Cryptocurrency Returns and Machine Learning

PCA, Clustering, and Return Forecasts using LSTM & GARCH

Runhua Li

Xinyu Liu

Joanna Zhang

Ying Zhou

March 12, 2020

Contribution Statement

Runhua: GARCH model construction and model comparison

Xinyu: LSTM model construction and tuning

Joanna: Data collection, LSTM model evaluation

Ying: Clustering and PCA

1 Introduction

Nowadays, machine learning has become a handy tool in explaining trends in the financial markets, such as asset return movements. Literature in applying machine learning algorithms in both the equity markets [HNW04, SAF15], and the cryptocurrency markets [AEAB18, RKDP18] has recognized machine learning as a reliable instrument.

Our project aims at applying machine learning techniques to generate financial time series forecasts. In particular, we use past returns of cryptocurrencies to forecast future returns. The motivation of this project is to test the weak form market efficiency hypothesis (MEH) in cryptocurrency markets. The weak form MEH states that future stock returns are not dependent on past returns, and thus analyzing past prices will not be helpful for forecasting returns. If this hypothesis truly holds, we shall expect very limited forecasting power from our machine learning models.

Specifically, we intend to investigate how accurate and consistent are the predictions for cryptocurrency returns from machine learning models, using only past returns, and how do they compare to traditional time series models?. We first cluster selected cryptocurrencies into groups based on their trading information. Then we construct a Long Short Time Memory (LSTM) network with tuned hyperparameters. We carefully evaluate test sample RMSEs between LSTM and the traditional econometric GARCH model. In the end, we conclude that despite the limited forecasting power, LSTM outperforms the benchmark econometric model at most levels.

This project extends on the existing literature by making a systematic comparison of forecasts generated by LSTM and GARCH, using returns instead of prices as inputs. It also provides crucial empirical evidence on the weak form MEH in crypto markets. In addition, it reveals the potential of forecast improvement by adding extra information and conducting clustering before building forecasting models.

2 Relevant Literature

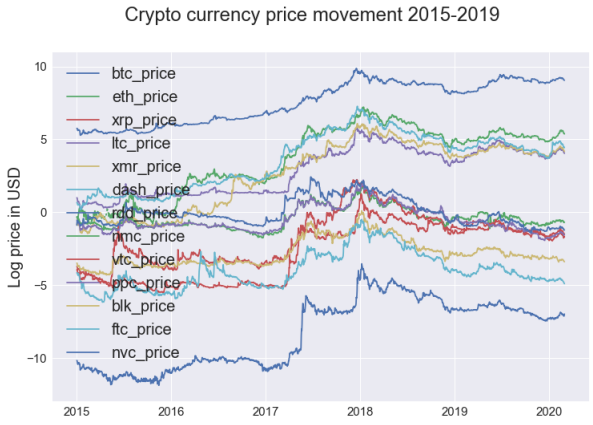
Pioneer research papers on using machine learning as a forecasting tool primarily come from studies focusing on the equity market. Huang et al. [HNW04] compare a model based on Support Vector Machine (SVM) and the random walk model in predicting the movement direction of the NIKKEI 225 index. The paper concludes that the SVM model makes more accurate predictions compared to the traditional random walk model. Enke and Thawornwong [ET05] and Sheta et

al. [SAF15] both demonstrate that the strategies guided by neural network classification models generate higher profits under the same risk exposure than the traditional linear regression models.

In the area of crypto-related research, existing literature has explored many different ways of forecasting cryptocurrency price movements. Alessandretti et al. [AEAB18] build their cryptocurrency forecasting algorithms based on XGBoost and compare the algorithms with a baseline strategy of taking the moving average of prices in the past few days. The paper attempts to build a crypto portfolio that is based on the predictions. The portfolio outperformed the baseline strategy on a daily basis and during the whole period considered in the paper. Rebane et al. [RKDP18] apply a recurrent neural network (RNN) approach to analyze Bitcoin time series price data. The results confirm that the RNN method may improve the classical autoregressive integrated moving average (ARIMA) model since RNN generates more accurate predictions.

3 Data

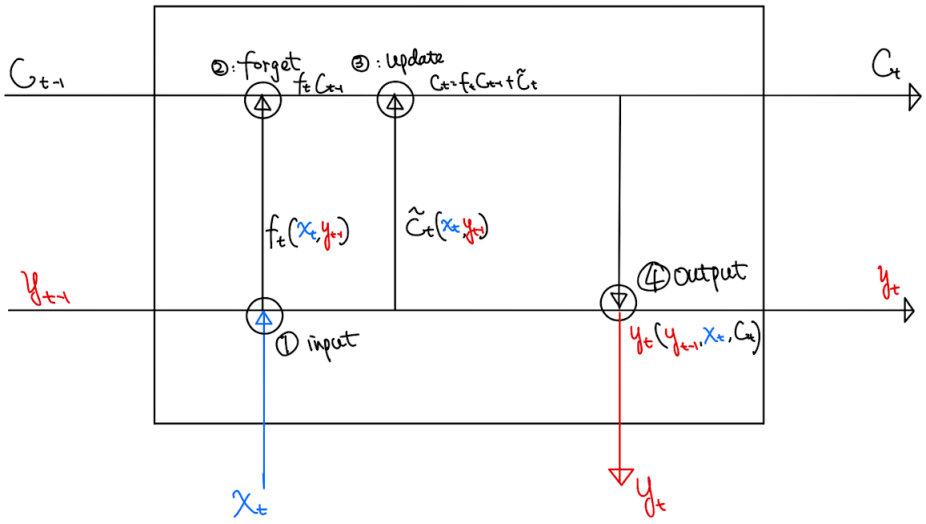
Our project examines historical daily trading details, including the prices, trading volume, and market capitalization of 13 major cryptocurrencies from January 2015 to February 2020. The cryptocoins in our data are Bitcoin, Ether, Ripple(XRP), Litecoin, Monero(XMR), Dash, Namecoin, Peercoin, Blackcoin, Novacoin, Reddcoin, Vertcoin, and Feathercoin. Later we will see that in order to make legitimate forecasts, we must transform prices into returns as model inputs. All data come from coinmarketcap.com, a leading cryptocurrency market information provider. The graph below displays the time series of all coins in our sample with log transformation. From the graph below we can tell that compared to other currencies, Bitcoin prices are much higher. Also their prices fluctuations seem to be correlated to some extent.



4 Methodology

For time series forecasts, we rely on a specific type of recurrent neural network (RNN) called Long Short Time Memory (LSTM). RNN allows previous outputs to be used as inputs while having hidden states. On top of that, LSTM Network makes an improvement on "gradient vanishing" problems compared with ordinary RNNs. That means, when the prediction output depends on not only recent inputs, but also distant inputs, LSTM is good at utilizing the information contained in distant past inputs. If there is a "long-term dependence" problem, then LSTM can excel ordinary RNN in predictions. LSTM is well-suited to making predictions based on time series data, since there can be lags of unknown duration between important events.

Generally, LSTMs can utilize distant past inputs' information to make predictions because of its 4-step procedure to process both past outputs and current inputs, i.e., a 4-layer structure suggested by [Ola15]. The four steps are to "forget", "input", "update", "output". See below a hand-drawn illustration.



LSTM is constructed by a series of boxes aligned horizontally in a time-series manner. The top line in the box is called a cell, a vector that keeps useful inputs. First, the previous output y_{t-1} and input x_t are used in the first step to generate a vector f_t that dictates the forgetting process. Second, the previous cell vector is modified according to f_t. Third, the modified cell plus an update vector $\tilde{C}_t$, which is generated from the input and the previous output, became the updated cell. Finally, an output is generated according to the updated cell, the current input and the previous output.

In our project, the cell can contain useful past return data, which are examined and modified each time a new data is observed. We then make predictions using both the cell and the new input data. The four steps can involve transformations using sigmoid function and tanh function.

GARCH stands for Generalized Autoregressive Conditional Heteroscedasticity. It is a time series model that captures two natures of the data generating process: that it is autoregressive, and that it is heteroscedastic, i.e., though the error term has a 0 mean, its variance is not a constant. It is widely used to estimate the volatility of returns of stocks, which augments the prediction of stock returns. It is acknowledged as one of the standard approaches to model returns, for which we make it a benchmark to compare.

Four hyperparameters specify a GARCH model. In our context, a GARCH model specifies returns as a autoregressive moving average process including p lagged values and q moving averages, i.e., an ARMA(p_1, q_1) process:

$$r_t = c + \sum_{i=1}^{p_1} \phi_i r_{t-i} + \sum_{j=1}^{q_1} \theta_j \epsilon_{t-j} + \epsilon_t$$

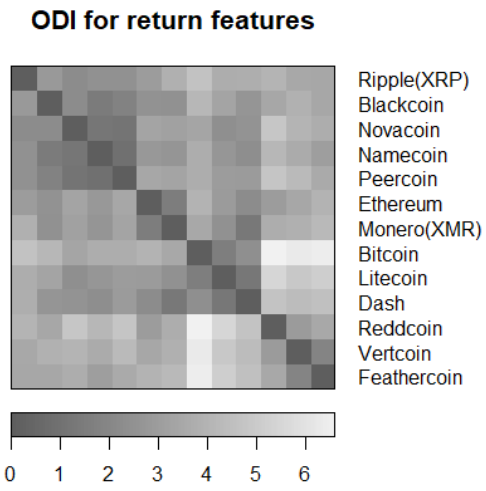
The variance of the error term ϵ_t, σ_t^2 in the above ARMA process is then modeled as another ARMA(p_2, q_2) process:

$$\sigma_t^2 = \omega + \sum_{i=1}^{p_2} \alpha_i \sigma_{t-i}^2 + \sum_{j=1}^{q_2} \theta_j \epsilon_{t-j}^2$$

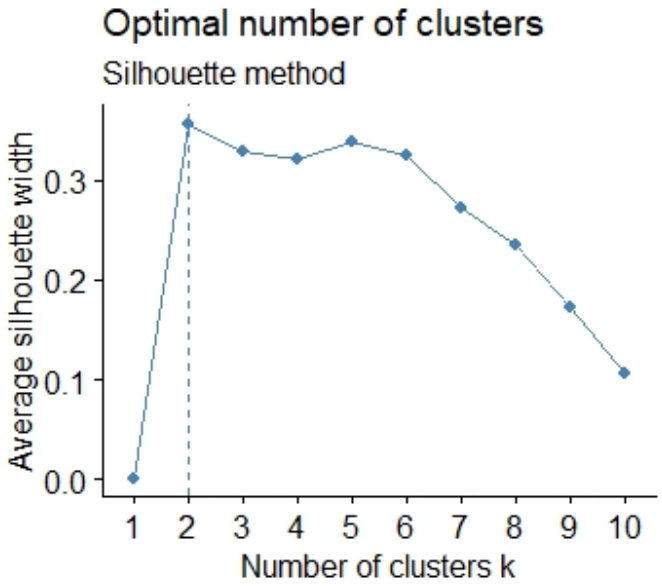
5 Workflow - Clustering & PCA

We do hierarchical agglomerative clustering, k-means clustering, and principal component analysis to get a sense of similarities and dissimilarities among coins by price and return features. Price features include standardized price mean, standard deviation ,25th percentile, 50th percentile, 75th percentile, average market capital, and average trading volume. Return features include standardized return mean, standard deviation, 25th percentile, 50th percentile, 75th percentile, and return zero rates, which indicates the degree of trading activity.

First, we cluster by return features. Before clustering, we diagnose clusterability to investigate the feature space. The graph below is the Ordered Dissimilarity Image for return features, which visualizes the dissimilarity matrix and display the spatially proximate observations in consecutive order. Darker blocks along the diagonal indicate greater spatial similarity, and lighter shaded blocks reflect greater dissimilarity. ODI suggests two or three clusters.

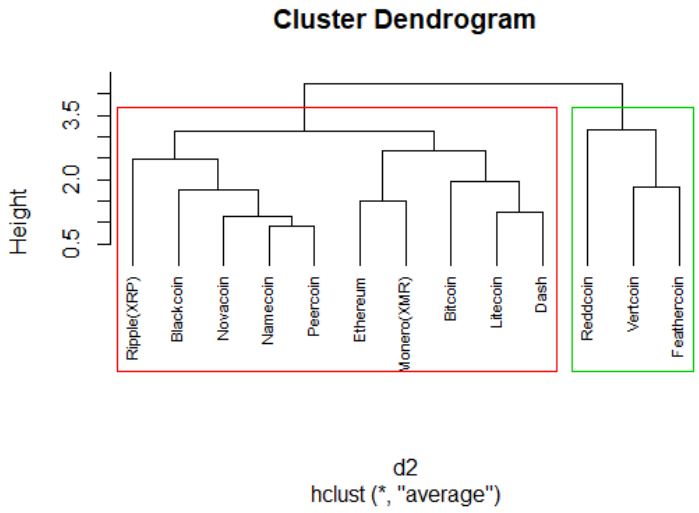


Then we use the Silhouette method to determine the optimal number of cluster, which is 2. Average Silhouette width evaluates clustering validity across values of k, and a high silhouette width indicates that observations are well clustered.

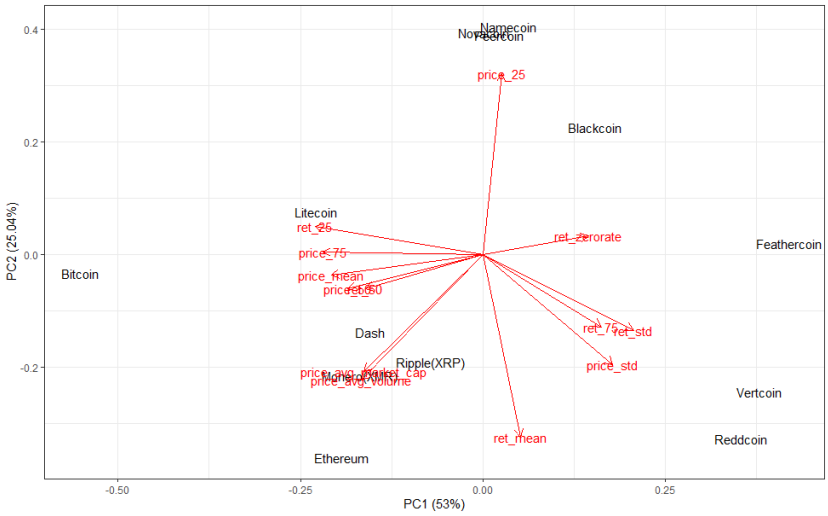


We divide the coins into two clusters by HAC and k-means. In HAC, each observation is regarded as a single cluster and joined with the spatially closest observation. This cluster is then fused with other similar observations pairwisely until there are k clusters. While HAC is in a pairwise fashion, K-means clustering partitions observations into k clusters that cover the full set of data points and are non-overlapping. The cluster is centered around a calculated mean to maximize intra-cluster homogeneity and maximize inter-cluster heterogeneity. Clustering by return features, HAC and k-means give the same results: Reddcoin, Vertcoin, Feathercoin belong to cluster 1, and

Bitcoin, Ethereum, Ripple(XRP), Litecoin, Monero(XMR), Dash, Namecoin, Peercoin, Blackcoin, Novacoin belong to cluster 2.



Turn to principal component analysis to figure out how coins resemble when including all price and return features. PCA converts a set of features to a set of principal components, which are orthogonal and uncorrelated linear combinations across features. The first component accounts for the most variance, and the second component accounts for the second greatest variance and orthogonal to the first component. Here the first PC explains 53% of the variance and the second PC explains 25%. The PCA plot indicates that coins at the top right (Namecoin, Peercoin, Blackcoin, Novacoin) have high 25th price percentile and low return mean; coins at the bottom right (Reddcoin, Vertcoin, Feathercoin) have high return zero rate price and return standard deviation, which indicates that they are inactive in trading and volatile respectively; coins at the bottom left (Bitcoin, Ether, Ripple, Litecoin, Monero, Dash) have high price features including average trading volume and average market capital, which imply that they are active cryptocurrencies. Additionally, When using all price features and price and return features to cluster, the optimal three clusters from HAC and k-means are the same as what the PCA plot shows.



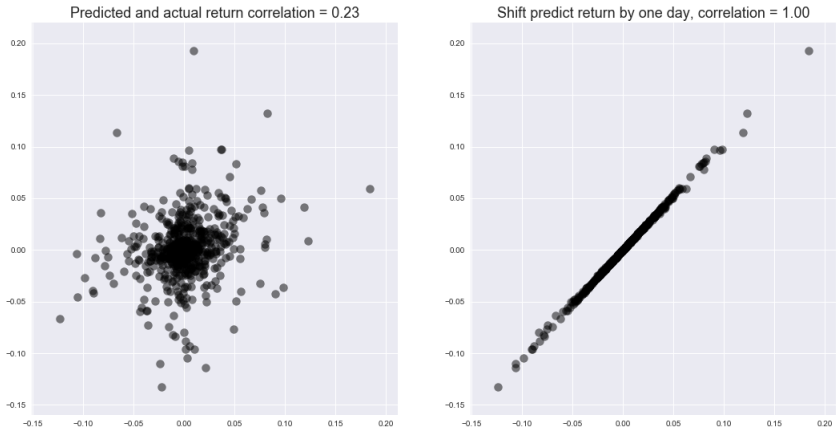
6 Workflow–LSTM

Using Prices vs. Returns as Inputs

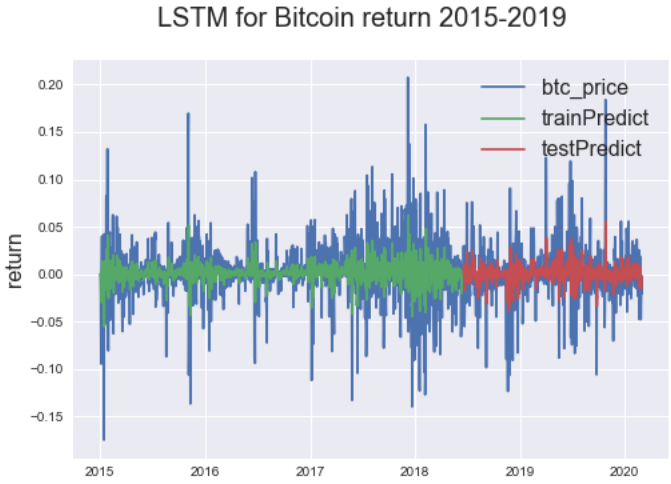
Although machine learning is promising in capturing nonlinear dynamics in the data, it is still crucial to fit the model with legitimate inputs. When we first started our project, we learned our lesson from making forecasts of Bitcoin price directly using its past prices. Since Bitcoin is the dominating coin in the cryptocurrency markets, we use it as an example to conduct a preliminary test on our hypothesis. The Bitcoin price forecasts from LSTM along with its actual values between 2015-2019, are presented in the graph below, which just looks too good to be true.

With 100 times of iteration, we reached a train score of 375.74 RMSE, and a test score of

359.23 RMSE. However, it will be misleading if we simply focus on RMSE without taking into account the nature of the input data. In both theory and empirical studies, it is stated that price time series approximately follow a random walk, which makes it almost impossible for people from making successful forecasts about future returns. If we insist on using prices as our inputs, then essentially what we get is a "shifted" actual value as our forecasts. To better specify the spurious forecast model, we can transform both the actual and forecast prices into returns. The scatter plot on the left shows that these two returns barely match. Whereas if we shift the forecast prices by one day, then the pattern looks almost identical, suggesting that the price-based forecast with LSTM delivers spurious overfitting.



In order to turn this into a forecastable problem (i.e., to find a weakly stationary process), we transform the price series into return series. Our assumption is that there may exist an autoregressive pattern in the return movements, as well as an autocorrelation pattern in return volatility, which suggests that we can try to develop a learning model that captures these nonlinear effects. The graph below presents our forecast result using Bitcoin's daily returns. Apparently, compared with the "spurious precision" of price forecast, the return forecast largely underestimates the scale of expected returns. Nevertheless, we can still get some sense of the forecast such as the direction and relative scale. For the rest of the project, we will stick to return-based forecasts and discuss key issues we face in model construction and validation.



Concerning our return-based forecasting process, there are a few main questions we must address. Firstly, how to tune hyperparameters? Secondly, how to evaluate the prediction? And most importantly, what does the result tell about EMH? Lastly, is there any extension we can make to improve forecasts? Next, we will be discussing these main issues one by one and provide concrete explanations to our findings.

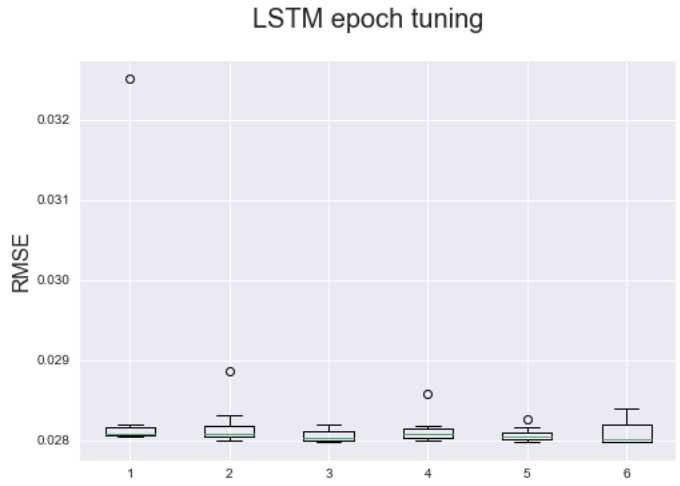
Modeling Fitting and Parameter Tuning

In order to find the hyperparameters that generate smaller RMSEs, we tune three of our parameters: batch size, the number of epochs, and the number of neurons in each layer. In the process of tuning, we hold other parameters constant, and tune one parameter each time. Without losing generality and introducing complexity, we use least square error as our loss function criteria and 'adam' as our optimizer, with a default learning rate of 0.1%.

Note that if we tune multiple parameters together without holding any one of them constant, we may find the most optimal parameters across all possible combinations of the hyperparameters. But it would take hours to run the tuning process, as we have a large data set. Therefore, to save more time, we could only hold other hyperparameters constant when tuning one of them. If we had more time, we would extend this optimization process to other hyperparameters such as the learning rate and the number of hidden layers. Last but not least, according to our test, there is not much difference between tuning based on Bitcoin and other cryptocurrencies, so we decide to showcase the process of tuning in the case of Bitcoin.

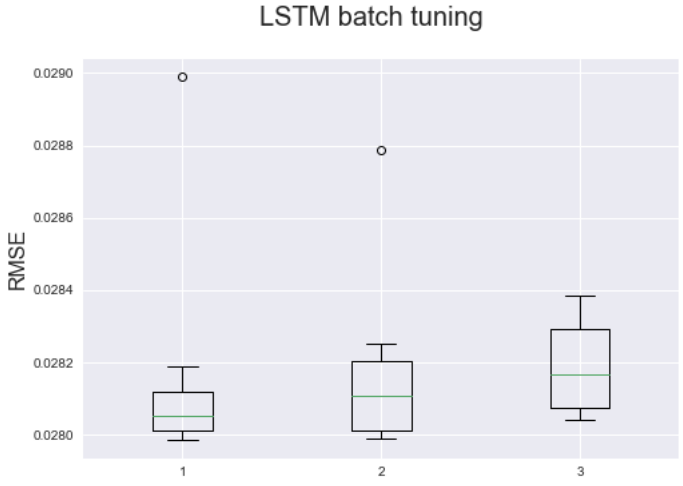
(1) Epochs represents the number of complete passes through the training data set. We compare

the model performance for epochs ranging from 1 to 5. (We hold batch = 1 and neuron = 1)



The average RMSE across all epochs is about the same, but we get the most consistent RMSE when epoch is 3. The average RMSE floats around 0.028, which means that our forecast is 2.8% away from the real return.

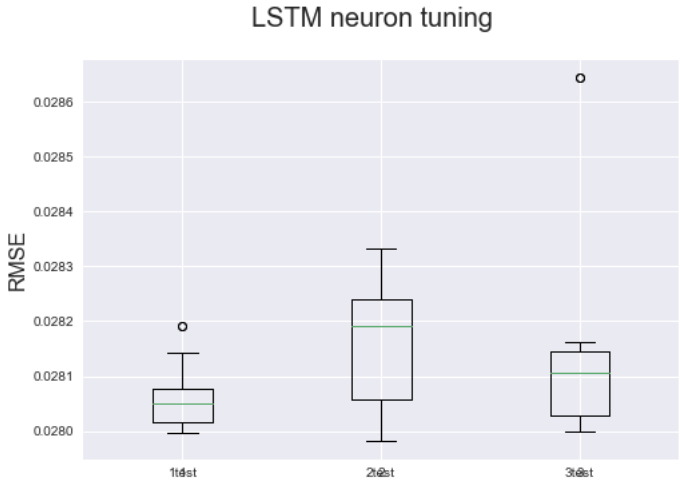
(2) Batch size is the subset size of your training sample that is going to be used in order to train the network during its learning process. Each batch trains the network in successive order, taking into account the updated weights coming from the training results of the previous batches. (We hold epochs = 3 and neuron = 1)



Batch size 1 generates the lowest mean RMSE and a relatively more consistent RMSE distri-

bution.

(3) Lastly, we tune the number of neurons in each hidden layer. (We hold epochs = 3 and batch = 1)

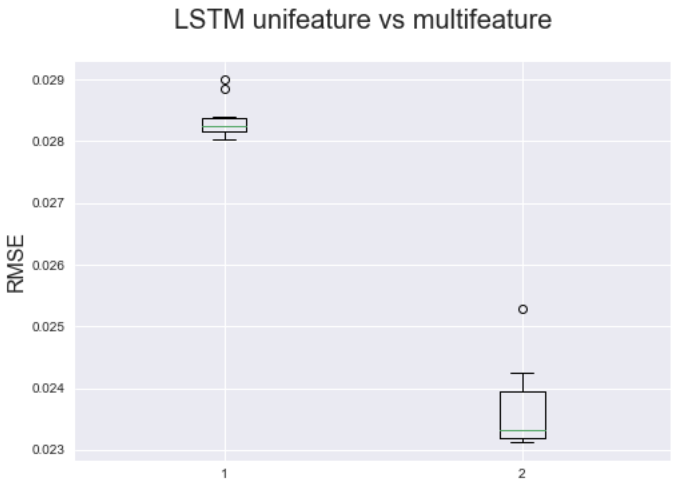


Having only one neuron gives the lowest RMSE mean and a relatively more consistent RMSE distribution.

Note that in tunning the three parameters, if we look over a broader range of possible parameter values, our current optimized solution may not generate the lowest and the most consistent RMSEs anymore. But to avoid waiting hours for the codes to run, we only look at small parameters values.

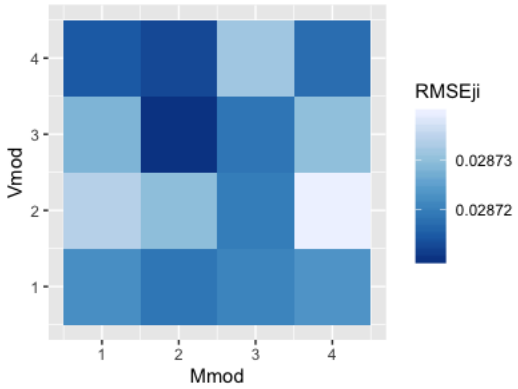
Multivariate v.s. Univariate

LSTM is highly flexible in terms of including additional features in the training process. Instead of using just past return information of Bitcoin, we also test the performance of our LSTM model after including past return information of a synthetic cryptocurrency index called CCI30. Under a simple holdout method, we conclude that additional features can indeed improve the ability to forecast compared to the univariate inputs. Below are the model RMSEs, with the axis being the number of variables included.



7 Workflow - GARCH

Though there are other theory-based approaches to determine parameters in a GARCH model, to be consistent with our tuning process of the LSTM model, we use cross validation to find the best GARCH model to predict returns. To be specific, we let $p_1 = q_1$ and $p_2 = q_2$ for computational convenience, and look for $p_1(q_1)$ and $p_2(q_2)$ that minimize test set RMSE. The result is illustrated by a heat map:



Telling from the figure, the GARCH model performs the best when $p_1 = q_1 = 2$ and $p_2 = q_2 = 3$.

8 Model Comparisons

Discussion of Data Properties and Model Comparability

As is discussed in the spurious price-based case, we are careful with dealing with time dependent characteristics in our input. This is the main reason why we transform prices into returns. Compared to a normal distribution, return time series are left-skewed with fatter tails. The Returns has important self-autoregression features in both mean and variance. Therefore, we propose a highly comparable comparison between LSTM and GARCH, which are both capable of capturing time-varying volatility in the inputs. Especially for finance time series modeling, GARCH has been serving as a mainstream tool for many years. This comparison provides us crucial evidence on how precise machine learning techniques can be, compared with a traditional benchmark model. More specifically, time is modeled in both models as the sequence of input data. Time steps consist of a dimension of our three-dimensional array. Forecasts are made only based on past returns. In other words, assuming returns are weakly stationary, then there will not be time dependent issues such as trend.

To compare how LSTM and GARCH perform on Bitcoin data as well as data of other coins, we introduce three ways of cross validation in time series settings. They have different implications for the models' fit.

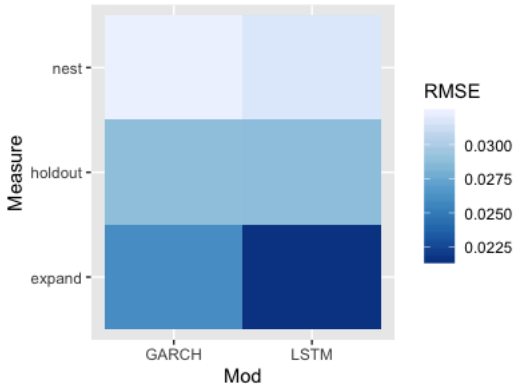
Cross Validation

The first cross validating approach is simply holding out the last 1/3 of our data as the test set. We calculate the RMSEs using the test set.

The nested cross validation approach cuts the data series into nests, and then each nest(excluding the first nest) are used as test set to calculate errors, where the training set is all data before the nest. The overall RMSEs calculate from errors we obtained from all nests but the first one. In our project, we specify the number of nests to be 5.

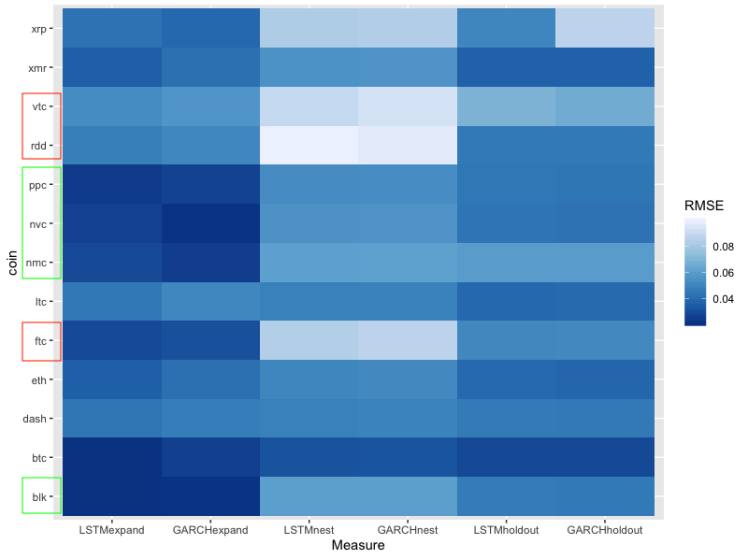
The expanding window cross validation aligns the best with real world scenario. Each time a forecast is made using all previous data as the training set, and then the error is calculated accordingly. In our project, due to computational resource constraints, we only make forecasts on the last ten days of our time frame in the expanding window manner, and then calculate RMSE based on these forecasts and corresponding actual values.

Comparison on Bitcoin Data



In this figure, the darker the color, the better the model fits measured by the corresponding cross validation. We see LSTM outperforms GARCH no matter which cross validation we use. This suggests that LSTM may be able to make better short term and long term forecasts than GARCH.

Comparison on All Coins



Some other interesting observations emerge when we use cross validations to evaluate both models' performance on all 13 coins' data. Combined with our PCA clustering result, we see the red cluster (Reddcoin, Vertcoin, Feathercoin) is more unforecastable in the nested world. The green cluster (Namecoin, Peercoin, Blackcoin, Novacoin) are coins where LSTM and GARCH do

not perform significantly differently. The other coins belong to the same cluster.

We are cautious about making a strong conclusion here, because due to limited time and computational resources, we only tune both models on Bitcoin data. This is also observable in the figure as Bitcoin's row is overall darker than other rows.

9 Conclusions

In this project, we adopt different machine learning methods to study the returns of selected cryptocurrencies. We use clustering techniques and PCA to detect similar patterns among the currencies' trading features. Then we apply LSTM models with past returns as inputs to generate forecasts for future returns. We also consider GARCH as our benchmark, a traditional time-series econometric regression approach to forecast returns. We compare the LSTM model with the GARCH model. LSTM outperforms GARCH on Bitcoin data. Different PCA clusters of coins demonstrate different patterns of forecastability, which is also interesting and suggestive of future investigation.

Finally, our results imply that the cryptocurrency markets satisfy the weak form market efficiency hypothesis. As our models in general produce over 2% RMSEs, this implies that using only past returns is insufficient for generating accurate future return forecasts. There is no chance to make money out of the markets using past returns as predictors.

References

[AEAB18] Laura Alessandretti, Abeer ElBahrawy, Luca Maria Aiello, and Andrea Baronchelli. Anticipating cryptocurrency prices using machine learning. *Complexity*, 2018.

[ET05] David Enke and Suraphan Thawornwong. The use of data mining and neural networks for forecasting stock market returns. *Expert Systems with Applications*, 2005.

[HNW04] Wei Huang, Yoshiteru Nakamoria, and Shou-Yang Wang. Forecasting stock market movement direction with support vector machine. *Computers Operations Research*, 2004.

[Ola15] Christopher Olah. *Understanding LSTM Networks*, 2015. https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[RKDP18] Jonathan Rebane, Isak Karlsson, Stojan Denic, and Panagiotis Papapetrou. Seq2seq rnns and arima models for cryptocurrency prediction: A comparative study. *SIGKDD Fintech*, 2018.

[SAF15] Alaa F. Sheta, Sara Elsir M. Ahmed, and Hossam Faris. A comparison between regression, artificial neural networks and support vector machines for predicting stock market index. *Soft Computing*, 2015.

Projectcode

Ying Zhou

2020/3/12

Clustering & PCA

```
pft<-read.csv("C:/Users/zhouy/Desktop/Uchicago/uchicourse/Intro to Machine Learning/project/coin_feature
# First, load a few libraries
library(tidyverse)
library(skimr)
library(dendextend) # for "cutree" function
df<-as.data.frame(pft)
rownames(df) <- df$name
#scale whole
df_sub <- df%>%
  select(price_mean,price_std,price_25, price_50,   price_75,   price_avg_market_cap,   price_avg_volume
  scale()
#scale price return features
df_sub1 <- df%>%
  select(price_mean,price_std,price_25, price_50,   price_75,   price_avg_market_cap,   price_avg_volume
  scale()
df_sub2 <- df%>%
  select(ret_mean,ret_std,ret_25,   ret_50, ret_75, ret_zerorate) %>%
  scale()

#ODI
library("seriation")
df_dist <- dist(df_sub,method = "euclidean")
dissplot(df_dist,method = "HC_average",options = list(main = "ODI for price and return features"))

df_dist1 <- dist(df_sub1,method = "euclidean")
dissplot(df_dist1,method = "HC_average",options = list(main = "ODI for price features"))

df_dist2 <- dist(df_sub2,method = "euclidean")
dissplot(df_dist2,method = "HC_average",options = list(main = "ODI for return features"))

#Silhouette method
fviz_nbclust(df_sub, FUN = hcut, method = "silhouette")+
  labs(subtitle = "Silhouette method")
fviz_nbclust(df_sub1, FUN = hcut, method = "silhouette")+
  labs(subtitle = "Silhouette method")

fviz_nbclust(df_sub2, FUN = hcut, method = "silhouette")+
  labs(subtitle = "Silhouette method")
```

```
#HAC
library(tidyverse)
library(skimr)
library(dendextend)
d <- dist(df_sub, method = "euclidean")
hc <- hclust(d, method = "average" )
grp <- cutree(hc, k = 3)
plot(hc, cex = 0.7, hang = -1)
rect.hclust(hc, k = 3, border = 2:5)

d1 <- dist(df_sub1, method = "euclidean")
hc1 <- hclust(d1, method = "average" )
grp1 <- cutree(hc1, k = 3)
plot(hc1, cex = 0.7, hang = -1)
rect.hclust(hc1, k = 3, border = 2:5)

d2 <- dist(df_sub2, method = "euclidean")
hc2 <- hclust(d2, method = "average" )
grp2 <- cutree(hc2, k = 2)
plot(hc2, cex = 0.7, hang = -1)
rect.hclust(hc2, k = 2, border = 2:6)


#k-means
set.seed(1234)
kmeans<- kmeans(df_sub,
                centers = 3,
                iter.max = 15)
str(kmeans)
kc <- as.table(kmeans$cluster)
kc <- data.frame(kc)
colnames(kc)[colnames(kc)=="Freq"] <- "km_assign"
colnames(kc)[colnames(kc)=="Var1"] <- "coin"
rownames(kc) <- kc$coin
kc

kmeans1<- kmeans(df_sub1,
                centers = 3,
                iter.max = 15)
str(kmeans1)


kc1 <- as.table(kmeans1$cluster)
kc1 <- data.frame(kc1)
colnames(kc1)[colnames(kc1)=="Freq"] <- "km_assign"
colnames(kc1)[colnames(kc1)=="Var1"] <- "coin"
rownames(kc1) <- kc1$coin
kc1

kmeans2<- kmeans(df_sub2,
                centers = 2,
                iter.max = 15)
str(kmeans2)
kc2 <- as.table(kmeans2$cluster)
```

```
kc2 <- data.frame(kc2)
colnames(kc2)[colnames(kc2)=="Freq"] <- "km_assign"
colnames(kc2)[colnames(kc2)=="Var1"] <- "coin"
rownames(kc2) <- kc2$coin
kc2

#PCA
library(tidyverse)
library(ggfortify)
leg_fit <- prcomp(df_sub); summary(df_sub) # no need to scale because we standardized earlier
loadings <- leg_fit$rotation; head(loadings)
scores <- leg_fit$x; head(scores)
# Biplot
autoplot(leg_fit,
         shape = FALSE,
         loadings.label = TRUE) +
  theme_bw()
arrest_fit <- prcomp(df_sub,
                     scale = TRUE)

summary(arrest_fit)
```

- How to Tune LSTM Hyperparameters with Keras for Time Series Forecasting https://machinelearningmastery.com/tune-lstm-hyperparameters-keras-time-series-forecasting/
- Time Series Prediction with LSTM Recurrent Neural Networks in Python with Keras https://machinelearningmastery.com/time-series-prediction-lstm-recurrent-neural-networks-python-keras/
- Simple Guide to Hyperparameter Tuning in Neural Networks https://towardsdatascience.com/simple-guide-to-hyperparameter-tuning-in-neural-networks-3fe03dad8594
- Stateful and Stateless LSTM for Time Series Forecasting with Python https://machinelearningmastery.com/stateful-stateless-lstm-time-series-forecasting-python/

In [1]:

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from pandas import datetime
```

In [2]:

```
# date-time parsing function for loading the dataset
def parser(x):
 return datetime.strptime(x, '%Y-%m-%d')
returninfo=pd.read_csv('returninfo.csv', parse_dates=[0], index_col=0, date_parser=parser)
# returninfo=pd.read_csv('returninfo.csv',index_col=0)
```

In [ ]:

```
# date-time parsing function for loading the dataset
def parser(x):
 return datetime.strptime(x, '%Y-%m-%d')
returninfo=pd.read_csv('priceinfo.csv', parse_dates=[0], index_col=0, date_parser=parser)
# returninfo=pd.read_csv('returninfo.csv',index_col=0)
```

In [3]:

```
cci30=pd.read_csv('cci30_OHLCV.csv',parse_dates=[0], index_col=0, date_parser=parser)
cci30=cci30.iloc[::-1][['Close']].pct_change()
cci30=cci30[cci30.index<=datetime.strptime('2020-03-01', '%Y-%m-%d')]
cci30=cci30.rename(columns={'Close':'index_return'})
cci30=cci30.reset_index()
cci30=cci30.rename(columns={'Date':'date'}).set_index('date')
```

In [48]:

```
dataframe = cci30.merge(returninfo[['btc_price']], how='inner', left_index=True, right_index=True).fill
na(0)
```

Construct index return information

In [4]:

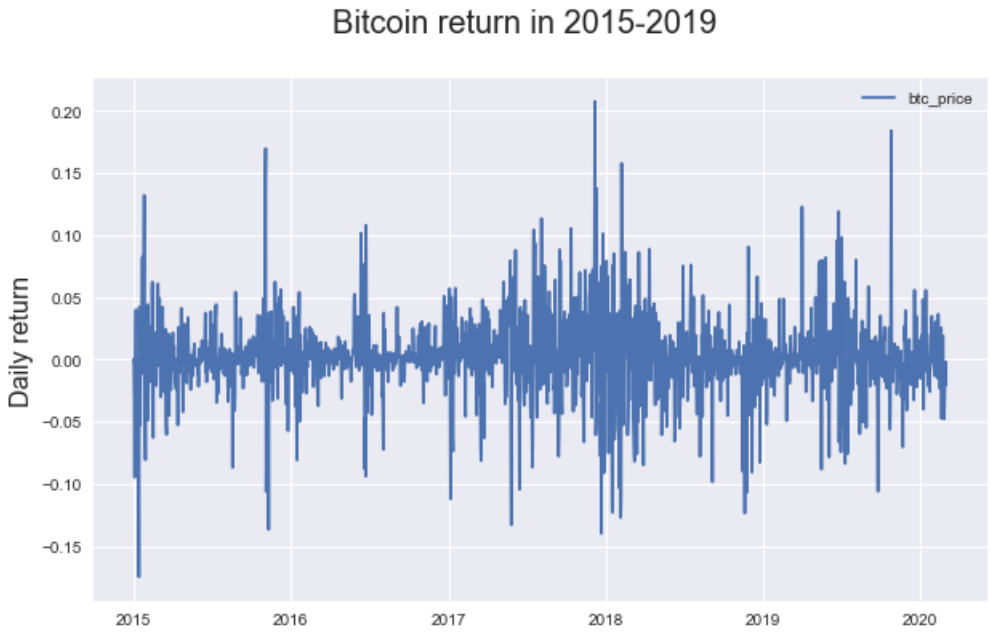
```
# preliminary look
plt.style.use('seaborn')

plt.figure(figsize=(10,6))
plt.plot(returninfo['btc_price'])
plt.suptitle('Bitcoin return in 2015-2019', fontsize=20)
plt.ylabel('Daily return', fontsize=16)
plt.legend(returninfo.columns)
# plt.savefig('output//crypto_return.png')
```

```
C:\Users\WENTWORTHLIU\AppData\Roaming\Python\Python37\site-packages\pandas\plotting\_converter.py:129:
```

```
FutureWarning: Using an implicitly registered datetime converter for a matplotlib plotting method. The
converter was registered by pandas on import. Future versions of pandas will require you to explicitly
register matplotlib converters.

To register the converters:
 >>> from pandas.plotting import register_matplotlib_converters
 >>> register_matplotlib_converters()
  warnings.warn(msg, FutureWarning)
```

Out[4]:

```
<matplotlib.legend.Legend at 0x197c89d9bc8>
```



Test multiple factors

In [162]:

```
# LSTM for cryptal currency problem with regression framing
import numpy
import matplotlib.pyplot as plt
from pandas import read_csv
import math
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
 dataX, dataY = [], []
 for i in range(len(dataset)-look_back-1):
  a = dataset[i:(i+look_back), :]
  dataX.append(a)
  dataY.append(dataset[i + look_back, 0])
 return numpy.array(dataX), numpy.array(dataY)
# fix random seed for reproducibility
numpy.random.seed(7)
# load the dataset
# dataframe = returninfo[['btc_price']]
dataframe = returninfo[['btc_price']].merge(cci30, how='inner', left_index=True, right_index=True).fill
na(0)
dataset = dataframe.values
dataset = dataset.astype('float32')
# normalize the dataset
scaler0 = MinMaxScaler(feature_range=(0, 1))
scaler1 = MinMaxScaler(feature_range=(0, 1))
dataset[:,0] = scaler0.fit_transform(numpy.reshape(dataset[:,0],(-1,1)))[:,0]
dataset[:,1] = scaler1.fit_transform(numpy.reshape(dataset[:,1],(-1,1)))[:,0]
```

```
# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
# reshape into X=t and Y=t+1
look_back = 1
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
# reshape input to be [samples, time steps, features]
trainX = numpy.reshape(trainX, (trainX.shape[0], 1, 2))
testX = numpy.reshape(testX, (testX.shape[0], 1, 2))
# create and fit the LSTM network
model = Sequential()
model.add(LSTM(1, input_shape=(look_back,2)))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')

nb_repeat = 10
nb_epoch = 3
batch_size=1
nb_neurons=1
train_rmse, test_rmse = list(), list()
for r in range(nb_repeat):

    trainX, trainY = create_dataset(train, look_back)
    testX, testY = create_dataset(test, look_back)
    # reshape input to be [samples, time steps, features]
    trainX = numpy.reshape(trainX, (trainX.shape[0], 1, 2))
    testX = numpy.reshape(testX, (testX.shape[0], 1, 2))
    model.fit(trainX, trainY, epochs=nb_epoch, batch_size=batch_size, verbose=0)
    # make predictions
    trainPredict = model.predict(trainX)
    testPredict = model.predict(testX)
    # invert predictions
    trainPredict = scaler0.inverse_transform(trainPredict)
    trainY = scaler0.inverse_transform([trainY])
    testPredict = scaler0.inverse_transform(testPredict)
    testY = scaler0.inverse_transform([testY])
    # calculate root mean squared error
    trainScore = math.sqrt(mean_squared_error(trainY[0], trainPredict[:,0]))
    print('Train Score: %.4f RMSE' % (trainScore))
    train_rmse.append(trainScore)
    testScore = math.sqrt(mean_squared_error(testY[0], testPredict[:,0]))
    print('Test Score: %.4f RMSE' % (testScore))
    test_rmse.append(testScore)

historyindex = DataFrame()
historyindex['train'], historyindex['test'] = train_rmse, test_rmse
# # shift train predictions for plotting
# trainPredictPlot = numpy.empty_like(numpy.reshape(dataset[:,0],(-1,1)))
# trainPredictPlot[:, :] = numpy.nan
# trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# # shift test predictions for plotting
# testPredictPlot = numpy.empty_like(numpy.reshape(dataset[:,0],(-1,1)))
# testPredictPlot[:, :] = numpy.nan
# testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
# # plot baseline and predictions
# plt.style.use('seaborn')
# plt.plot(returninfo.index.values, scaler0.inverse_transform(numpy.reshape(dataset[:,0],(-1,1))),label
='btc_price')
# plt.plot(returninfo.index.values,trainPredictPlot, label='trainPredict')
# plt.plot(returninfo.index.values,testPredictPlot, label='testPredict')
# plt.figsize=(5.6,4)
# plt.suptitle('LSTM for Bitcoin return with additional feature 2015-2019', fontsize=20)
# plt.ylabel('return', fontsize=16)
# plt.legend(fontsize=16)
# plt.savefig('output//Bitcoin_return_LSTM_INDEX.png')

# ###
# #total RMSE
# ###
# Ypredict = np.concatenate([trainPredict[:,0],testPredict[:,0]])
# Y = np.concatenate([trainY[0],testY[0]])
# math.sqrt(mean_squared_error(Y,Ypredict))
```

```
Train Score: 0.0302 RMSE
```

```
Test Score: 0.0253 RMSE
Train Score: 0.0293 RMSE
Test Score: 0.0237 RMSE
Train Score: 0.0290 RMSE
Test Score: 0.0233 RMSE
Train Score: 0.0290 RMSE
Test Score: 0.0232 RMSE
Train Score: 0.0289 RMSE
Test Score: 0.0231 RMSE
Train Score: 0.0290 RMSE
Test Score: 0.0231 RMSE
Train Score: 0.0297 RMSE
Test Score: 0.0243 RMSE
Train Score: 0.0290 RMSE
Test Score: 0.0232 RMSE
Train Score: 0.0298 RMSE
Test Score: 0.0240 RMSE
Train Score: 0.0291 RMSE
Test Score: 0.0233 RMSE
```

In [167]:

```
merged_his=history.merge(historyindex, how='inner', left_index=True, right_index=True)
plt.boxplot(merged_his[[x for x in merged_his.columns.values if x[:4]=='test']].T)
plt.suptitle('LSTM unifeature vs multifeature', fontsize=20)
plt.ylabel('RMSE', fontsize=16)
# plt.legend(fontsize=16)
plt.savefig('output//Bitcoin_return_feature_LSTM.png')
```

LSTM

In [24]:

```
# LSTM for cryptal currency problem with regression framing
import numpy
import matplotlib.pyplot as plt
from pandas import read_csv
import math
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
from pandas import DataFrame
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
 dataX, dataY = [], []
 for i in range(len(dataset)-look_back-1):
  a = dataset[i:(i+look_back), 0]
```

```
  dataX.append(a)
  dataY.append(dataset[i + look_back, 0])
 return numpy.array(dataX), numpy.array(dataY)

# load the dataset
history = DataFrame()
for coin in returninfo.columns.values:
    dataframe = returninfo[[coin]]
    dataset = dataframe.values
    dataset = dataset.astype('float32')
    # normalize the dataset
    scaler = MinMaxScaler(feature_range=(0, 1))
    dataset = scaler.fit_transform(dataset)
    # split into train and test sets
    train_size = int(len(dataset) * 0.67)
    test_size = len(dataset) - train_size
    train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
    # reshape into X=t and Y=t+1
    look_back = 1
    trainX, trainY = create_dataset(train, look_back)
    testX, testY = create_dataset(test, look_back)
    # reshape input to be [samples, time steps, features]
    trainX = numpy.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
    testX = numpy.reshape(testX, (testX.shape[0], 1, testX.shape[1]))
    # create and fit the LSTM network
    model = Sequential()
    model.add(LSTM(1, input_shape=(1, look_back)))
    model.add(Dense(1))
    model.compile(loss='mean_squared_error', optimizer='adam')

    # fit model

    nb_repeat = 10
    nb_epoch = 3
    batch_size=1
    nb_neurons=1
    train_rmse, test_rmse = list(), list()
    for r in range(nb_repeat):

        trainX, trainY = create_dataset(train, look_back)
        testX, testY = create_dataset(test, look_back)
        # reshape input to be [samples, time steps, features]
        trainX = numpy.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
        testX = numpy.reshape(testX, (testX.shape[0], 1, testX.shape[1]))
        model.fit(trainX, trainY, epochs=nb_epoch, batch_size=batch_size, verbose=0)
        # make predictions
        trainPredict = model.predict(trainX)
        testPredict = model.predict(testX)
        # invert predictions
        trainPredict = scaler.inverse_transform(trainPredict)
        trainY = scaler.inverse_transform([trainY])
        testPredict = scaler.inverse_transform(testPredict)
        testY = scaler.inverse_transform([testY])
        # calculate root mean squared error
        trainScore = math.sqrt(mean_squared_error(trainY[0], trainPredict[:,0]))
#         print('Train Score: %.4f RMSE' % (trainScore))
        train_rmse.append(trainScore)
        testScore = math.sqrt(mean_squared_error(testY[0], testPredict[:,0]))
#         print('Test Score: %.4f RMSE' % (testScore))
        test_rmse.append(testScore)
    history[coin[:-6]+'train'], history[coin[:-6]+'test'] = train_rmse, test_rmse

# # shift train predictions for plotting
# trainPredictPlot = numpy.empty_like(dataset)
# trainPredictPlot[:, :] = numpy.nan
# trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# # shift test predictions for plotting
# testPredictPlot = numpy.empty_like(dataset)
# testPredictPlot[:, :] = numpy.nan
# testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
# # plot baseline and predictions
# plt.style.use('seaborn')
# plt.plot(returninfo.index.values, scaler.inverse_transform(dataset),label='btc_price')
# plt.plot(returninfo.index.values,trainPredictPlot, label='trainPredict')
# plt.plot(returninfo.index.values,testPredictPlot, label='testPredict')
# plt.figsize=(5.6,4)
# plt.suptitle('LSTM for Bitcoin return 2015-2019', fontsize=20)
```

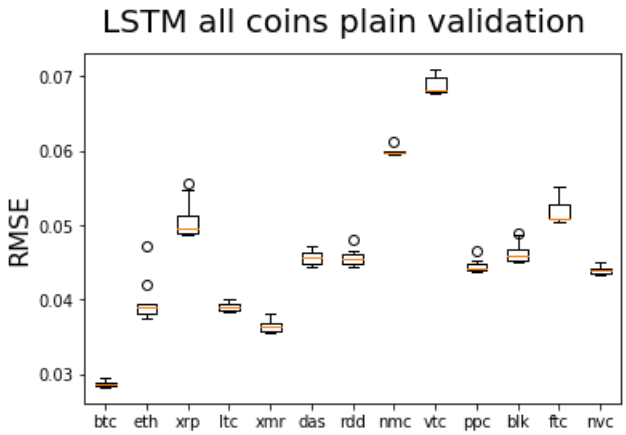
```
# plt.ylabel('return', fontsize=16)
# plt.legend(fontsize=16)
# # plt.savefig('output//Bitcoin_return_LSTM.png')
# ###
# #total RMSE
# ###
# Ypredict = np.concatenate([trainPredict[:,0],testPredict[:,0]])
# Y = np.concatenate([trainY[0],testY[0]])
# math.sqrt(mean_squared_error(Y,Ypredict))
```

In [19]:

```
for coin in returninfo.columns.values[:2]:
    dataframe = returninfo[[coin]]
```

In [34]:

```
history.to_csv('output//plain_validation_RMSE_allcoins.csv')
plt.boxplot(history[[x for x in history.columns.values if x[-4:]=='test']].T,labels=[x[:3] for x in his
tory.columns.values if x[-4:]=='test'])
plt.suptitle('LSTM all coins plain validation', fontsize=20)
plt.ylabel('RMSE', fontsize=16)
# plt.legend(fontsize=16)
plt.savefig('output//plain_validation_RMSE_allcoins.png')
```



In [32]:

```
```

Out[32]:

```
['btc',
 'eth',
 'xrp',
 'ltc',
 'xmr',
 'das',
 'rdd',
 'nmc',
 'vtc',
 'ppc',
 'blk',
 'ftc',
 'nvc']
```

In [10]:

```
totalsse = (np.power(testY,2).sum()+np.power(trainY,2).sum())/(len(testY[0])+len(trainY[0]))
totalsse
# errorsse = np.power(testPredict[:,0],2).sum()/len(testPredict[:,0])
# Rsquared = errorsse/totalsse
```

Out[10]:

```
0.0010217162043197412
```

In [18]:

```
ssetest = numpy.empty_like(dataset)
ssetest[:, :] = numpy.nan
ssetest[look_back:len(trainPredict)+look_back, :] = trainPredict
ssetest[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
ssetest
```

Out[18]:

```
array([[        nan],
       [ 0.00164034],
       [ 0.00026396],
       ...,
       [-0.00871995],
       [-0.00215861],
       [        nan]], dtype=float32)
```

In [36]:

```
len(trainPredict[:,0])+len(testPredict[:,0])
```

In [33]:

```
len(Y)
```

Out[33]:

```
1883
```

Nested

In [3]:

```
# LSTM for cryptal currency problem with regression framing
import numpy
import matplotlib.pyplot as plt
from pandas import read_csv
import math
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
from pandas import DataFrame
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
 dataX, dataY = [], []
 for i in range(len(dataset)-look_back-1):
  a = dataset[i:(i+look_back), 0]
  dataX.append(a)
  dataY.append(dataset[i + look_back, 0])
 return numpy.array(dataX), numpy.array(dataY)
batch_size=1
nb_neurons=1
nb_repeat = 10
nb_epoch = 3
historytotal = DataFrame()
# fix random seed for reproducibility
# numpy.random.seed(7)
# load the dataset
for coin in returninfo.columns.values:
    history = DataFrame()
    dataframe = returninfo[[coin]]
    dataset = dataframe.values
```

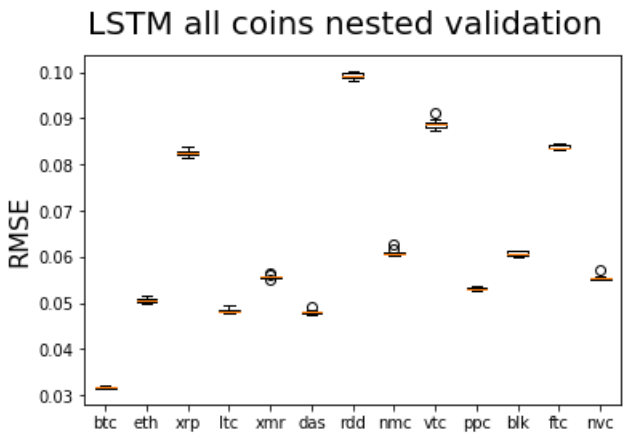
```
    dataset = dataset.astype('float32')
    # normalize the dataset
    scaler = MinMaxScaler(feature_range=(0, 1))
    dataset = scaler.fit_transform(dataset)

    gap = 0.2
    for split in numpy.arange(gap,1,gap):
    # split into train and test sets
        train_size = int(len(dataset) * split)
        test_size = int(len(dataset)*(split+gap)) - train_size
        train, test = dataset[0:train_size,:], dataset[train_size:train_size+test_size,:]
        # reshape into X=t and Y=t+1
        look_back = 1
        trainX, trainY = create_dataset(train, look_back)
        testX, testY = create_dataset(test, look_back)
        # reshape input to be [samples, time steps, features]
        trainX = numpy.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
        testX = numpy.reshape(testX, (testX.shape[0], 1, testX.shape[1]))
        # create and fit the LSTM network
        model = Sequential()
        model.add(LSTM(nb_neurons, input_shape=(1, look_back)))
        model.add(Dense(1))
        model.compile(loss='mean_squared_error', optimizer='adam')

        # fit model
        train_mse, test_mse = list(), list()
        total_train_rmse, total_test_rmse = list(), list()
        for r in range(nb_repeat):
            trainX, trainY = create_dataset(train, look_back)
            testX, testY = create_dataset(test, look_back)
            # reshape input to be [samples, time steps, features]
            trainX = numpy.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
            testX = numpy.reshape(testX, (testX.shape[0], 1, testX.shape[1]))
            model.fit(trainX, trainY, epochs=nb_epoch, batch_size=batch_size, verbose=0)
        #         model.reset_states()

            # make predictions
            trainPredict = model.predict(trainX, batch_size=batch_size)
        #         model.reset_states()
            testPredict = model.predict(testX, batch_size=batch_size)
        #         model.reset_states()
            # invert predictions
            trainPredict = scaler.inverse_transform(trainPredict)
            trainY = scaler.inverse_transform([trainY])
            testPredict = scaler.inverse_transform(testPredict)
            testY = scaler.inverse_transform([testY])
            # calculate root mean squared error
            trainScore = mean_squared_error(trainY[0], trainPredict[:,0])
            train_mse.append(trainScore)
    #         print('Train Score: %.4f RMSE' % (trainScore))
            testScore = mean_squared_error(testY[0], testPredict[:,0])
            test_mse.append(testScore)
    #         print('Test Score: %.4f RMSE' % (testScore))
        history[str(split)+'train'], history[str(split)+'test'] = train_mse, test_mse
#         fordroping = history.columns.values[2:]
    historytotal[coin[:-6]+'train']=np.power(history[[x for x in history.columns.values if x[-5:]=='tra
in']].sum(axis=1)/len(numpy.arange(gap,1,gap)),0.5)
    historytotal[coin[:-6]+'test']=np.power(history[[x for x in history.columns.values if x[-4:]=='test
']].sum(axis=1)/len(numpy.arange(gap,1,gap)),0.5)
#     history=history.drop(columns=fordroping)

#     plt.plot(history['train'], color='blue')
#     plt.plot(history['test'], color='orange')

historytotal.to_csv('output//nested_validation_RMSE_allcoins.csv')
plt.boxplot(historytotal[[x for x in historytotal.columns.values if x[-4:]=='test']].T,labels=[x[:3] fo
r x in historytotal.columns.values if x[-4:]=='test'])
plt.suptitle('LSTM all coins nested validation', fontsize=20)
plt.ylabel('RMSE', fontsize=16)
# plt.legend(fontsize=16)
# plt.savefig('output//nested_validation_RMSE_allcoins.png')
```

```
Using TensorFlow backend.
```

Out[3]:

Out[5]:

```
Text(0, 0.5, 'RMSE')
```



In [5]:

```
historytotal
```

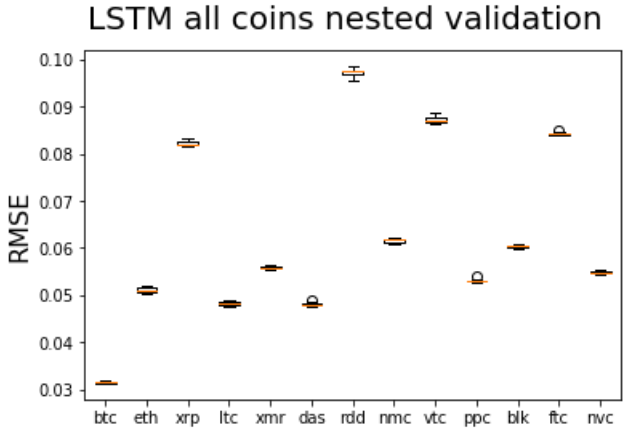
Out[5]:

	btctrain	btctest	ethtrain	ethtest	xrptrain	xrptest	ltctrain	ltctest	xmrtrain	xmrtest	...	vtctrain	vtctest	pp
0	0.030844	0.031942	0.086049	0.050163	0.069141	0.083811	0.051240	0.048496	0.058984	0.056570	...	0.110198	0.089743	0.0
1	0.030586	0.031980	0.085659	0.050107	0.068560	0.082943	0.050938	0.048867	0.058567	0.055671	...	0.109110	0.088498	0.0
2	0.030448	0.031785	0.086016	0.051644	0.068621	0.082221	0.050946	0.049349	0.059290	0.055715	...	0.108874	0.087924	0.0
3	0.030375	0.031689	0.086273	0.050812	0.068337	0.082218	0.050548	0.048269	0.058292	0.055476	...	0.108979	0.089148	0.0
4	0.030329	0.031593	0.085615	0.049922	0.068985	0.083046	0.050424	0.047958	0.058647	0.056105	...	0.108580	0.087953	0.0
5	0.030347	0.031988	0.085801	0.050494	0.068182	0.082593	0.050316	0.047774	0.058283	0.055400	...	0.109721	0.091244	0.0
6	0.030202	0.031563	0.085611	0.050569	0.068196	0.082212	0.050559	0.048629	0.058456	0.055601	...	0.108878	0.088798	0.0
7	0.030175	0.031522	0.085676	0.051138	0.068959	0.083348	0.050410	0.047742	0.058341	0.055504	...	0.107706	0.087966	0.0
8	0.030158	0.031621	0.085628	0.050052	0.068618	0.081409	0.050056	0.048067	0.058277	0.055148	...	0.107902	0.088796	0.0
9	0.030088	0.031468	0.085626	0.050899	0.068544	0.082433	0.049915	0.047948	0.058827	0.055616	...	0.107476	0.087474	0.0

10 rows × 26 columns

In [30]:

```

```

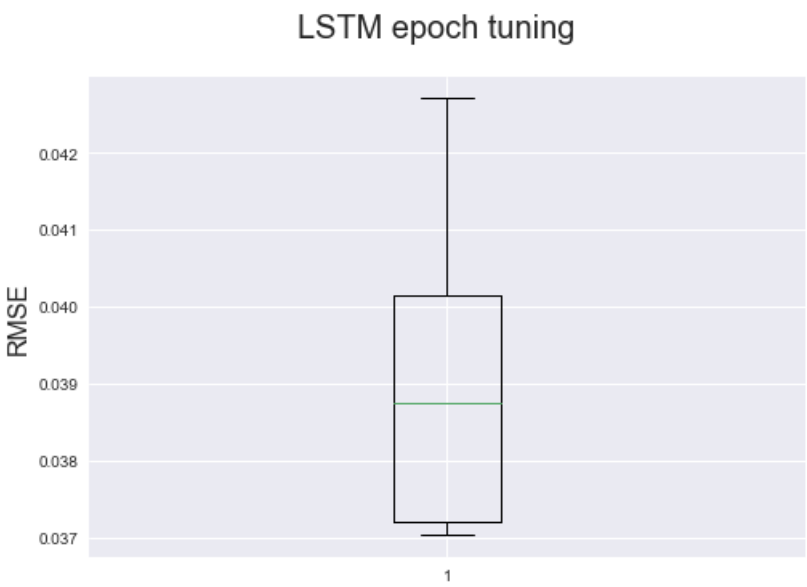


In [169]:

```
plt.boxplot(history[[x for x in history.columns.values if x[-4:]=='test']].T)
plt.suptitle('LSTM epoch tuning', fontsize=20)
plt.ylabel('RMSE', fontsize=16)
# plt.legend(fontsize=16)
# plt.savefig('output//Tuning_epoch_Bitcoin_return_LSTM.png')
```

Out[169]:

```
Text(0, 0.5, 'RMSE')
```



In [212]:

```
totalsse = np.power(testY,2).sum()/len(testY[0])
errorsse = np.power(testPredict[:,0],2).sum()/len(testPredict[:,0])
Rsquared = errorsse/totalsse
```

In [213]:

```
```

Out[213]:

```
0.022324336634538016
```

In [ ]:

```
```

In [210]:

```
math.sqrt(np.power(testPredict[:,0],2).sum()/len(testPredict[:,0]))
```

Out[210]:

```
0.00430064569914518
```

In [31]:

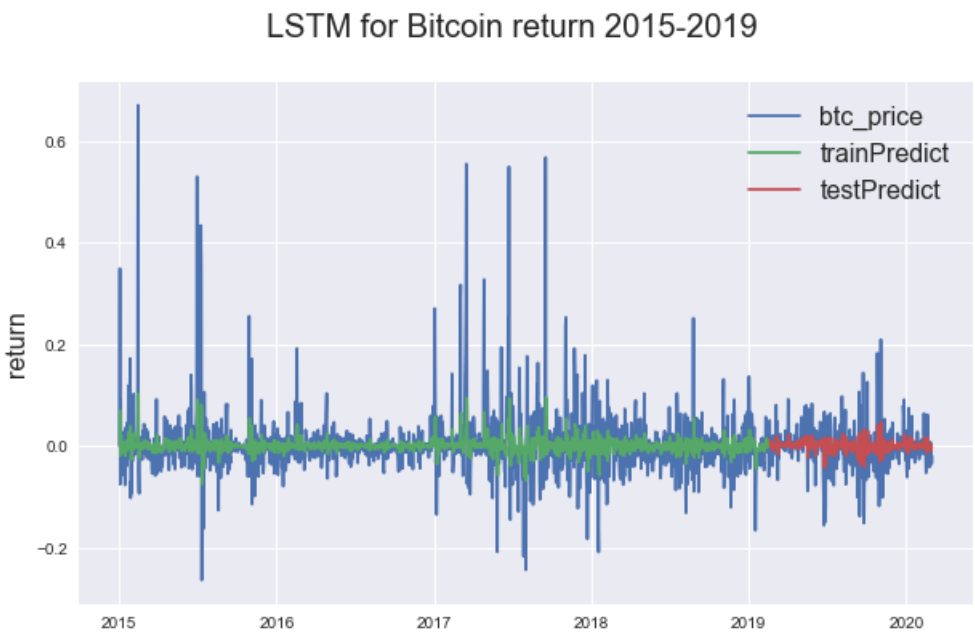
```
# shift train predictions for plotting
trainPredictPlot = numpy.empty_like(dataset)
trainPredictPlot[:, :] = numpy.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# shift test predictions for plotting
testPredictPlot = numpy.empty_like(dataset)
testPredictPlot[:, :] = numpy.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
```

```
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
# plot baseline and predictions
plt.style.use('seaborn')
plt.figure(figsize=(10,6))
plt.plot(returninfo.index.values, scaler.inverse_transform(dataset),label='btc_price')
plt.plot(returninfo.index.values,trainPredictPlot, label='trainPredict')
plt.plot(returninfo.index.values,testPredictPlot, label='testPredict')

plt.suptitle('LSTM for Bitcoin return 2015-2019', fontsize=20)
plt.ylabel('return', fontsize=16)
plt.legend(fontsize=16)
# plt.savefig('output//Bitcoin_return_LSTM.png')
```

```
C:\Users\WENTWORTHLIU\AppData\Roaming\Python\Python37\site-packages\pandas\plotting\_converter.py:129:
FutureWarning: Using an implicitly registered datetime converter for a matplotlib plotting method. The
converter was registered by pandas on import. Future versions of pandas will require you to explicitly
register matplotlib converters.

To register the converters:
 >>> from pandas.plotting import register_matplotlib_converters
 >>> register_matplotlib_converters()
  warnings.warn(msg, FutureWarning)
```

Out[31]:

```
<matplotlib.legend.Legend at 0x1bf8760ebc8>
```



In [151]:

```
# LSTM for cryptal currency problem with regression framing
import numpy
import matplotlib.pyplot as plt
from pandas import read_csv
import math
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
 dataX, dataY = [], []
 for i in range(len(dataset)-look_back-1):
  a = dataset[i:(i+look_back), 0]
  dataX.append(a)
  dataY.append(dataset[i + look_back, 0])
 return numpy.array(dataX), numpy.array(dataY)
batch_size=4
nb_neurons=1
```

```
# fix random seed for reproducibility
# numpy.random.seed(7)
# load the dataset
dataframe = returninfo[['btc_price']]
dataset = dataframe.values
dataset = dataset.astype('float32')
# normalize the dataset
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)
# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
# reshape into X=t and Y=t+1
look_back = 1
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
# reshape input to be [samples, time steps, features]
trainX = numpy.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
testX = numpy.reshape(testX, (testX.shape[0], 1, testX.shape[1]))
# create and fit the LSTM network
model = Sequential()
model.add(LSTM(nb_neurons, input_shape=(1, look_back)))
# model.add(LSTM(1, batch_input_shape=(batch_size, trainX.shape[1], trainX.shape[2]), stateful=True))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')

# fit model

nb_repeat = 10
nb_epoch = 10

history = DataFrame()
for epoch in [1,2,3,4,5,6]:
    train_rmse, test_rmse = list(), list()
    for r in range(nb_repeat):
        trainX, trainY = create_dataset(train, look_back)
        testX, testY = create_dataset(test, look_back)
        # reshape input to be [samples, time steps, features]
        trainX = numpy.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
        testX = numpy.reshape(testX, (testX.shape[0], 1, testX.shape[1]))
        model.fit(trainX, trainY, epochs=epoch, batch_size=batch_size, verbose=0)
    #         model.reset_states()

        # make predictions
        trainPredict = model.predict(trainX)
    #         model.reset_states()
        testPredict = model.predict(testX)
    #         model.reset_states()
        # invert predictions
        trainPredict = scaler.inverse_transform(trainPredict)
        trainY = scaler.inverse_transform([trainY])
        testPredict = scaler.inverse_transform(testPredict)
        testY = scaler.inverse_transform([testY])
        # calculate root mean squared error
        trainScore = math.sqrt(mean_squared_error(trainY[0], trainPredict[:,0]))
        train_rmse.append(trainScore)
        print('Train Score: %.4f RMSE' % (trainScore))
        testScore = math.sqrt(mean_squared_error(testY[0], testPredict[:,0]))
        test_rmse.append(testScore)
        print('Test Score: %.4f RMSE' % (testScore))

    history[str(epoch)+'train'], history[str(epoch)+'test'] = train_rmse, test_rmse
#     plt.plot(history['train'], color='blue')
#     plt.plot(history['test'], color='orange')
history.boxplot(column=[x for x in history.columns.values if x[-4:]=='test'])
```

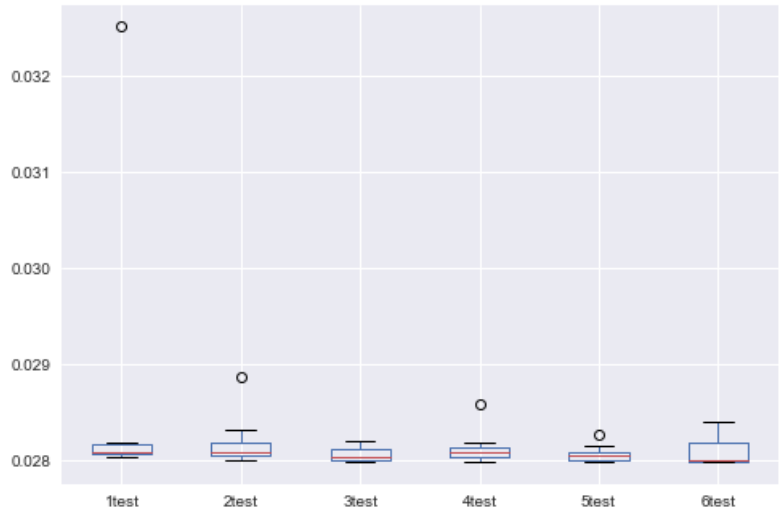
```
Train Score: 0.0371 RMSE
Test Score: 0.0325 RMSE
Train Score: 0.0325 RMSE
Test Score: 0.0281 RMSE
Train Score: 0.0325 RMSE
Test Score: 0.0281 RMSE
Train Score: 0.0325 RMSE
Test Score: 0.0281 RMSE
```

```
Train Score: 0.0325 RMSE
Test Score: 0.0282 RMSE
Train Score: 0.0325 RMSE
Test Score: 0.0282 RMSE
Train Score: 0.0325 RMSE
Test Score: 0.0280 RMSE
Train Score: 0.0326 RMSE
Test Score: 0.0281 RMSE
Train Score: 0.0325 RMSE
Test Score: 0.0281 RMSE
Train Score: 0.0325 RMSE
Test Score: 0.0281 RMSE
Train Score: 0.0326 RMSE
Test Score: 0.0280 RMSE
Train Score: 0.0325 RMSE
Test Score: 0.0282 RMSE
Train Score: 0.0329 RMSE
Test Score: 0.0289 RMSE
Train Score: 0.0326 RMSE
Test Score: 0.0281 RMSE
Train Score: 0.0324 RMSE
Test Score: 0.0280 RMSE
Train Score: 0.0324 RMSE
Test Score: 0.0281 RMSE
Train Score: 0.0324 RMSE
Test Score: 0.0280 RMSE
Train Score: 0.0325 RMSE
Test Score: 0.0283 RMSE
Train Score: 0.0324 RMSE
Test Score: 0.0281 RMSE
Train Score: 0.0324 RMSE
Test Score: 0.0281 RMSE
Train Score: 0.0325 RMSE
Test Score: 0.0282 RMSE
Train Score: 0.0325 RMSE
Test Score: 0.0280 RMSE
Train Score: 0.0324 RMSE
Test Score: 0.0281 RMSE
Train Score: 0.0325 RMSE
Test Score: 0.0280 RMSE
Train Score: 0.0324 RMSE
Test Score: 0.0280 RMSE
Train Score: 0.0324 RMSE
Test Score: 0.0281 RMSE
Train Score: 0.0324 RMSE
Test Score: 0.0280 RMSE
Train Score: 0.0324 RMSE
Test Score: 0.0280 RMSE
Train Score: 0.0324 RMSE
Test Score: 0.0281 RMSE
Train Score: 0.0324 RMSE
Test Score: 0.0281 RMSE
Train Score: 0.0324 RMSE
Test Score: 0.0280 RMSE
Train Score: 0.0324 RMSE
Test Score: 0.0280 RMSE
Train Score: 0.0324 RMSE
Test Score: 0.0281 RMSE
Train Score: 0.0324 RMSE
Test Score: 0.0280 RMSE
Train Score: 0.0324 RMSE
Test Score: 0.0281 RMSE
Train Score: 0.0327 RMSE
Test Score: 0.0286 RMSE
Train Score: 0.0324 RMSE
Test Score: 0.0280 RMSE
Train Score: 0.0325 RMSE
Test Score: 0.0282 RMSE
Train Score: 0.0326 RMSE
Test Score: 0.0281 RMSE
Train Score: 0.0327 RMSE
Test Score: 0.0282 RMSE
Train Score: 0.0326 RMSE
Test Score: 0.0281 RMSE
Train Score: 0.0326 RMSE
Test Score: 0.0280 RMSE
Train Score: 0.0324 RMSE
```

```
Test Score: 0.0280 RMSE
Train Score: 0.0324 RMSE
Test Score: 0.0280 RMSE
Train Score: 0.0328 RMSE
Test Score: 0.0282 RMSE
Train Score: 0.0324 RMSE
Test Score: 0.0281 RMSE
Train Score: 0.0324 RMSE
Test Score: 0.0281 RMSE
Train Score: 0.0325 RMSE
Test Score: 0.0283 RMSE
Train Score: 0.0324 RMSE
Test Score: 0.0280 RMSE
Train Score: 0.0324 RMSE
Test Score: 0.0280 RMSE
Train Score: 0.0325 RMSE
Test Score: 0.0280 RMSE
Train Score: 0.0324 RMSE
Test Score: 0.0280 RMSE
Train Score: 0.0325 RMSE
Test Score: 0.0282 RMSE
Train Score: 0.0324 RMSE
Test Score: 0.0282 RMSE
Train Score: 0.0326 RMSE
Test Score: 0.0284 RMSE
Train Score: 0.0324 RMSE
Test Score: 0.0280 RMSE
Train Score: 0.0324 RMSE
Test Score: 0.0280 RMSE
Train Score: 0.0326 RMSE
Test Score: 0.0284 RMSE
Train Score: 0.0326 RMSE
Test Score: 0.0280 RMSE
Train Score: 0.0324 RMSE
Test Score: 0.0280 RMSE
```

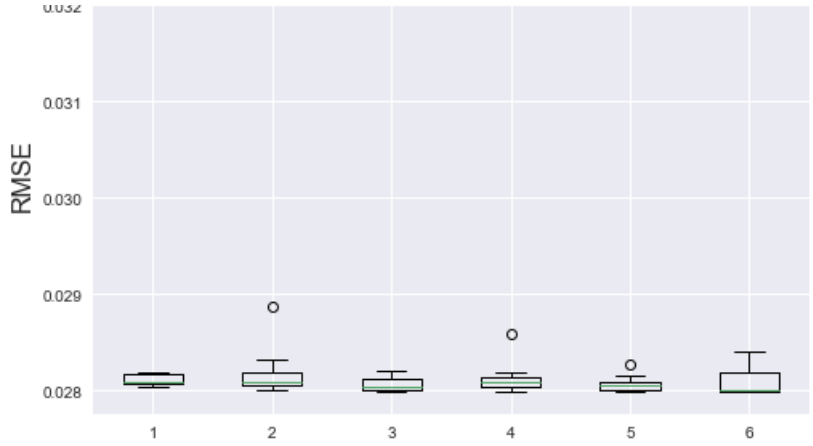
Out[151]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x2076882ae48>
```



In [159]:

```
plt.boxplot(history[[x for x in history.columns.values if x[-4:]=='test']].T)
plt.suptitle('LSTM epoch tuning', fontsize=20)
plt.ylabel('RMSE', fontsize=16)
# plt.legend(fontsize=16)
plt.savefig('output//Tuning_epoch_Bitcoin_return_LSTM.png')
```



In [155]:

```
history[[x for x in history.columns.values if x[-4:]=='test']]
```

Out[155]:

	1test	2test	3test	4test	5test	6test
0	0.032511	0.028047	0.028206	0.028049	0.028075	0.027986
1	0.028095	0.028200	0.027990	0.027996	0.028036	0.027987
2	0.028080	0.028874	0.028124	0.028104	0.028009	0.028202
3	0.028113	0.028083	0.027990	0.027996	0.028010	0.028167
4	0.028193	0.028008	0.028012	0.028098	0.028161	0.028401
5	0.028188	0.028091	0.028054	0.028584	0.028081	0.027985
6	0.028046	0.027997	0.027998	0.028035	0.028095	0.027984
7	0.028058	0.028317	0.028000	0.028190	0.028273	0.028376
8	0.028080	0.028144	0.028134	0.028072	0.028022	0.028027
9	0.028061	0.028083	0.028097	0.028154	0.027986	0.027996

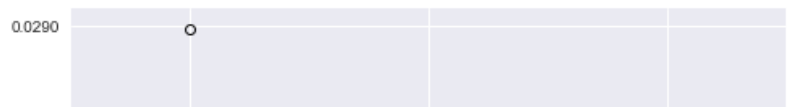
In [160]:

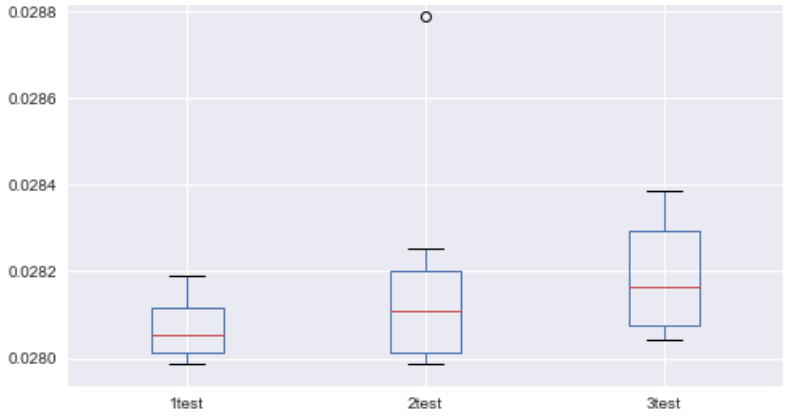
```
# LSTM for cryptal currency problem with regression framing
import numpy
import matplotlib.pyplot as plt
from pandas import read_csv
import math
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
 dataX, dataY = [], []
 for i in range(len(dataset)-look_back-1):
  a = dataset[i:(i+look_back), 0]
  dataX.append(a)
  dataY.append(dataset[i + look_back, 0])
 return numpy.array(dataX), numpy.array(dataY)
batch_size=4
nb_neurons=1
# fix random seed for reproducibility
# numpy.random.seed(7)
# load the dataset
dataframe = returninfo[['btc_price']]
dataset = dataframe.values
dataset = dataset.astype('float32')
# normalize the dataset
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)
```

```
# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
# reshape into X=t and Y=t+1
look_back = 1
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
# reshape input to be [samples, time steps, features]
trainX = numpy.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
testX = numpy.reshape(testX, (testX.shape[0], 1, testX.shape[1]))

nb_repeat = 10
epoch = 3
history = DataFrame()
for batch_size in [1,2,3]:
    # create and fit the LSTM network
    model = Sequential()
    model.add(LSTM(nb_neurons, input_shape=(1, look_back)))
    # model.add(LSTM(1, batch_input_shape=(batch_size, trainX.shape[1], trainX.shape[2]), stateful=True
))
    model.add(Dense(1))
    model.compile(loss='mean_squared_error', optimizer='adam')

    # fit model



    train_rmse, test_rmse = list(), list()
    for r in range(nb_repeat):
        trainX, trainY = create_dataset(train, look_back)
        testX, testY = create_dataset(test, look_back)
        # reshape input to be [samples, time steps, features]
        trainX = numpy.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
        testX = numpy.reshape(testX, (testX.shape[0], 1, testX.shape[1]))
        model.fit(trainX, trainY, epochs=epoch, batch_size=batch_size, verbose=0)
    #         model.reset_states()

        # make predictions
        trainPredict = model.predict(trainX)
    #         model.reset_states()
        testPredict = model.predict(testX)
    #         model.reset_states()
        # invert predictions
        trainPredict = scaler.inverse_transform(trainPredict)
        trainY = scaler.inverse_transform([trainY])
        testPredict = scaler.inverse_transform(testPredict)
        testY = scaler.inverse_transform([testY])
        # calculate root mean squared error
        trainScore = math.sqrt(mean_squared_error(trainY[0], trainPredict[:,0]))
        train_rmse.append(trainScore)
#         print('Train Score: %.4f RMSE' % (trainScore))
        testScore = math.sqrt(mean_squared_error(testY[0], testPredict[:,0]))
        test_rmse.append(testScore)
#         print('Test Score: %.4f RMSE' % (testScore))

    history[str(batch_size)+'train'], history[str(batch_size)+'test'] = train_rmse, test_rmse
#     plt.plot(history['train'], color='blue')
#     plt.plot(history['test'], color='orange')
history.boxplot(column=['1test','2test','3test'])

plt.boxplot(history[[x for x in history.columns.values if x[-4:]=='test']].T)
plt.suptitle('LSTM batch tuning', fontsize=20)
plt.ylabel('RMSE', fontsize=16)
# plt.legend(fontsize=16)
plt.savefig('output//Tuning_batch_Bitcoin_return_LSTM.png')
```

Out[160]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x20769b95e08>
```

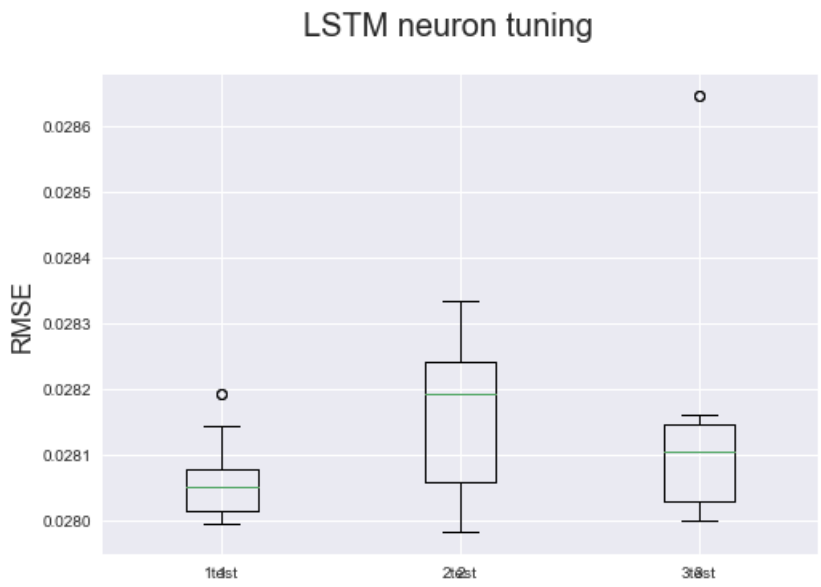




In [163]:

```
# LSTM for cryptal currency problem with regression framing
import numpy
import matplotlib.pyplot as plt
from pandas import read_csv
import math
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
 dataX, dataY = [], []
 for i in range(len(dataset)-look_back-1):
  a = dataset[i:(i+look_back), 0]
  dataX.append(a)
  dataY.append(dataset[i + look_back, 0])
 return numpy.array(dataX), numpy.array(dataY)
batch_size=4
nb_neurons=1
# fix random seed for reproducibility
# numpy.random.seed(7)
# load the dataset
dataframe = returninfo[['btc_price']]
dataset = dataframe.values
dataset = dataset.astype('float32')
# normalize the dataset
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)
# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
# reshape into X=t and Y=t+1
look_back = 1
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
# reshape input to be [samples, time steps, features]
trainX = numpy.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
testX = numpy.reshape(testX, (testX.shape[0], 1, testX.shape[1]))

nb_repeat = 10
epoch = 3
history = DataFrame()
for nb_neurons in [1,2,3]:
    # create and fit the LSTM network
    model = Sequential()
    model.add(LSTM(nb_neurons, input_shape=(1, look_back)))
    # model.add(LSTM(1, batch_input_shape=(batch_size, trainX.shape[1], trainX.shape[2]), stateful=True
))
    model.add(Dense(1))
    model.compile(loss='mean_squared_error', optimizer='adam')

    # fit model
```

```
    train_rmse, test_rmse = list(), list()
    for r in range(nb_repeat):
        trainX, trainY = create_dataset(train, look_back)
        testX, testY = create_dataset(test, look_back)
        # reshape input to be [samples, time steps, features]
        trainX = numpy.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
        testX = numpy.reshape(testX, (testX.shape[0], 1, testX.shape[1]))
        model.fit(trainX, trainY, epochs=epoch, batch_size=batch_size, verbose=0)
    #         model.reset_states()

        # make predictions
        trainPredict = model.predict(trainX)
    #         model.reset_states()
        testPredict = model.predict(testX)
    #         model.reset_states()
        # invert predictions
        trainPredict = scaler.inverse_transform(trainPredict)
        trainY = scaler.inverse_transform([trainY])
        testPredict = scaler.inverse_transform(testPredict)
        testY = scaler.inverse_transform([testY])
        # calculate root mean squared error
        trainScore = math.sqrt(mean_squared_error(trainY[0], trainPredict[:,0]))
        train_rmse.append(trainScore)
#         print('Train Score: %.4f RMSE' % (trainScore))
        testScore = math.sqrt(mean_squared_error(testY[0], testPredict[:,0]))
        test_rmse.append(testScore)
#         print('Test Score: %.4f RMSE' % (testScore))

    history[str(nb_neurons)+'train'], history[str(nb_neurons)+'test'] = train_rmse, test_rmse
#     plt.plot(history['train'], color='blue')
#     plt.plot(history['test'], color='orange')
history.boxplot(column=['1test','2test','3test'])

plt.boxplot(history[[x for x in history.columns.values if x[-4:]=='test']].T)
plt.suptitle('LSTM neuron tuning', fontsize=20)
plt.ylabel('RMSE', fontsize=16)
# plt.legend(fontsize=16)
plt.savefig('output//Tuning_neuron_Bitcoin_return_LSTM.png')
```



In [149]:

```

```

Out[149]:

```
['5test']
```

Project code

Joanna Zhang

3/12/2020

```
# LSTM project

library(keras)
library(tensorflow)
library(readr)
library(tidyverse)
library(dplyr)

coinnames <- as.list(as.character(coinlist$Coins))
# pricedf <- vector("list", length(coinlist$Coins))
# for (i in 48:49){
#   link <- str_c("https://coinmarketcap.com/currencies/",
#                 coinnames[i], "/historical-data/?start=20190201&end=20200130")
#   table <- read_html(link) %>%
#     html_nodes(css = "table") %>%
#     # the third table on the webpage is the one that we will use
#     nth(3) %>%
#     html_table(header = TRUE)
#   # Some of the currencies in the list have numbers in the numeric form already but
#   # others don't. Need to convert strings to numbers for future calculation.
#   if (!is.numeric(table$`Open*`)){
#     table <- mutate(table,
#                     Open = as.numeric(gsub(",", "", table$`Open*`)),
#                     Close = as.numeric(gsub(",", "", table$`Close**`)))
#   } else{
#     #If the numbers in the table are already in numeric form, then just rename the columns
#     table <- mutate(table, Open = `Open*`, Close = `Close**`)
#   }
#   pricedf[[i]]<-table %>%
#     #Calculate the price of each day by taking the average of open price and close price
#     mutate(avgprice = (as.numeric(Open)+as.numeric(Close))/2) %>%
#     # Create a column that marks the cryptocurrency is associated with this table
#     mutate(currency_name = coinnames[i]) %>%
#     #Drop useless columns
#     subset(select = c(Date, currency_name, Open, Close, avgprice))
# }
# crypto_prices <- bind_rows(pricedf)
# crypto_prices$currency_name <- unlist(crypto_prices$currency_name)
# crypto_prices$Date <- as.Date(crypto_prices$Date, format = "%b %d, %Y")

returns <- read_csv("Desktop/returninfo.csv")

lag_transform <- function(x, k= 1){

  lagged =  c(rep(NA, k), x[1:(length(x)-k)])
  DF = as.data.frame(cbind(lagged, x))
  colnames(DF) <- c( paste0('x-', k), 'x')
```

```
  DF[is.na(DF)] <- 0
  return(DF)
}

scale_data = function(train, test, feature_range = c(0, 1)) {
  x = train
  fr_min = feature_range[1]
  fr_max = feature_range[2]
  std_train = ((x - min(x) ) / (max(x) - min(x)  ))
  std_test  = ((test - min(x) ) / (max(x) - min(x)  ))

  scaled_train = std_train *(fr_max -fr_min) + fr_min
  scaled_test = std_test *(fr_max -fr_min) + fr_min
  return( list(scaled_train = as.vector(scaled_train),
               scaled_test = as.vector(scaled_test),
               scaler= c(min =min(x), max = max(x))) )
}

invert_scaling = function(scaled, scaler, feature_range = c(0, 1)){
  min = scaler[1]
  max = scaler[2]
  t = length(scaled)
  mins = feature_range[1]
  maxs = feature_range[2]
  inverted_dfs = numeric(t)

  for( i in 1:t){
    X = (scaled[i]- mins)/(maxs - mins)
    rawValues = X *(max - min) + min
    inverted_dfs[i] <- rawValues
  }
  return(inverted_dfs)
}

reshape_X_3d <- function(X) {
  dim(X) <- c(dim(X)[1], dim(X)[2], 1)
  X
}


ethseries <- returns$nvc_price
#diffed = diff(ethseries, differences = 1)
supervised = lag_transform(ethseries, 1)
N<-nrow(supervised)
n = 1320
train = supervised[1:n, ]
test  = supervised[(n+4):N,  ]

Scaled = scale_data(train, test, c(-1, 1))

y_train = Scaled$scaled_train[, 2]
x_train = Scaled$scaled_train[, 1]
```

```
y_test = Scaled$scaled_test[, 2]
x_test = matrix(Scaled$scaled_test[, 1])

dim(x_train) <- c(length(x_train), 1, 1)

# specify required arguments
X_shape2 = dim(x_train)[2]
X_shape3 = dim(x_train)[3]
units = 1

#Tune batch size
rmse_batch <- data.frame(matrix(nrow=10, ncol=3))
for (number in 1:10){
  rmse_vec<-vector()
for (b in 1:3){
  model <- keras_model_sequential()
  model%>%
    layer_lstm(units, batch_input_shape = c(batch_size = b, X_shape2, X_shape3),
               stateful= F)%>%
    layer_dense(units = 1)
  model %>% compile(
    loss = 'mean_squared_error',
    optimizer = optimizer_adam( lr= 0.01, decay = 1e-6 ),
    metrics = c('accuracy')
  )
  Epochs = 3
  for(i in 1:Epochs ){
    model %>% fit(x_train, y_train, epochs=1, batch_size=b, verbose=1, shuffle=FALSE)
    model %>% reset_states()
  }
  scaler = Scaled$scaler
  predictions = numeric(length(x_test))
    X_test <- reshape_X_3d(x_test)
    yhat = model %>% predict(X_test, batch_size = b)
    # invert scaling
    yhat = invert_scaling(yhat, scaler,  c(-1, 1))
  # for(a in 1:length(yhat)){
  #   # invert differencing
  #   predictions[a]  = yhat[a] + ethseries[(n+a)]
  # }
  actual <- returns[(n+4):N,]$btc_price
  rmse <- sqrt(mean((actual - yhat)^2))
  rmse_vec <- append(rmse_vec, rmse)
}
  rmse_batch[number,] = rmse_vec
}


data.frame(RMSE=c(rmse_batch$X1, rmse_batch$X2, rmse_batch$X3)) %>%
  mutate(batch = as.factor(rep(1:3, times=10, each=1))) %>%
  ggplot()+
  geom_boxplot(aes(x = batch, y = RMSE))
```

```
##Expanded Window
#a = as.integer(2/3 * length(returns$btc_price))
#b = length(returns$btc_price) - 1

a = 1870
train = supervised
test  = supervised
Scaled = scale_data(train, test, c(-1, 1))
units = 1

rmse_window <- vector()
for(i in 1:10){
  x_train_walk = Scaled$scaled_train[, 2][1:(a+i)]
  y_train_walk = Scaled$scaled_train[, 1][1:(a+i)]
  y_test_walk = Scaled$scaled_test[, 2][a+i+1]
  x_test_walk = matrix(Scaled$scaled_test[, 1][a+i+1])

  dim(x_train_walk) <- c(length(x_train_walk), 1, 1)

  X_shape2 = dim(x_train_walk)[2]
  X_shape3 = dim(x_train_walk)[3]

  for (n in 1:2){
    error <- vector()
  model <- keras_model_sequential()
  model%>%
    layer_lstm(units, batch_input_shape = c(batch_size = 1, X_shape2, X_shape3),
               stateful= F)%>%
    layer_dense(units = 1)
  model %>% compile(
    loss = 'mean_squared_error',
    optimizer = optimizer_adam( lr= 0.01, decay = 1e-6 ),
    metrics = c('accuracy')
  )
  Epochs = 3
  for(e in 1:Epochs ){
    model %>% fit(x_train_walk, y_train_walk, epochs=1, batch_size=1, verbose=1, shuffle=FALSE)
    model %>% reset_states()
  }
  scaler = Scaled$scaler
  predictions = numeric(length(x_test_walk))
  X_test <- reshape_X_3d(x_test_walk)
  yhat = model %>% predict(X_test, batch_size = 1)
  # invert scaling
  yhat = invert_scaling(yhat, scaler,  c(-1, 1))
  actual <- returns[i+a+1,]$btc_price
  error <- append(error, (actual - yhat)^2)
  }
  #rmse <- sqrt(mean(error))
  rmse_window <- append(rmse_window, sqrt(mean(error)))
}
```

```r
rmse_exwin <-rmse_exwin %>%
  mutate(nvc = rmse_window)


rmse_exwin %>%
  pivot_longer(colnames(rmse_exwin), names_to = "coin", values_to = "rmse_exp_wind") %>%
  ggplot()+
  geom_boxplot(aes(as.factor(coin), rmse_exp_wind))+
  labs(title = "RMSE All Coins Expanded Window Validation",
       x = "Coins", y = "RMSE",
       subtitle = "Using the last 10 days' prices as test set")
```

Final Project (parts) - Runhua Li

Runhua Li

3/12/2020

Introduction

In this document I show how some parts of our final project. GARCH model selection, generation of model comparison results are included here.

Selecting GARCH model

Set Up

```
library(rugarch) #univariate GARCH
```

```
## Loading required package: parallel
```

```
##
## Attaching package: 'rugarch'
```

```
## The following object is masked from 'package:stats':
##
##     sigma
```

```
  #library(rmgarch) #multivariate GARCH
library(tidyverse)
```

```
## -- Attaching packages --------------------------------- tidyverse 1.2.1 --
```

```
## v ggplot2 3.2.1     v purrr   0.3.3
## v tibble  2.1.3     v dplyr   0.8.4
## v tidyr   1.0.0     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0
```

```
## -- Conflicts ------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## x purrr::reduce() masks rugarch::reduce()
```

```
library(tseries) #arma fcn.
```

```
## Registered S3 method overwritten by 'quantmod':
##   method            from
##   as.zoo.data.frame zoo
```

```
setwd("~/R")
rm(list = ls())

coin <- read.csv("priceinfo.csv")
```

GARCH Variance & Mean Model Specification

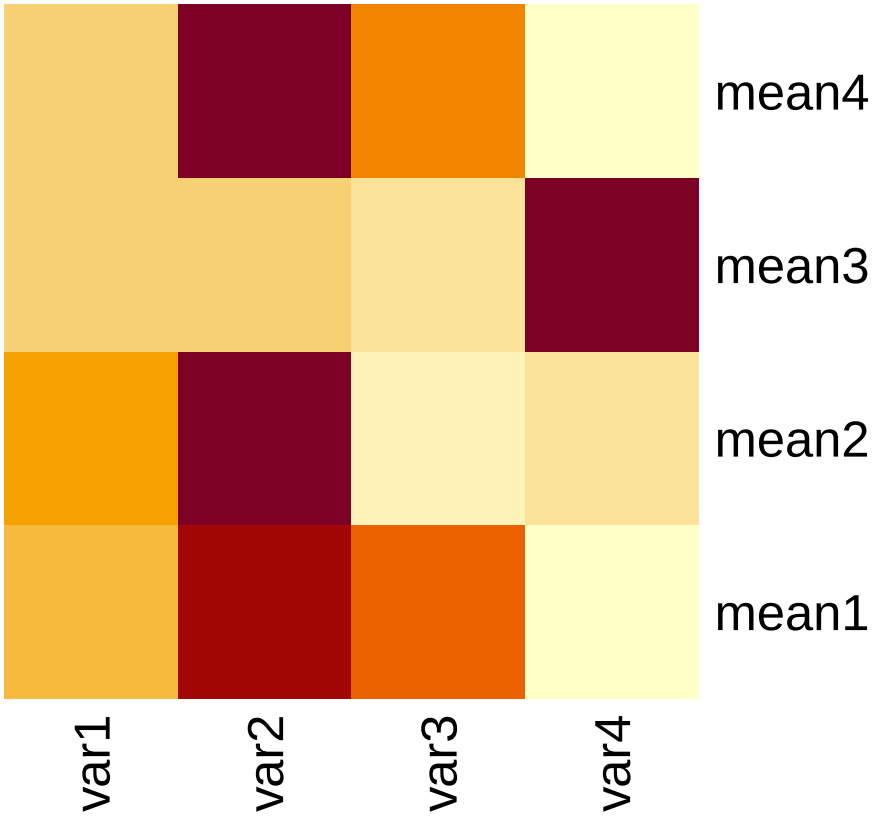
```r
coin <- coin %>%
  mutate(btc_lag = c(NA, btc_price[1:(length(btc_price) - 1)]),
         btc_return = btc_price / btc_lag)

# sqrt(var(coin$btc_return, na.rm = TRUE))
  # SD of outcomes

arma <- arma(coin$btc_return[-1], order = c(5, 5))
```

```
## Warning in arma(coin$btc_return[-1], order = c(5, 5)): singular Hessian
```

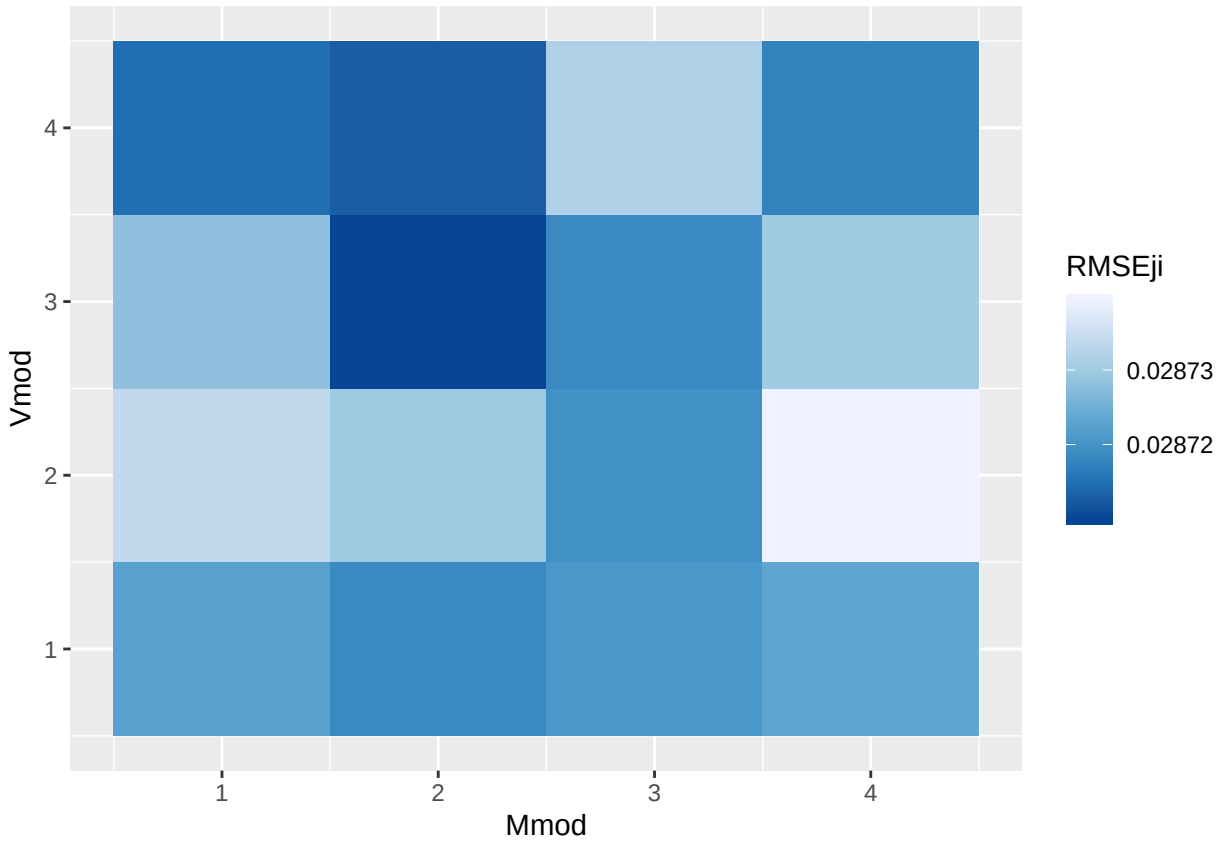
```r
arma
```

```
##
## Call:
## arma(x = coin$btc_return[-1], order = c(5, 5))
##
## Coefficient(s):
##       ar1        ar2        ar3        ar4        ar5        ma1
##    0.9707    -0.6976    -0.4459     0.4792     0.2417    -0.7636
##       ma2        ma3        ma4        ma5  intercept
##    0.2752     0.6417    -0.3484    -0.2760     0.4688
```

```r
# telling form the result, only try ARMA from (1, 1) to (4, 4)
# note: also try variance-model from (1, 1) to (4, 4)

x <- coin$btc_return[-1]
# 2/3 * length(x) generates 1257.3, 1886 - 1257 = 629

RMSE_ji = c(0, 0, 0, 0, 0)
for (j in 1:4){
  RMSE_i = 0
  for (i in 1:4){
    ug_spec <- ugarchspec(mean.model = list(armaOrder = c(i, i)),
                          variance.model = list(garchOrder = c(j, j)))

    ugfit = ugarchfit(spec = ug_spec, data = x[1:1257])
    ugforecast = ugarchforecast(ugfit, n.ahead = 629)
    RMSE_i = c(RMSE_i,
                  sqrt(mean((x[1258:1886] -
                               ugforecast@forecast$seriesFor)^2)))
  }
  RMSE_ji = rbind(RMSE_ji, RMSE_i)
}

RMSE_ji = RMSE_ji[-1, -1]
rownames(RMSE_ji) <- c("mean1", "mean2", "mean3", "mean4")
colnames(RMSE_ji) <- c("var1", "var2", "var3", "var4")

heatmap(RMSE_ji, Rowv = NA, Colv = NA) # this fcn. seems not the right one
```




```
## Drawing heatmap by ggplot2
Vmod <- c(c(1:4), c(1:4), c(1:4), c(1:4))
Mmod <- c(1, 1, 1, 1,
          2, 2, 2, 2,
          3, 3, 3, 3,
          4, 4, 4, 4) # I later learned this can be done better by gather()
RMSEji <- c(RMSE_ji[1, ], RMSE_ji[2, ], RMSE_ji[3, ], RMSE_ji[4, ])

RMSEmap <- data.frame(Vmod, Mmod, RMSEji)

ggplot(RMSEmap, aes(Mmod, Vmod, fill = RMSEji)) +
  geom_tile() +
  scale_fill_distiller()
```




```
# So: let's do GARCH(3, 3) with ARMA(2, 2)!
```

Model Comparison

Set Up

```
library(tidyverse)

setwd("~/R")
rm(list = ls())

coin <- read.csv("returninfo.csv")

#Model specification
ug_spec <- ugarchspec(mean.model = list(armaOrder = c(2, 2)),
                      variance.model = list(garchOrder = c(3, 3)))
```

Loading Functions of Cross Validation

```
## 1/3 Holdout (fcn.)
holdout <- function(return){
  ugfit = ugarchfit(spec = ug_spec, data = return[1:1257], solver = "hybrid")
  ugforecast = ugarchforecast(ugfit, n.ahead = 629)
  RMSE = sqrt(mean((return[1258:1886] -
                      ugforecast@forecast$seriesFor)^2))
  RMSE
```

```
}

## Expanding Window CV (fcn.)
expand <- function(return, j){ # j = 1 for full sample, j = 1521 for last year
  a = length(return) - 11
  b = length(return) - 1
  e = 0

  for(i in a:b){
    uGARCHfit <- ugarchfit(spec = ug_spec, data = return[j:i], solver = "hybrid")
    uGARCHforecast <-ugarchforecast(uGARCHfit, n.ahead = 1)
    e = c(e, return[i+1] - uGARCHforecast@forecast$seriesFor)
  }
  e = e[-1]
  RMSE = sqrt(mean(e^2))
  RMSE
}

## Nested CV (fcn.)
nested <- function(return){
  l = length(return)
  n = (l - 1) / 5
  e = 0

  for(i in 1:4){
    uGARCHfit <- ugarchfit(spec = ug_spec, data = return[1:(i * n)], solver = "hybrid")
    uGARCHforecast <- ugarchforecast(uGARCHfit, n.ahead = n)
    e = c(e, return[c((i * n + 1):(i * n + n))] - uGARCHforecast@forecast$seriesFor)
  }
  e = e[-1]

  RMSE = sqrt(mean(e^2))
  RMSE
}
```

Running CV Functions

```
# ---------- Let's Run!

hRMSE = 0
for (i in 2:14){
  return <- coin[-1, i]
  hRMSE = c(hRMSE, holdout(return))
}
hRMSE = hRMSE[-1]
hRMSE
```

```
##  [1] 0.02873640 0.03810667 0.08698240 0.03981588 0.03651563 0.04542437
##  [7] 0.04558480 0.05986396 0.06576448 0.04424060 0.04562711 0.05189716
## [13] 0.04317495
```

```
nRMSE = 0
for (i in 2:14){
  return <- coin[-1, i]
```

```
  nRMSE = c(nRMSE, nested(return))
}
```

```
## Warning in arima(data, order = c(modelinc[2], 0, modelinc[3]), include.mean
## = modelinc[1], : possible convergence problem: optim gave code = 1
```

```
nRMSE = nRMSE[-1]
nRMSE
```

```
##  [1] 0.03235494 0.05196581 0.08350835 0.04899687 0.05633834 0.04917099
##  [7] 0.09700495 0.06153333 0.09328647 0.05348302 0.06089896 0.08700659
## [13] 0.05614778
```

```
fRMSE = 0
for (i in 2:14){
  return <- coin[-1, i]
  fRMSE = c(fRMSE, expand(return, 1))
}
fRMSE = fRMSE[-1]
fRMSE
```

```
##  [1] 0.02600870 0.04240162 0.03920594 0.05122048 0.04232776 0.04738156
##  [7] 0.05130925 0.02549011 0.05671115 0.02675011 0.02175306 0.03088691
## [13] 0.02188859
```

```
yRMSE = 0
for (i in 2:14){
  return <- coin[-1, i]
  yRMSE = c(yRMSE, expand(return, 1521))
}
```

```
## Warning in arima(data, order = c(modelinc[2], 0, modelinc[3]), include.mean
## = modelinc[1], : possible convergence problem: optim gave code = 1

## Warning in arima(data, order = c(modelinc[2], 0, modelinc[3]), include.mean
## = modelinc[1], : possible convergence problem: optim gave code = 1

## Warning in arima(data, order = c(modelinc[2], 0, modelinc[3]), include.mean
## = modelinc[1], : possible convergence problem: optim gave code = 1
```

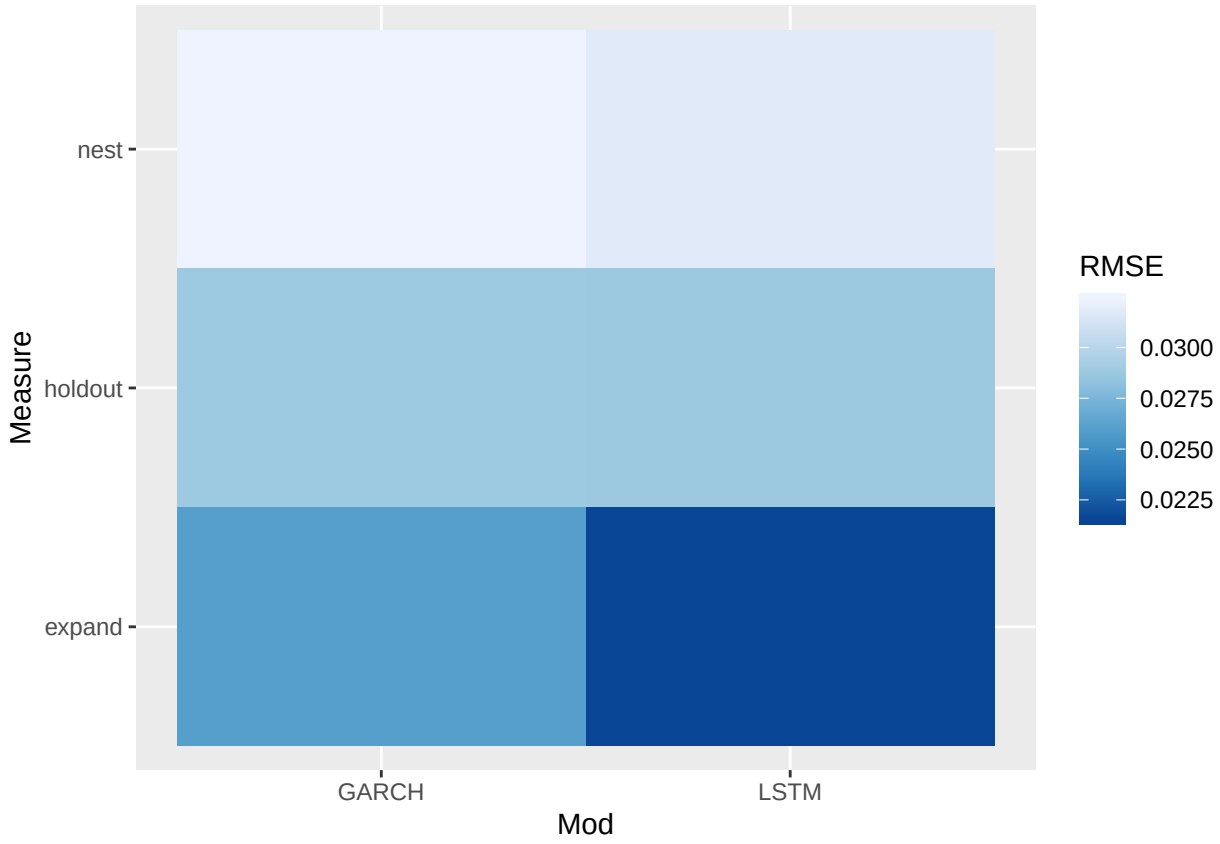
```
yRMSE = yRMSE[-1]
yRMSE
```

```
##  [1] 0.02563527 0.04226585 0.03925731 0.05380066 0.04223646 0.04801784
##  [7] 0.05179442 0.02931637 0.05236172 0.02664826 0.02317243 0.03342657
## [13] 0.02888460
```

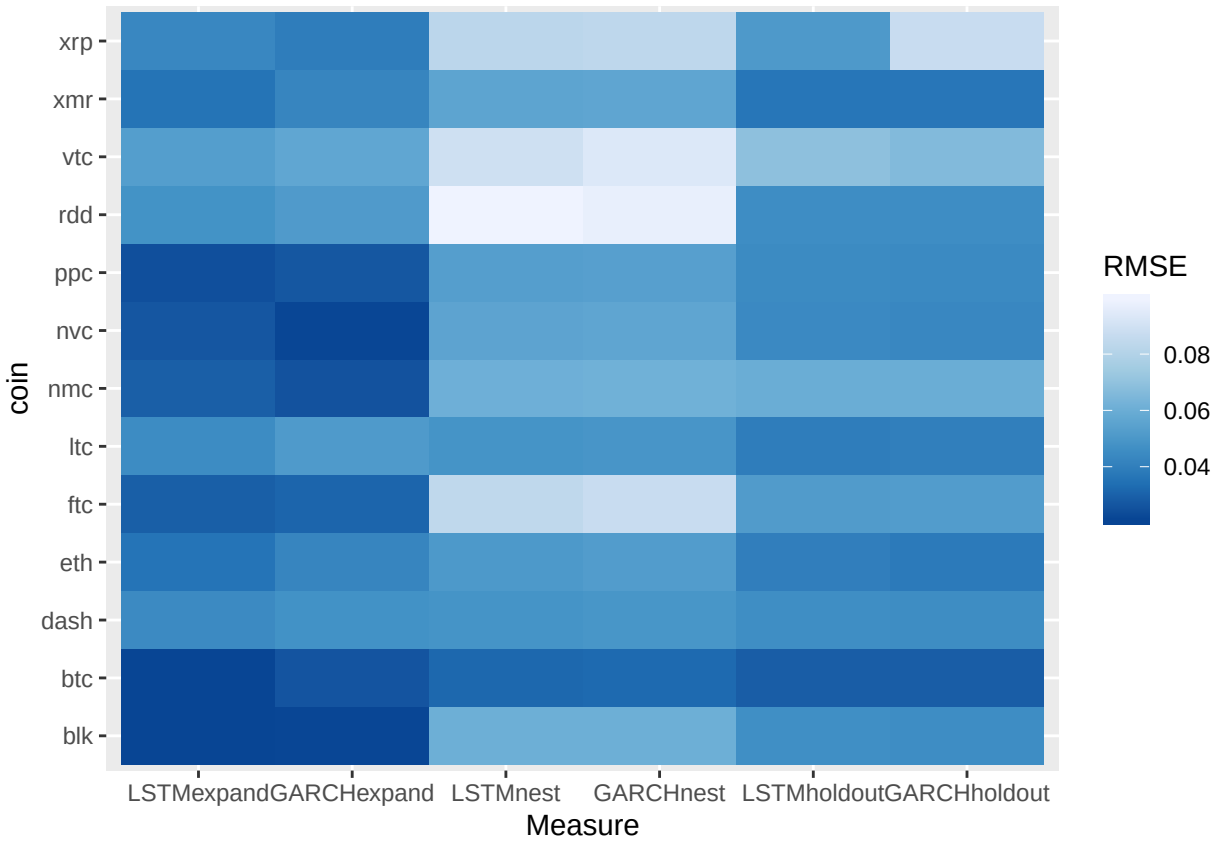
Obtaining Comparison Results

```
RMSEmap <- read.csv("combined_RMSE.csv") #from Xinyu Liu

names(RMSEmap) <- c("coin", "LSTMexpand", "LSTMnest", "LSTMholdout")
RMSEmap$coin <- c("btc", "eth", "xrp", "ltc", "xmr", "dash", "rdd",
                  "nmc", "vtc", "ppc", "blk", "ftc", "nvc")

GARCHexpand = fRMSE
GARCHnest = nRMSE
```

```
GARCHholdout = hRMSE

RMSEmap <- data.frame(RMSEmap, GARCHexpand, GARCHnest, GARCHholdout)

Mod = c("LSTM", "LSTM", "LSTM", "GARCH", "GARCH", "GARCH")
Measure = c("expand", "nest", "holdout", "expand", "nest", "holdout")

BTCmap <- data.frame(Mod, Measure, t(RMSEmap[1, 2:7]))
names(BTCmap)[3] <- "RMSE"

ggplot(BTCmap, aes(Mod, Measure, fill = RMSE)) +
  geom_tile() +
  scale_fill_distiller() #drawing comparison heatmap on Bitcoin data
```




```
RMSEwide <- RMSEmap[, c(1, 2, 5, 3, 6, 4, 7)]

RMSElong <- gather(RMSEwide, Measure, RMSE, LSTMexpand:GARCHholdout, factor_key=TRUE)

ggplot(RMSElong, aes(Measure, coin, fill = RMSE)) +
  geom_tile() +
  scale_fill_distiller() #drawing comparison heatmap on all coins
```


xrp
xmr
vtc
rdd
ppc
nvc
nmc
ltc
ftc
eth
dash
btc
blk
coin
LSTMexpand
GARCHexpand
LSTMnest
GARCHnest
LSTMholdout
GARCHholdout
Measure
RMSE
0.08
0.06
0.04