set.seed(490)
bag_ether = randomForest(ld_price ~ ld_blocksize + ld_hashrate + ld_trans_hist, data=train, mtry=3, imp
bag_ether
```

```
##
## Call:
##  randomForest(formula = ld_price ~ ld_blocksize + ld_hashrate +      ld_trans_hist, data = train, mt
##                Type of random forest: regression
##                      Number of trees: 500
```

```
## No. of variables tried at each split: 3
##
##           Mean of squared residuals: 0.1836853
##                     % Var explained: 98.55
```

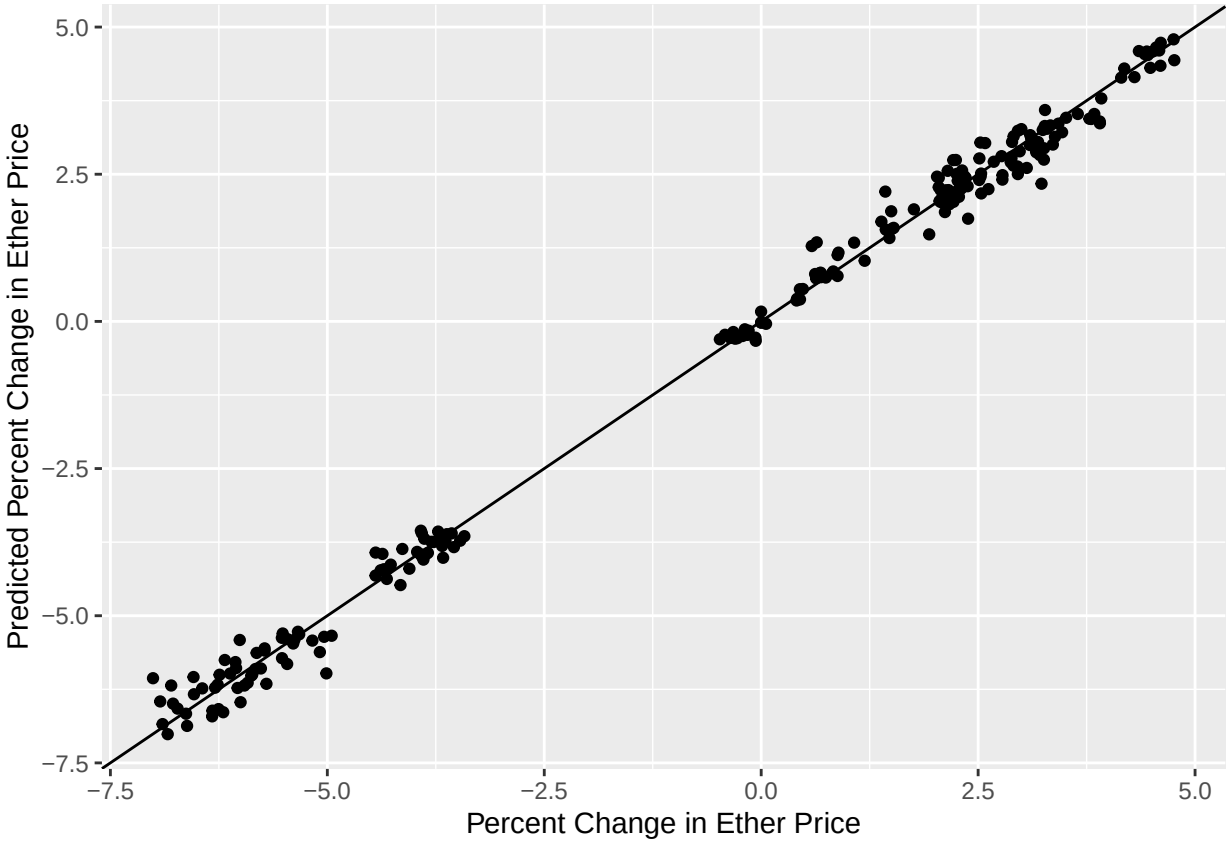
From the 500 bagged trees, the mean squared error (0.1836853) was significantly lowered compared to the single tree model from the previous section.

```
plot(bag_ether)
```



From the bagged model, the out-of-bag (OOB) error appeared to approach an asymptote of 0.18 around 200 trees.

```
yhat.bag = predict(bag_ether, newdata = test)

ggplot() +
    geom_point(aes(x = test$ld_price, y = yhat.bag)) +
    geom_abline()+
    labs(x="Percent Change in Ether Price", y="Predicted Percent Change in Ether Price")
```



Here the difference between the bagged trees and the individual tree is night and day. The estimated values are significantly closer to the actual values.

```
mean((yhat.bag-test$ld_price)^2)
```

```
## [1] 0.07257791
```

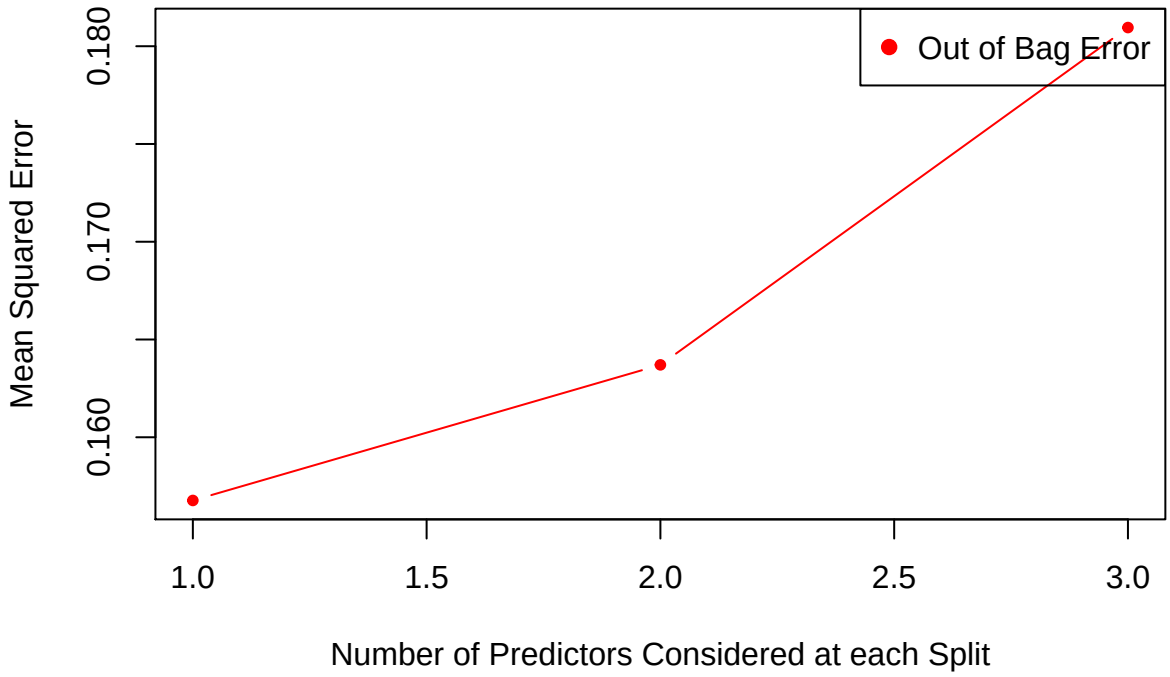
The test mean squared error of 0.07257791 is orders of magnitudes lower, and is on-par with the regression analyses done earlier.

Random Forest

The random forest is a similar method to bagging, except for the number of predictors used. In bagging, the full set of predictors was used, whereas in random forest regression, random subset of the predictors are used. Using hyperparameter tuning, by minimizing the out of bag error rate, the number of predictors can be chosen to run the random forest.

```
# Set mtry using hyperparamter tuning

oob.err<-double(3)
test.err<-double(3)

#mtry is no of Variables randomly chosen at each split
for(mtry in 1:3)
```

```
{
  rf = randomForest(ld_price ~ ld_blocksize + ld_hashrate + ld_trans_hist, data = train, mtry=mtry, ntr
  oob.err[mtry] = rf$mse[500] #Error of all Trees fitted on training

  pred <- predict(rf, test) #Predictions on Test Set for each Tree
  test.err[mtry]= with(test, mean((ld_price - pred)^2)) # "Test" Mean Squared Error
}
matplot(1:mtry, oob.err, pch=20 , col="red",type="b",ylab="Mean Squared Error",xlab="Number of Predicto
legend("topright",legend=c("Out of Bag Error"),pch=19, col=c("red"))
```



Given that a single predictor considered at each split minimizes the OOB MSE, the random forest can be train in this manner.

```
set.seed(490)
rf_ether = randomForest(ld_price ~ ld_blocksize + ld_hashrate + ld_trans_hist, data=train, mtry=1, impo
rf_ether
```

```
##
## Call:
##  randomForest(formula = ld_price ~ ld_blocksize + ld_hashrate +      ld_trans_hist, data = train, mt
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 1
##
##           Mean of squared residuals: 0.157859
##                     % Var explained: 98.75
```

Similar to the bagged model, the random forest does significantly better than the individual trees and appears to be similar in inference to the bagged model.

```
yhat.rf = predict(rf_ether, newdata = test)
mean((yhat.rf-test$ld_price)^2)
```

```
## [1] 0.08419493
```

The test mean squared error (0.08419493) is slightly higher than the bagged model, but this is to be expected given that the bagged model uses all predictors, while the random forest takes a subset of those predictors, an applicable example of the bias-variance tradeoff in model development.

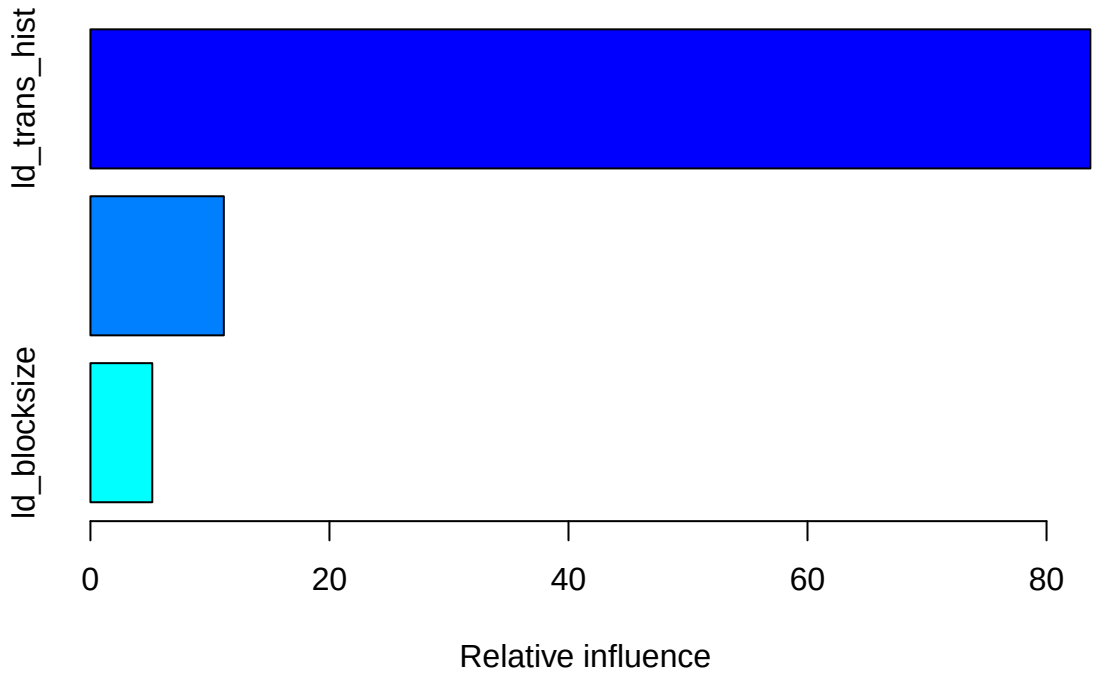
Boosting

Boosted regression trees are based on sequential improvement of the previous tree, where the final boosted tree is given by:

$$\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^b(x)$$

The shrinkage parameter, λ, control the learning rate of the tree and is typically between 0.01 and 0.001.

```
set.seed(490)
boost_ether = gbm(ld_price ~ ld_blocksize + ld_hashrate + ld_trans_hist, data=train, distribution = "ga
yhat.boost = predict(boost_ether, newdata = test, n.trees = 5000)
summary(boost_ether)
```



```
##                          var   rel.inf
## ld_trans_hist ld_trans_hist 83.666085
## ld_hashrate     ld_hashrate 11.159891
## ld_blocksize   ld_blocksize  5.174024
```

Obserbing the influence statistics, transaction history change is the most important driver in Ether price changes, significantly higher than both hashrate and blocksize. Next, evaluating this model on the test data set.

```
yhat.boost = predict(boost_ether, newdata = test, n.trees = 5000)
mean((yhat.boost - test$ld_price)^2)
```

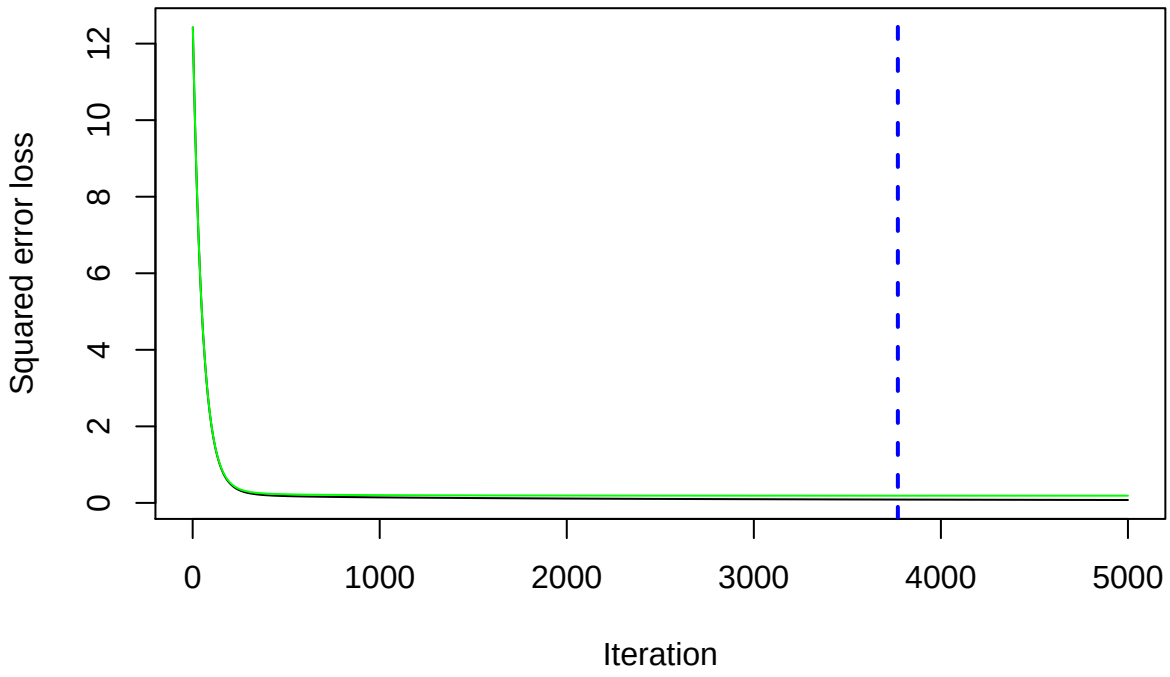
```
## [1] 0.143154
```

The test mean squared error of 0.143154 is similar to the previous ensemble tree models as above. However, with additional hyperparameter tuning, to find the optimal interations, we get:

```
boost.cv <- gbm(ld_price ~ ld_blocksize + ld_trans_hist + ld_hashrate, data = train,
                        distribution = "gaussian",
                        n.trees=5000,
                        interaction.depth=4,
                        shrinkage = 0.01,
                        verbose=F,
                        cv.folds=5)
```

```
## Warning in predict.gbm(model, newdata = my.data, n.trees = best.iter.cv): NAs
## introduced by coercion

## Warning in predict.gbm(model, newdata = my.data, n.trees = best.iter.cv): NAs
## introduced by coercion

## Warning in predict.gbm(model, newdata = my.data, n.trees = best.iter.cv): NAs
## introduced by coercion

## Warning in predict.gbm(model, newdata = my.data, n.trees = best.iter.cv): NAs
## introduced by coercion

## Warning in predict.gbm(model, newdata = my.data, n.trees = best.iter.cv): NAs
## introduced by coercion
```

```
bestTreeForPrediction <- gbm.perf(boost.cv) #GBM performance stimates the optimal number of boosting it
```




```
bestTreeForPrediction
```

```
## [1] 3770
```

The black line above is the training RMSE, and the green line is 5-fold cross validation RMSE, giving us the optimal number of trees of 3566.

```
boost.cv <- gbm(ld_price ~ ld_blocksize + ld_trans_hist + ld_hashrate, data = train,
                        distribution = "gaussian",
                        n.trees=3566,
                        interaction.depth=4,
shrinkage = 0.01,
                        verbose=F)
yhat.boost <- predict(boost.cv, test, n.trees = 3566)
mean((yhat.boost-test$ld_price)^2)
```

```
## [1] 0.1002167
```

The optimized mean squared error of 0.1002237 is significantly smaller than the untuned mean squared error of the boosted tree model of 0.143154.

XGboost

```
dtrain = xgb.DMatrix(data = x_train, label = y_train)
ether_xgb = xgboost(data=dtrain,
                    max_depth=5,
                    eta = 0.1,
                    nrounds=500, # max number of boosting iterations (trees)
                    lambda=0,
                    print_every_n = 50,
                    objective="reg:linear")
```

```
## [1]  train-rmse:3.223496
## [51] train-rmse:0.182737
## [101]    train-rmse:0.135133
## [151]    train-rmse:0.100923
## [201]    train-rmse:0.079414
## [251]    train-rmse:0.065723
## [301]    train-rmse:0.054347
## [351]    train-rmse:0.045154
## [401]    train-rmse:0.037743
## [451]    train-rmse:0.031696
## [500]    train-rmse:0.027004
```

From the train mean squared error, observe that there is significant improvement with every additional tree used, due the gradient boosting factor in the objective function:

$$obj = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{t-1} + f_t(x_t)) + \Omega(f_t) = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{t-1} + f_t(x_t)) + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} \omega_j^2$$

```
yhat.xgb <- predict(ether_xgb, x_test)
mean((yhat.xgb - test$ld_price)^2)
```

```
## [1] 0.07768476
```

This test MSE of 0.07768476 is quite good, but by tuning the hyperparameters using a grid search iteration, improvements can be made.

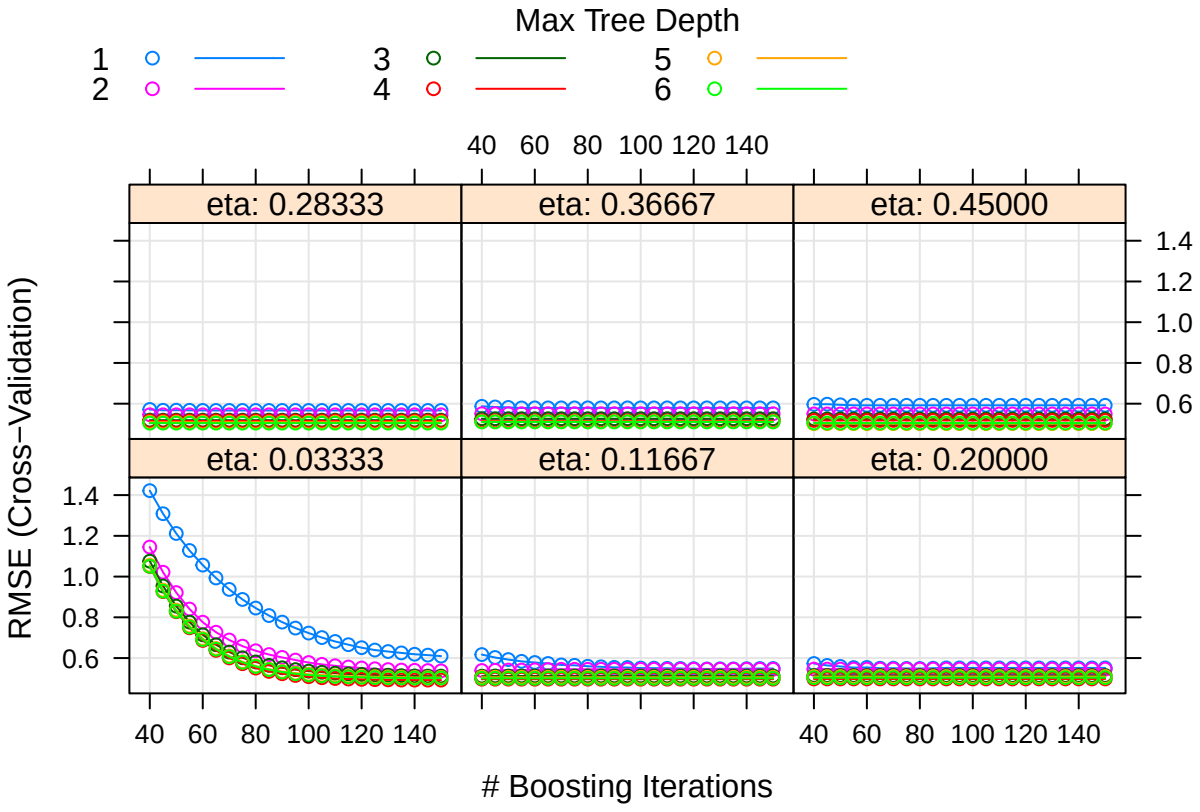
```
hyper_grid <- expand.grid(
    nrounds = seq(40,150,5),
    max_depth = c(1,2,3,4,5,6),
    eta = seq(2,30,5)/60,
    gamma = 1,
    colsample_bytree = 1.0,
    min_child_weight = 1,
    subsample = 1.0
)
xgb_control <- trainControl(
  method="cv", #re-samoling methods: "boot", "boot632", "optimism_boot", "boot_all","repeatedcv", "LOOC
  number = 3
)
```

```
set.seed(490)
ether.xgb.tuned <- train(ld_price~ ld_trans_hist + ld_hashrate + ld_blocksize, data=train,
                         trControl=xgb_control,
                         tuneGrid=hyper_grid,
                         lambda=0,
                         method="xgbTree")
ether.xgb.tuned$bestTune
```

```
##    nrounds max_depth        eta gamma colsample_bytree min_child_weight
## 92     150         4 0.03333333     1                1                1
##    subsample
## 92         1
```

From the entirety of the different 3-fold cross validation models set up, we see that the best model has the hyperparameters.

```
plot(ether.xgb.tuned)
```



Here are the individual hyperparameters with respect to both iterations and cross-validated RMSE.

```
ether.xgb = xgb.train(param = ether.xgb.tuned$bestTune,
                      data=dtrain,
                      nround = 150)
```

```
yhat.xgb <- predict(ether.xgb,x_test)
mean((yhat.xgb - test$ld_price)^2)
```

```
## [1] 0.1058906
```

This mean squared error (0.1058906) is slightly higher than the MSE of the default hyperparameters (0.07768476). Given that the hyperparameters are found from a grid search, it is important to ensure that the hyperparameters encompass an adequate range of values. However, increasing these ranges is a tradeoff that comes with computational time with respect to the thoroughness of this search.

Neural Net

In order to train the neural network, normalization of the variables is important in order for the weighting of the neurons to be correct.

```
model <- keras_model_sequential() %>%
 layer_dense(units = 64, activation = "relu", input_shape = dim(train_data)[2]) %>%
 layer_dense(units = 64, activation = "relu") %>%
 layer_dense(units = 1)
```

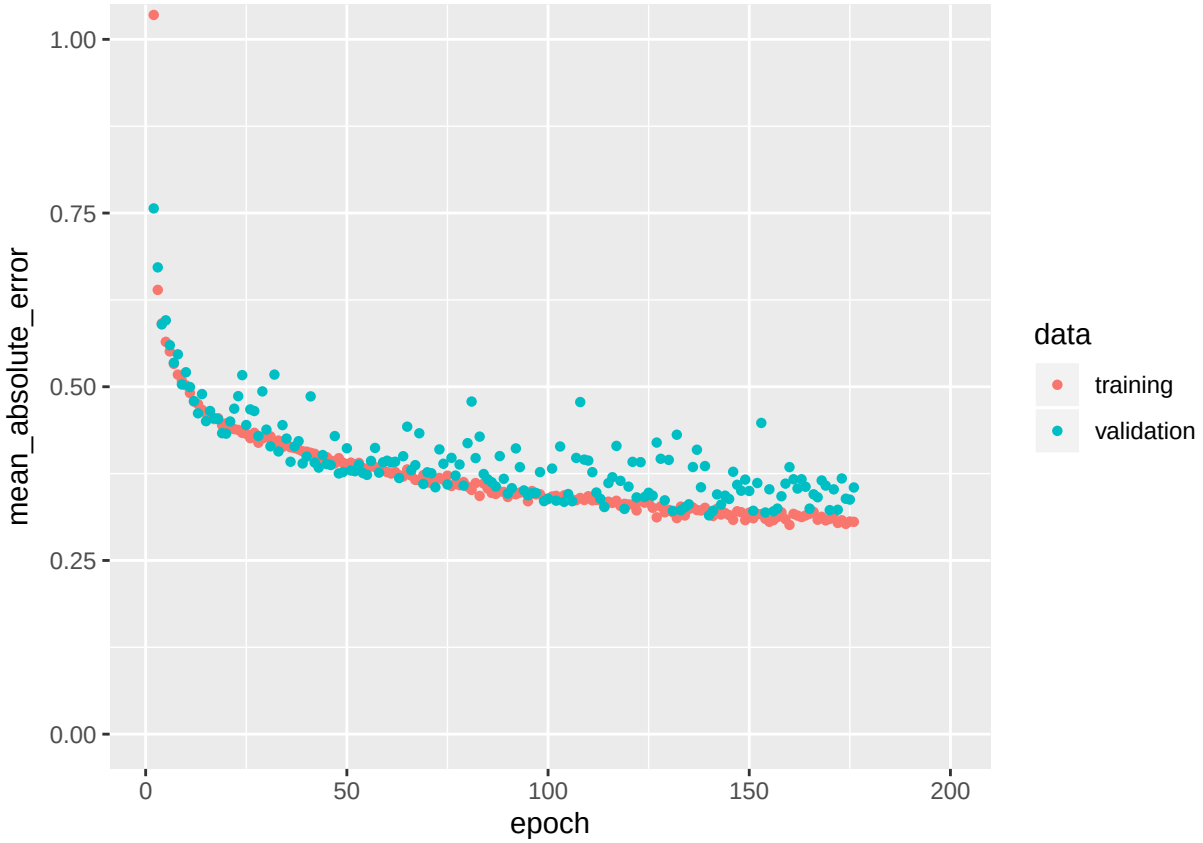
Here we have the model with three different layers, the input layer with 64 neurons, a hidden layer with 64 neuron, and an output layer with a single neuron for regression.

```
model %>% compile(
 loss = "mse",
 optimizer = optimizer_rmsprop(),
 metrics = list("mean_absolute_error")
)

early_stop <- callback_early_stopping(monitor = "val_loss", patience = 20)
```

```
epochs <- 200
history <- model %>% fit(
 train_data,
 train_labels,
 epochs = epochs,
 validation_split = 0.2,
 callbacks = list(early_stop)
)
```

Using the early stop function, it takes only 40 epochs to fit the data sufficiently such that the prediction on the validation set does not suffer from overfitting to the training data.

```
plot(history, metrics = "mean_absolute_error", smooth = FALSE) +
 coord_cartesian(ylim = c(0, 1))
```




```
test_predictions <- model %>% predict(test_data)
mean((test_labels - test_predictions)^2)
```

```
## [1] 0.7029185
```

The neural network gives a MSE of 0.6252429. This is noticeably worse than a number of other methods. However, given that the number of data points used to train the model (931) is relatively small. Typically a neural network requires at least the number of inputs as the number of convolutions between its layers.

Model Comparisons

Here is a table of all of the methods used with respect to MSE. This can be done because of the fact that the training and testing sets were the same for all methods.

Method	MSE
OLS Regression	0.5044388
Lasso Regression	0.435132
Ridge Regression	0.5916838
Single Decision Tree	144329.1
Bagged Trees	0.07257791
Random Forest	0.08419493
Boosted (untuned)	0.143154
Boosted (tuned)	0.1002237

Method	MSE
XGboost	0.1058906
Neural Network	0.6252429

There are number of observations that can be made. First, hyperparameter tuning allows for many different combinations of models to be analyzed. These are critical to improving performance of models. Additionally, as the complexity of the statistical methods increase, the more training data that is required to ensure adequate robustness and accuracy in regards to predictive power. This was most apparent when evaluatingthe neural network. Finally, ensemble methods far outperform single methods, as cross-validation and interations allow for incremental improvement that adds up over epochs. The best method based on MSE was the Bagged Trees method, which was suprising given that this a regression task, rather than a classfication task.

Conclusion

Pricing analysis of financial assets is difficult. Even more difficult is to determine the pricing of an asset without any underlying value. Cryptocurrencies function based on the strength of encryption and as a mutual understanding of the participants as medium of exchange and value. With the lack of centralization of oversight and authority, the prices of cryptocurrencies are prone to much greater volatility than traditional financial assets. The 3-factor model used based on transaction history, hashrate, and blocksize was quite accurate when modelling pricing, once the data was normalized. Moving forward, for future research, it may be prudent to see if the same factors play similar roles and have equivalent predictive power for other cryptocurrencies.

In regards to the methods found this paper, as the amount of data available to statistical practitioners increases, the need for more advanced statistical techniques arises. In finance, where every advantage is crucial, mathematical and statistical modelling is becoming more and more necessary and prevalent. Given that there are a near infinite number of predictors and drivers when it comes to financial asset prices, it becomes important to understand how to select models and tune them in order to get actionable intelligence from data.

Appendix

The purpose of this section is to show the background work that went into determining the specific models shown above. The individual code is given here, but is not evaluated for the sake of presentability.

Heteroskedasticity

```
qqnorm(ether_data2$e)
qqline(ether_data2$e)
```

Here we see the deviation from normality from the tails of the error distribution, giving credence to the leptokurtic nature of the distribution.

Correlations Between Variables

```
cor(ether_data[,c(9:12)])
pairs(ether_data[,c(9:12)])
```

Regressions

This is the initial linear model selection. We will look at 5 different dependent variables and their combinations and select the best model based of off the adjusted R^2 value. The independent variables are blocksize, hashrate, transaction history, month, and day of the week.

```
as.factor(ether_data$month)
as.factor(ether_data$dayofweek)
summary(lm(price ~ blocksize + hashrate + trans_hist + month + dayofweek, data = ether_data))
summary(lm(price ~ blocksize + hashrate + trans_hist + month, data = ether_data))
summary(lm(price ~ blocksize + hashrate + trans_hist, data = ether_data))
summary(lm(price ~ blocksize + hashrate + month + dayofweek, data = ether_data))
summary(lm(price ~ trans_hist + month + dayofweek, data = ether_data))
```

Looking at the regression results, we can see that because the months and day of the week are factors with a large number of dummy variables, that there may be an overfitting of the model with this many independent variables. The relationships between the independent variables and dependent variables make sense. There should be a positive relationship between transaction history and price, as more trading tends to mean that prices are more speculative and volatile, which should be positively correlated with price. This is similar to blocksize and inversely proportional to hashrate, which make their relationships make sense. Additionally, having only some months and days being significant makes sense, as it shouldn't show any marked seasonal behavior as it is a cryptocurrency and not a commodity or equity.

$\hat{Y}$ vs Y

```
ether_data2 = data.frame(ether_data, price_hat = fitted(model), e = residuals(model))
plot(ether_data2$price, ether_data2$price_hat)
abline(1,1, col = "red", lty = 2)
```

Looking at the plot, it appears that price_hat is approximately equal to price. There appears to be some heteroskedasticity with the errors as we increase in magnitude for price. It may be prudent in future analysis to take the log(price) as our dependent variable. The dashed red line across the plot is the line $\hat{price} = price$.

Lasso and Ridge Regression

```
plot(cv.out.ridge)
plot(cv.out.lasso)
```

Here we see the training MSE for each value of lambda. MSE increases as lambda increases. The standard error bars are shown at each point.

```
out = glmnet(x_train, y_train, alpha = 0)
predict(out, type = "coefficients", s = bestlam.ridge) # Display coefficients using lambda chosen by CV
plot(out, xvar = "lambda")
```

```
out = glmnet(x_train, y_train, alpha = 1) # Fit lasso regression model on the FULL dataset (train and t
predict(out, type = "coefficients", s = bestlam.lasso) # Display coefficients using lambda chosen by CV
plot(out, xvar = "lambda")
```

From the above two plots, the shrinkage of the predictors with respect to lambda are shown.