ST326 Project

Candidate Number: 50537

Q1

The constituents I have chosen for this project are: Apple Inc. (AAPL), Nvidia Corp (NVDA), Microsoft Corp (MSFT), Amazon.com Inc (AMZN), Meta Platforms, Inc. Class A (META), Tesla, Inc. (TSLA), Alphabet Inc. Class A (GOOGL), Eli Lilly & Co. (LLY), Broadcom Inc. (AVGO), and Jpmorgan Chase & Co. (JPM). Also shown is the S&P500 (GSPC).

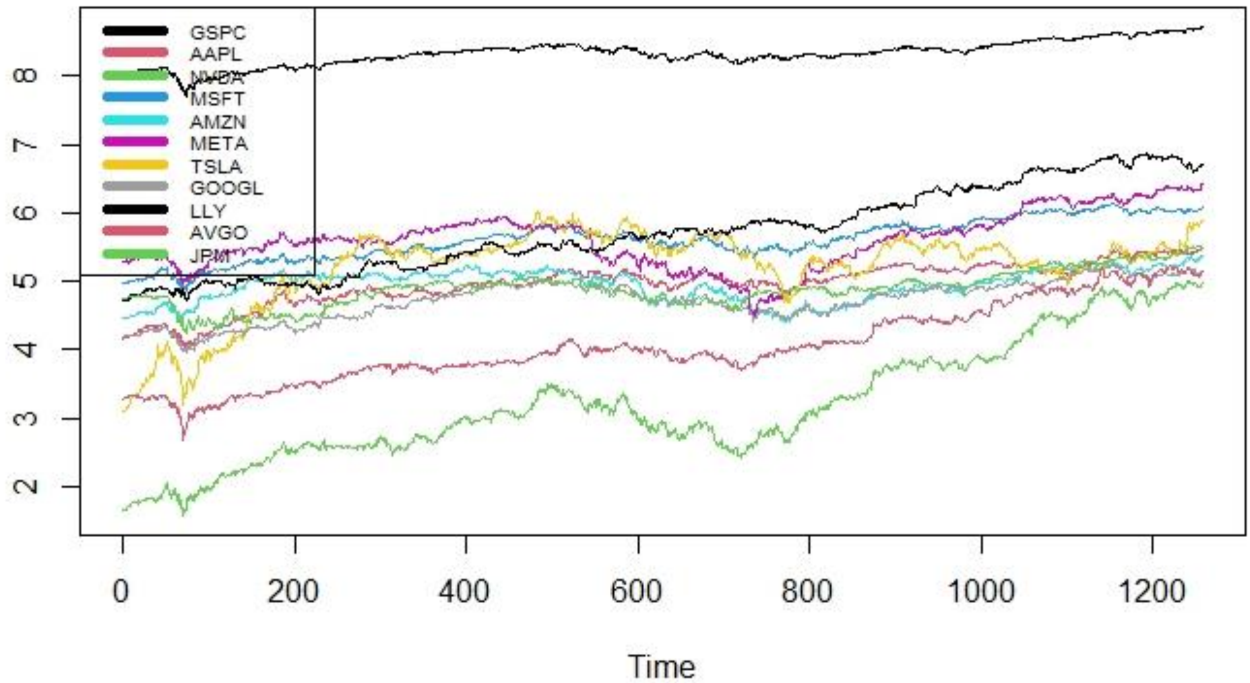


Figure 1 (daily closing prices between 2019-12-04 and 2024-12-04)

Figure 1 shows a common trend at around time 80 (which translates to February 2020) which is likely due to the covid-19 pandemic. There's also a slight fall around time 750 (which is December 2021) which could be to the Omicron variant of covid-19 from South Africa, which was more transmissible, being announced or inflation concerns. Otherwise, there is a general upwards trend for all stocks.

Q2

I chose to use 0.95 as the lambda tuning parameter in 'first.acf.squares.train' as it the produces the smallest ss value (0.6879487).

The sum of exponentially smoothed square residuals are as follows:

Training data (50% of total data)	8121.848
Validation data (25% of total data)	4129.534
Test data (25% of total data)	4297.076

Q3

Prediction algorithm:

i. Start from a certain $t = t_0$. Fix D. We can write model

$Y_{s+1} = a_0 + \Sigma^{q+10}_{k=1} a_k Z^k_s + e_{s+1}, \quad s = t - D, \ldots, t - 1$

as

$\mathbf{Y} = \mathbf{Z}a + \mathbf{e}$,

where $\mathbf{Y} = (Y_{t-D+1}, \ldots, Y_t)^T$, $\mathbf{Z} = (\mathbf{1}_D, \mathbf{Z}^1, \ldots, \mathbf{Z}^{q+10})$ with $\mathbf{Z}^k = (Z^k_{t-D}, \ldots, Z^k_{t-1})^T$, $a = (a_0, a_1, \ldots, a_{q+10})^T$ and $\mathbf{e} = (e_{t-D+1}, \ldots, e_t)^T$. The vector $\mathbf{1}_D$ is a column vector of ones of dimension D. Estimate a by the least square estimator

$\hat{a} = \hat{a}_D(t) = (\mathbf{Z}^T\mathbf{Z})^{-1}\mathbf{Z}^T\mathbf{Y}$.

ii. The predicted return at time $t + 1$ is then

$\hat{Y}_{t+1} = \hat{a}^T\mathbf{z}_t$,

where $\mathbf{z}_t = (1, Z^1_t, \ldots, Z^{q+10}_t)^T$.

iii. We take the predicted return as the new position (ignore all transaction costs to achieve this). We simplify the procedure by not multiplying back the forecasted volatility effect at time $t + 1$. With this, the actual return at time $t + 1$ is

$R_{t+1} = \hat{Y}_{t+1}Y_{t+1}$.

iv. Return to step i., but with t replaced by $t + 1$. Stop when t reaches the end of the training set.

v. Repeat steps i to iv for a grid of values of D.

This tuning parameter D is to be determined using the validation set. We can still see which D is doing the best job under the training set. Here we use the Sharpe ratio, defined by

(Annualised) Sharpe ratio = sqrt(250) x Average daily return / SD of daily returns

Q4

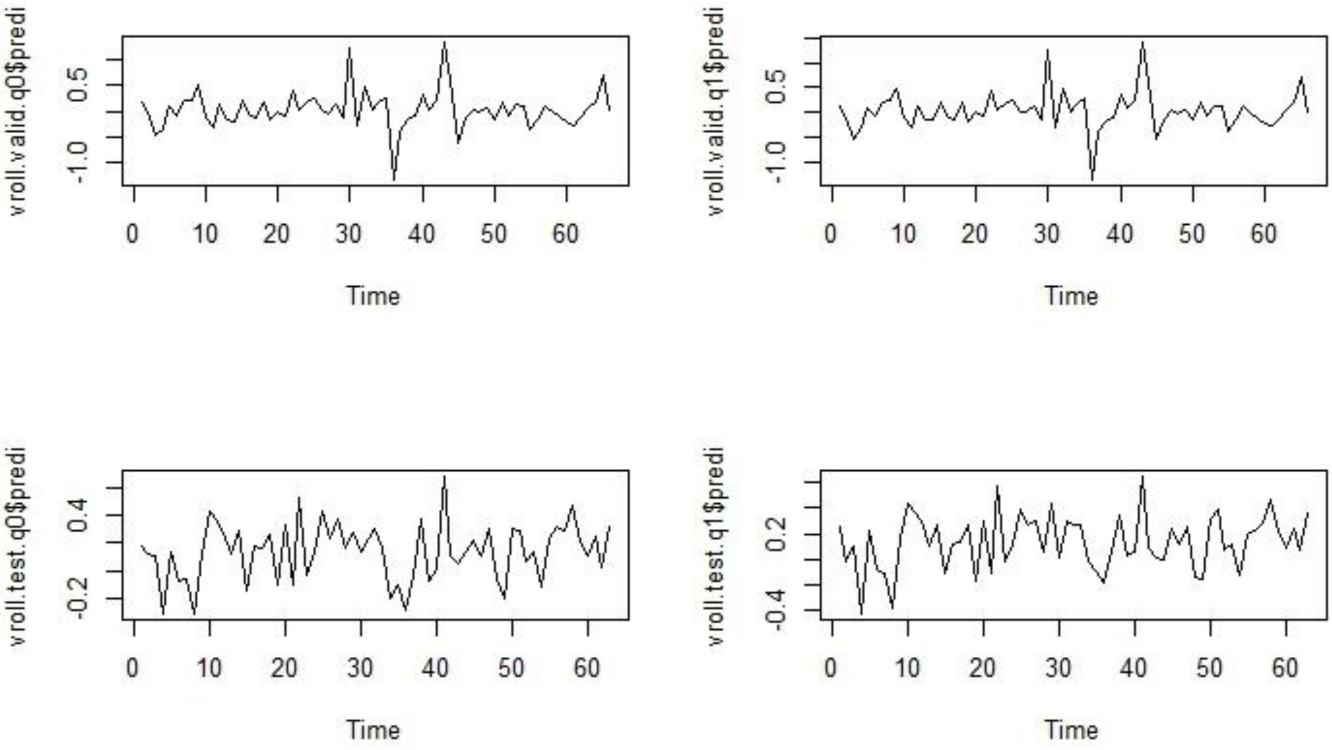


Figure 2. *q=0 used in 'pred.footsie.prepare' on the left and q=1 on the right. Validation data on the top row and test data on the bottom.*

```
> sc$train.curve
 [1] -0.9478990 -1.5939193 -2.3934799 -1.1244269 -0.7623283 -0.2302294 -0.3768879 -0.4012880 -0.7710883
[10] -0.9324770 -1.2319600
> sc$valid.curve
 [1] -0.2872610  0.9169223  1.4882568  0.9034626  0.6652047  0.8263377  1.4730729  1.7642369  1.6617927
[10]  1.8494339  2.1597516
> sc$test.curve
 [1]  0.80364175 -1.48247720 -0.03232336 -0.55351630 -1.43094097 -1.27096030 -1.11888632 -0.68730041
 [9] -0.56525039 -0.29350081  0.11199456
```

Figure 3. *Output of Sharpe curves to find suitable tuning parameters*

```
> sim.grid(data, l[1], 500, warmup = 500) # choose theta=0.1 or higher
  [1] -2.7766403 -3.2022633 -2.0264224 -2.1545972 -2.5442909 -2.6118275 -2.5954607 -1.6278209
  [9] -1.1563025 -1.5473364 -1.2540835 -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518
 [17] -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518
 [25] -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518
 [33] -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518
 [41] -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518
 [49] -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518
 [57] -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518
 [65] -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518
 [73] -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518
 [81] -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518
 [89] -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518
 [97] -0.8383518 -0.8383518 -0.8383518 -0.8383518 -0.8383518
```

Figure 4. *Used to find suitable tuning parameter, theta >= 0.1*

Using least squares is appropriate because, as the algorithm from question 3 specifies, we are using least squares in order to find â.

Q5

```
> train.pca
Standard deviations (1, .., p=11):
 [1] 2.7043156 1.2095899 1.1393054 1.0013841 0.9065743 0.8344564 0.7314880 0.6657191 0.5906530
[10] 0.5142117 0.3450226

Rotation (n x k) = (11 x 11):
            PC1          PC2          PC3         PC4          PC5         PC6         PC7
 [1,] 0.3737879 -0.188418746  0.099439801  0.16172989  0.013562829 -0.05058541  0.07712212
 [2,] 0.3230585  0.032553331  0.020930725 -0.02485843  0.045611486 -0.13765221  0.16696751
 [3,] 0.3135290  0.049180912  0.073056587 -0.25693597 -0.040124460 -0.37783553 -0.32583178
 [4,] 0.3428792 -0.007158874 -0.001401281 -0.17629505  0.158770271 -0.23273584  0.43579337
 [5,] 0.3248384  0.275802105 -0.048803505 -0.17285049  0.613293870  0.56954081 -0.29061123
 [6,] 0.3116970  0.474519634 -0.434184374  0.52420320 -0.399415477  0.06414107 -0.14299653
 [7,] 0.2307100  0.079769798  0.339153718 -0.44171179 -0.636240423  0.46019166  0.10170268
 [8,] 0.3453464  0.062726989 -0.010931912  0.12106759  0.124058768 -0.03892584  0.51959988
 [9,] 0.1714813 -0.602300364 -0.705689827 -0.25118503 -0.113197142  0.14988877 -0.06242942
[10,] 0.3194320 -0.097206590  0.195484767 -0.04710096 -0.046407494 -0.36176118 -0.52243956
[11,] 0.1828669 -0.526517061  0.376934174  0.54441169  0.009791614  0.29401894 -0.08955964
               PC8          PC9        PC10         PC11
 [1,]  0.052370160 -0.010788229  0.04289266 -0.880537741
 [2,]  0.841148016  0.020626784 -0.31981553  0.185400540
 [3,] -0.241826134  0.679801126 -0.23455189  0.042040539
 [4,] -0.038879138  0.090235476  0.72945069  0.200645191
 [5,] -0.007970882 -0.007769713  0.03215878 -0.005920213
 [6,] -0.008307436  0.056044431  0.15242564  0.061908252
 [7,] -0.019677869 -0.051731099  0.01316086  0.010812979
 [8,] -0.473115944 -0.239376384 -0.52009168  0.153295006
 [9,] -0.020381718 -0.036277696 -0.05103417  0.056769148
[10,] -0.066125037 -0.639159913  0.07845910  0.151854019
[11,] -0.024279953  0.236994021  0.07119478  0.307388496
```

Figure 5. *Training principal component analysis*

```
> valid.pca
Standard deviations (1, .., p=11):
 [1] 2.7941699 1.1858379 1.1135223 0.9965134 0.9146546 0.7881892 0.7500393 0.6628091 0.5698091
[10] 0.4939361 0.3140114

Rotation (n x k) = (11 x 11):
             PC1         PC2         PC3         PC4          PC5         PC6         PC7         PC8
 [1,] -0.33800239 -0.05698204 -0.14186696  0.07906954 -0.007330716  0.13638336 -0.22765213 -0.09551264
 [2,] -0.30336982 -0.06958413 -0.02553061 -0.06781852  0.088315410  0.11204706 -0.72195873  0.29484280
 [3,] -0.34736734  0.18654944 -0.30124955 -0.04052705  0.351672480 -0.33001739  0.48915587  0.22987570
 [4,] -0.30883812 -0.03161884  0.07892130  0.01954272  0.367455934  0.33731582  0.05451042  0.08059001
 [5,] -0.29378919  0.03316475  0.23998998  0.01650687  0.004839612  0.49464742  0.22811613 -0.60669825
 [6,] -0.34374285  0.01508961  0.60380402  0.26718223 -0.175648332 -0.59346350 -0.08403248 -0.15313010
 [7,] -0.30184088  0.17040753 -0.07097440 -0.78977427 -0.461050023 -0.07230287  0.03126068 -0.04681536
 [8,] -0.35951422 -0.01213434  0.30136926  0.06327257 -0.034018114  0.21804804  0.23037662  0.53511150
 [9,] -0.08353955 -0.95650485 -0.04222161 -0.18051907  0.038651793 -0.12696130  0.14588713 -0.04625816
[10,] -0.28374545  0.07528710 -0.35984687  0.02446013  0.355241497 -0.27216176 -0.20312336 -0.40160110
[11,] -0.25564843 -0.07090424 -0.48263277  0.50451286 -0.600964771  0.07945284  0.09494234  0.03341600
               PC9        PC10        PC11
 [1,]  0.006330425 -0.07337200  0.87812491
 [2,]  0.158560011  0.42214323 -0.25693699
 [3,]  0.267783381  0.38838785  0.06990804
 [4,]  0.447109570 -0.63384229 -0.19254559
 [5,]  0.000959236  0.41824972 -0.12234123
 [6,]  0.161551770 -0.07625644 -0.01311328
 [7,]  0.037769410 -0.16214397 -0.04890460
 [8,] -0.621501581 -0.03567253 -0.01089999
 [9,] -0.006250379  0.04328975 -0.02829734
[10,] -0.528673205 -0.21992955 -0.23034074
[11,]  0.099391541 -0.07250104 -0.22228739
```

Figure 6. *Validation principal component analysis*

```
> test.pca
Standard deviations (1, .., p=11):
 [1] 2.7270702 1.6415113 1.2244569 1.1369602 1.0929411 1.0094806 0.8616936 0.8064126 0.7359458
[10] 0.6913685 0.3949916

Rotation (n x k) = (11 x 11):
             PC1         PC2         PC3         PC4         PC5         PC6         PC7         PC8
 [1,] -0.3722608  0.107982648  0.14422874  0.18349396 -0.06829209 -0.02243217 -0.04258894 -0.08904344
 [2,] -0.2548275 -0.125546224  0.07227050  0.02494367 -0.52345427 -0.27519362 -0.61767634 -0.07820073
 [3,] -0.2551766 -0.063929719 -0.09667133  0.34945180  0.03472759 -0.11716705  0.51203558 -0.52461854
 [4,] -0.2866710  0.017252364 -0.18334896  0.12031181 -0.24437974 -0.01003297 -0.02542340 -0.40491651
 [5,] -0.3787769  0.207547804 -0.20770142 -0.29849190  0.11996948  0.15309568  0.12300294  0.07779987
 [6,] -0.3943760  0.169674027 -0.40160942 -0.41003402  0.33611199  0.05497835 -0.26096463 -0.07988506
 [7,] -0.2182571  0.161936307  0.53117201 -0.47711124 -0.10265789 -0.49504584  0.34276974  0.07195082
 [8,] -0.2770485  0.081045002 -0.17885847  0.12242694 -0.53202513  0.32650906  0.31970661  0.55040031
 [9,] -0.1457090  0.006518917 -0.29462484  0.41357066  0.29797282 -0.64153039 -0.02274900  0.44796166
[10,] -0.4389625 -0.628147313  0.39637939  0.10133581  0.33492374  0.27138060 -0.06973646  0.15899073
[11,] -0.1089677  0.684780746  0.41052208  0.38465289  0.19618694  0.21584336 -0.21335178  0.01656347
              PC9        PC10        PC11
 [1,]  0.08934319 -0.02738068  0.87843510
 [2,] -0.14433299  0.35571212 -0.16946223
 [3,] -0.36194699  0.31067059 -0.13955278
 [4,]  0.48055770 -0.59395717 -0.24755535
 [5,]  0.54299479  0.56331704 -0.10015866
 [6,] -0.48878132 -0.23814767  0.01267948
 [7,] -0.04829972 -0.16241826 -0.09680932
 [8,] -0.23563953 -0.12293051 -0.06517299
 [9,]  0.11161170 -0.05457577 -0.06253008
[10,]  0.02738867 -0.06953127 -0.15430481
[11,] -0.06654154 -0.00852876 -0.25950649
```

Figure 7. *Test principal component analysis*

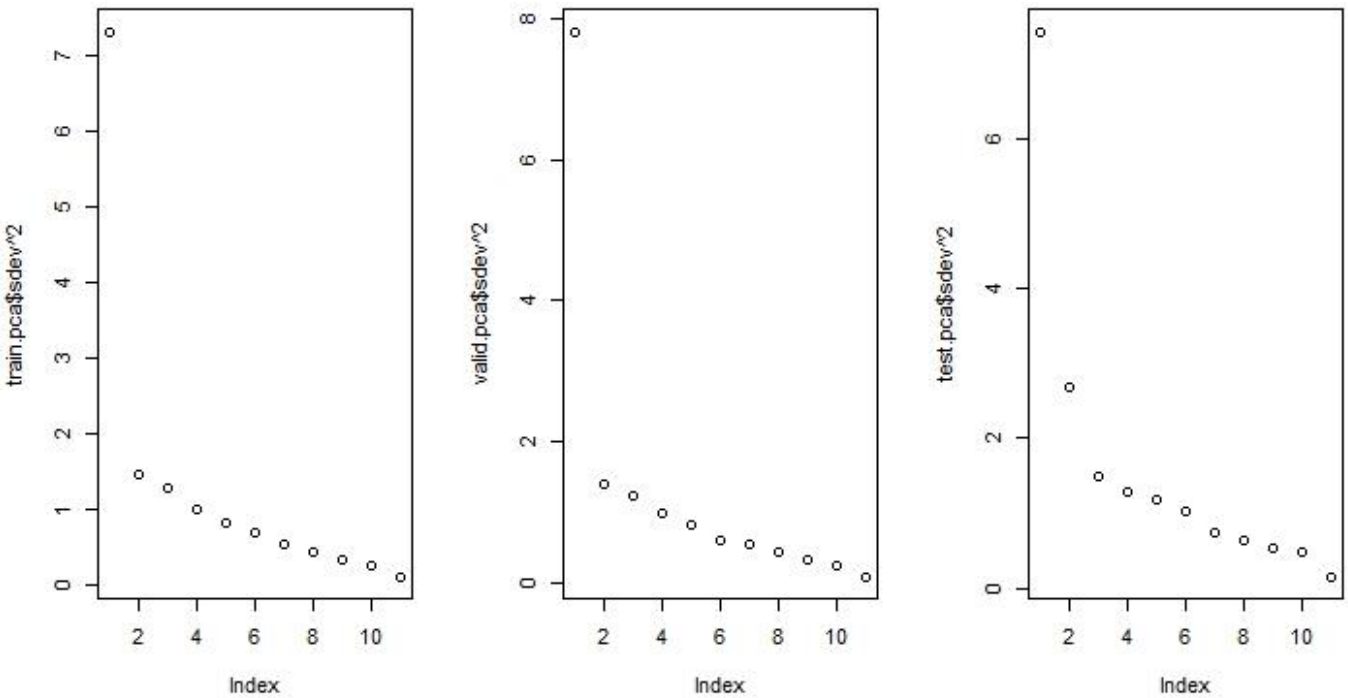


Figure 8. *Scree plots*

Principal component analysis (PCA) is better than just using ordinary least squares (OLS) for high dimensional data whereas OLS is better for more standard (large dataset, less predictors) linear regression.

Appendix

R Code:

ST326_script

50537

2024-12-13

```
##### ST326 PROJECT #####



### Q1 ###

# chosen constituents:
#  Apple Inc. (AAPL), Nvidia Corp (NVDA), Microsoft Corp (MSFT), Amazon.com Inc
#  (AMZN), Meta Platforms, Inc. Class A (META), Tesla, Inc. (TSLA), Alphabet
#  Inc. Class A (GOOGL), Eli Lilly & Co. (LLY), Broadcom Inc. (AVGO), Jpmorgan
#  Chase & Co. (JPM).

setwd("C:/Users/olive/Documents/ST326/Project")

library(quantmod)

## Warning: package 'quantmod' was built under R version 4.3.3

## Loading required package: xts

## Warning: package 'xts' was built under R version 4.3.3

## Loading required package: zoo

## Warning: package 'zoo' was built under R version 4.3.3

##
## Attaching package: 'zoo'

## The following objects are masked from 'package:base':
##
##     as.Date, as.Date.numeric

## Loading required package: TTR

## Warning: package 'TTR' was built under R version 4.3.3

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

constituents <- c('GSPC', 'AAPL', 'NVDA', 'MSFT', 'AMZN', 'META', 'TSLA', 'GOOGL',
                  'LLY', 'AVGO', 'JPM')
start_date <- as.numeric(as.Date('2019-12-04'))
end_date <- as.numeric(as.Date('2024-12-04'))


getSymbols('^GSPC')

## [1] "GSPC"
```

```r
GSPC = as.data.frame(GSPC)
sum(is.na(GSPC))
```

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

```r
GSPC = cbind(as.numeric(as.Date(rownames(GSPC))), GSPC)
GSPC <- GSPC[which(GSPC[1] > start_date & GSPC[1] <= end_date), ]
write.table(GSPC, "data/GSPC.txt", row.names=FALSE)

getSymbols('AAPL')
```

```
## [1] "AAPL"
```

```r
AAPL = as.data.frame(AAPL)
sum(is.na(AAPL))
```

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

```r
AAPL = cbind(as.numeric(as.Date(rownames(AAPL))), AAPL)
AAPL <- AAPL[which(AAPL[1] > start_date & AAPL[1] <= end_date), ]
write.table(AAPL, "data/AAPL.txt", row.names=FALSE)

getSymbols('NVDA')
```

```
## [1] "NVDA"
```

```r
NVDA = as.data.frame(NVDA)
sum(is.na(NVDA))
```

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

```r
NVDA = cbind(as.numeric(as.Date(rownames(NVDA))), NVDA)
NVDA <- NVDA[which(NVDA[1] > start_date & NVDA[1] <= end_date), ]
write.table(NVDA, "data/NVDA.txt", row.names=FALSE)

getSymbols('MSFT')
```

```
## [1] "MSFT"
```

```r
MSFT = as.data.frame(MSFT)
sum(is.na(MSFT))
```

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

```r
MSFT = cbind(as.numeric(as.Date(rownames(MSFT))), MSFT)
MSFT <- MSFT[which(MSFT[1] > start_date & MSFT[1] <= end_date), ]
write.table(MSFT, "data/MSFT.txt", row.names=FALSE)

getSymbols('AMZN')
```

```
## [1] "AMZN"
```

```r
AMZN = as.data.frame(AMZN)
sum(is.na(AMZN))
```

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

```r
AMZN = cbind(as.numeric(as.Date(rownames(AMZN))), AMZN)
AMZN <- AMZN[which(AMZN[1] > start_date & AMZN[1] <= end_date), ]
write.table(AMZN, "data/AMZN.txt", row.names=FALSE)

getSymbols('META')
```

```
## [1] "META"
```

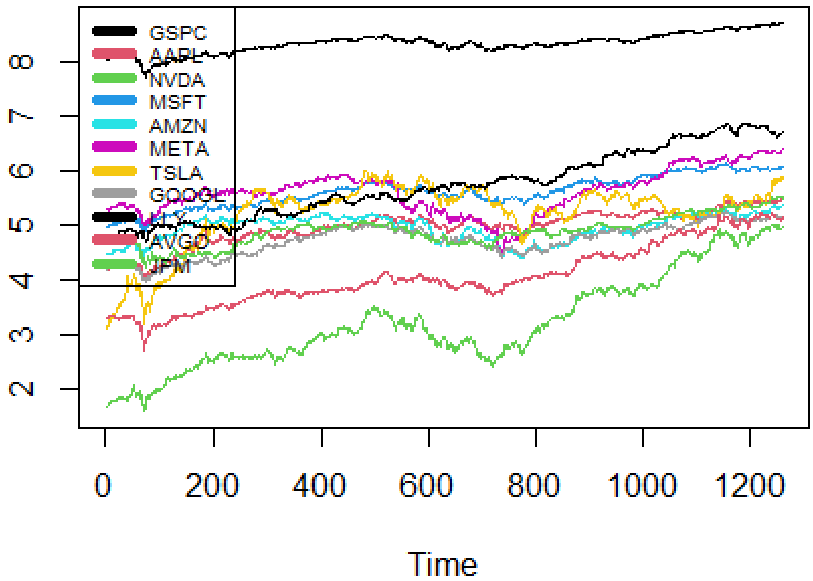
```r
META = as.data.frame(META)
sum(is.na(META))

## [1] 0

META = cbind(as.numeric(as.Date(rownames(META))), META)
META <- META[which(META[1] > start_date & META[1] <= end_date), ]
write.table(META, "data/META.txt", row.names=FALSE)

getSymbols('TSLA')

## [1] "TSLA"

TSLA = as.data.frame(TSLA)
sum(is.na(TSLA))

## [1] 0

TSLA = cbind(as.numeric(as.Date(rownames(TSLA))), TSLA)
TSLA <- TSLA[which(TSLA[1] > start_date & TSLA[1] <= end_date), ]
write.table(TSLA, "data/TSLA.txt", row.names=FALSE)

getSymbols('GOOGL')

## [1] "GOOGL"

GOOGL = as.data.frame(GOOGL)
sum(is.na(GOOGL))

## [1] 0

GOOGL = cbind(as.numeric(as.Date(rownames(GOOGL))), GOOGL)
GOOGL <- GOOGL[which(GOOGL[1] > start_date & GOOGL[1] <= end_date), ]
write.table(GOOGL, "data/GOOGL.txt", row.names=FALSE)

getSymbols('LLY')

## [1] "LLY"

LLY = as.data.frame(LLY)
sum(is.na(LLY))

## [1] 0

LLY = cbind(as.numeric(as.Date(rownames(LLY))), LLY)
LLY <- LLY[which(LLY[1] > start_date & LLY[1] <= end_date), ]
write.table(LLY, "data/LLY.txt", row.names=FALSE)

getSymbols('AVGO')

## [1] "AVGO"

AVGO = as.data.frame(AVGO)
sum(is.na(AVGO))

## [1] 0

AVGO = cbind(as.numeric(as.Date(rownames(AVGO))), AVGO)
AVGO <- AVGO[which(AVGO[1] > start_date & AVGO[1] <= end_date), ]
write.table(AVGO, "data/AVGO.txt", row.names=FALSE)

getSymbols('JPM')

## [1] "JPM"
```

```r
JPM = as.data.frame(JPM)
sum(is.na(JPM))
```

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

```r
JPM = cbind(as.numeric(as.Date(rownames(JPM))), JPM)
JPM <- JPM[which(JPM[1] > start_date & JPM[1] <= end_date), ]
write.table(JPM, "data/JPM.txt", row.names=FALSE)


read.bossa.data <- function(vec.names) {
  p <- length(vec.names)
  n1 <- 20000
  dates <- matrix(99999999, p, n1)
  closes <- matrix(0, p, n1)
  max.n2 <- 0

  for (i in 1:p) {
    filename <- paste("data/", vec.names[i], ".txt", sep="")
    tmp <- scan(filename, list(date=numeric(), NULL, NULL, NULL, NULL, NULL,
                               close=numeric()), skip=1, sep="")
    n2 <- length(tmp$date)
    max.n2 <- max(n2, max.n2)
    dates[i,1:n2] <- tmp$date
    closes[i,1:n2] <- tmp$close
  }

  dates <- dates[,1:max.n2]
  closes <- closes[,1:max.n2]

  days <- rep(0, n1)
  arranged.closes <- matrix(0, p, n1)
  date.indices <- starting.indices <- rep(1, p)
  already.started <- rep(0, p)
  day <- 1


  while(max(date.indices) <= max.n2) {
    current.dates <- current.closes <- rep(0, p)
    for (i in 1:p) {
      current.dates[i] <- dates[i,date.indices[i]]
      current.closes[i] <- closes[i,date.indices[i]]
    }
    min.indices <- which(current.dates == min(current.dates))
    days[day] <- current.dates[min.indices[1]]
    arranged.closes[min.indices,day] <- log(current.closes[min.indices])
    arranged.closes[-min.indices,day] <- arranged.closes[-min.indices, max(day-1, 1)]
    already.started[min.indices] <- 1
    starting.indices[-which(already.started == 1)] <- starting.indices[-which(already.
started == 1)] + 1
    day <- day + 1
    date.indices[min.indices] <- date.indices[min.indices] + 1
  }


  days <- days[1:(day-1)]
  arranged.closes <- arranged.closes[,1:(day-1)]
  max.st.ind <- max(starting.indices)
  r <- matrix(0, p, (day-max.st.ind-1))
```

```
  for (i in 1:p) {
    r[i,] <- diff(arranged.closes[i,max.st.ind:(day-1)])
    r[i,] <- r[i,] / sqrt(var(r[i,]))
    r[i,r[i,]==0] <- rnorm(sum(r[i,]==0))
  }

  return(list(dates=dates, closes=closes, days=days, arranged.closes=arranged.closes,
starting.indices=starting.indices, r=r))
}


ind <- read.bossa.data(constituents)

p1 <- ts.plot(t(ind$arranged.closes), col=1:11)
par(new=TRUE)
legend("topleft", constituents, lty=c(rep(1,11)), col=1:11, lwd=5, cex=0.6, bg='transp
arent')
```




```
dates <- as.Date(c(1:length(ind$arranged.closes)), origin='2019-12-04')

as.Date(80, origin='2019-12-04')

## [1] "2020-02-22"

# there is a clear drop around the start of 2020 which is likely due to covid-19
as.Date(750, origin='2019-12-04')

## [1] "2021-12-23"

### Q2 ###

pred.footsie.prepare <- function(max.lag = 5, split = c(50, 25), mask = rep(1, 10)) {

  # this function prepares the data for the prediction exercise and splits them into a
train, validation and test sets
  # max.lag - the maximum lag to include in the prediction
  # split - how much of the data (in percentage terms) to include in the training and
```

```
validation sets, respectively
  # mask - which other indices to include (1 for yes, 0 for no)

  ind <- read.bossa.data(constituents)

  d <- dim(ind$r)

  start.index <- max(3, max.lag + 1)

  y <- matrix(0, d[2] - start.index + 1, 1)

  x <- matrix(0, d[2] - start.index + 1, d[1] - 1 + max.lag)

  y[,1] <- ind$r[1,start.index:d[2]]

  for (i in 1:max.lag) {

    x[,i] <- ind$r[1,(start.index-i):(d[2]-i)]

  }

  shift.indices <- c(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)  ## For American and Latin Ameri
can exchanges, we look at data up to t-1 if t is current time

  for (i in 2:(d[1])) {
    x[,i+max.lag-1] <- ind$r[i,(start.index-1-shift.indices[i-1]):(d[2]-1-shift.indice
s[i-1])]

  }

  end.training <- round(split[1] / 100 * d[2])

  end.validation <- round(sum(split[1:2]) / 100 * d[2])

  x <- x[,as.logical(c(rep(1, max.lag), mask))]

  y.train <- as.matrix(y[1:end.training], end.training, 1)
  x.train <- x[1:end.training,]

  y.valid <- as.matrix(y[(end.training+1):(end.validation)], end.validation-end.traini
ng, 1)
  x.valid <- x[(end.training+1):(end.validation),]

  y.test <- as.matrix(y[(end.validation+1):(d[2] - start.index + 1)], d[2]-start.index
-end.validation+1, 1)
  x.test <- x[(end.validation+1):(d[2] - start.index + 1),]

  list(x=x, y=y, x.train=x.train, y.train=y.train, x.valid=x.valid, y.valid=y.valid, x
.test=x.test, y.test=y.test)

}


q=5
data <- pred.footsie.prepare(q)


vol.exp.sm <- function(x, lambda) {
```

```r
  # Exponential smoothing of x^2 with parameter lambda

  sigma2 <- x^2
  n <- length(x)

  for (i in 2:n)
    sigma2[i] <- sigma2[i-1] * lambda + x[i-1]^2 * (1-lambda)

  sigma <- sqrt(sigma2)

  resid <- x/sigma
  resid[is.na(resid)] <- 0
  sq.resid <- resid^2

  list(sigma2=sigma2, sigma=sigma, resid = resid, sq.resid = sq.resid)

}

first.acf.squares.train <- function(x, lambda) {

  # x is an object returned by "pred.footsie.prepare"
  # this function computes the volatility for each covariate and the response in the t
raining part
  # it then computes the acfs of the squared residuals after removing the volatility,
and adds up
  # the first acfs for each covariate and response
  # the point is to choose lambda so that as much as possible of the acf has been remo
ved

  d <- dim(x$x.train)

  ss <- 0

  x.train.dev <- x$x.train
  y.train.dev <- x$y.train

  x.valid.dev <- x$x.valid
  y.valid.dev <- x$y.valid

  x.test.dev <- x$x.test
  y.test.dev <- x$y.test


  for (i in 1:(d[2])) {
    v <- vol.exp.sm(x$x.train[,i], lambda)
    ss <- ss + abs(acf(v$sq.resid, plot=FALSE)$acf[2])
    x.train.dev[,i] <- v$resid

    v <- vol.exp.sm(x$x.valid[,i], lambda)
    x.valid.dev[,i] <- v$resid

    v <- vol.exp.sm(x$x.test[,i], lambda)
    x.test.dev[,i] <- v$resid

  }

  v <- vol.exp.sm(x$y.train, lambda)
  ss <- ss + abs(acf(v$sq.resid, plot=FALSE)$acf[2])
  y.train.dev <- v$resid
```

```
  v <- vol.exp.sm(x$y.valid, lambda)
  y.valid.dev <- v$resid

  v <- vol.exp.sm(x$y.test, lambda)
  y.test.dev <- v$resid

  list(ss=ss, y.train.dev=y.train.dev, x.train.dev=x.train.dev, y.valid.dev=y.valid.de
v, x.valid.dev=x.valid.dev, y.test.dev=y.test.dev, x.test.dev=x.test.dev)

}

l <- rep(100, 11)

for (i in 1:11) {
  for (j in seq(0.5, 1, by=0.01)) { # bestchoice to 2dp
    if (first.acf.squares.train(data, j)$ss < l[i]) {
      l[i] <- j
    }
  }
}


first.acf.squares.train(data, l[1])$ss

## [1] 0.8070244

summary(first.acf.squares.train(data, l[1])$x.train.dev)

##        V1                  V2                  V3                  V4           
##  Min.   :-4.99024   Min.   :-4.99024   Min.   :-4.99024   Min.   :-4.99024  
##  1st Qu.:-0.50783   1st Qu.:-0.51581   1st Qu.:-0.51581   1st Qu.:-0.51581  
##  Median : 0.14389   Median : 0.16036   Median : 0.16036   Median : 0.15178  
##  Mean   : 0.02953   Mean   : 0.04267   Mean   : 0.05278   Mean   : 0.04017  
##  3rd Qu.: 0.71036   3rd Qu.: 0.74508   3rd Qu.: 0.74508   3rd Qu.: 0.74508  
##  Max.   : 2.53396   Max.   : 2.94038   Max.   : 6.39489   Max.   : 2.80643  
##        V5                  V6                  V7                  V8           
##  Min.   :-4.99024   Min.   :-3.31796   Min.   :-4.06062   Min.   :-3.79168  
##  1st Qu.:-0.48879   1st Qu.:-0.49686   1st Qu.:-0.51130   1st Qu.:-0.53577  
##  Median : 0.14389   Median : 0.05958   Median : 0.10626   Median : 0.08180  
##  Mean   : 0.03354   Mean   : 0.09950   Mean   : 0.08005   Mean   : 0.07006  
##  3rd Qu.: 0.71381   3rd Qu.: 0.75066   3rd Qu.: 0.72991   3rd Qu.: 0.75679  
##  Max.   : 2.53430   Max.   : 5.30528   Max.   : 4.92000   Max.   : 3.69920  
##        V9                V10                  V11                 V12          
##  Min.   :-6.31732   Min.   :-13.704881   Min.   :-4.11196   Min.   :-4.15497  
##  1st Qu.:-0.58333   1st Qu.: -0.556473   1st Qu.:-0.45397   1st Qu.:-0.53007  
##  Median : 0.04137   Median :  0.050423   Median : 0.08114   Median : 0.09378  
##  Mean   : 0.03399   Mean   : -0.008535   Mean   : 0.12001   Mean   : 0.05801  
##  3rd Qu.: 0.66371   3rd Qu.:  0.656177   3rd Qu.: 0.70747   3rd Qu.: 0.68061  
##  Max.   : 7.31733   Max.   :  4.774362   Max.   : 6.68959   Max.   : 4.04160  
##       V13                V14                  V15            
##  Min.   :-6.59238   Min.   :-4.17055   Min.   :-4.940026  
##  1st Qu.:-0.46278   1st Qu.:-0.51773   1st Qu.:-0.624261  
##  Median : 0.02630   Median : 0.08089   Median :-0.025197  
##  Mean   : 0.09657   Mean   : 0.05191   Mean   :-0.003163  
##  3rd Qu.: 0.58057   3rd Qu.: 0.73777   3rd Qu.: 0.653781  
##  Max.   : 9.61709   Max.   : 4.61400   Max.   : 5.796735  

summary(first.acf.squares.train(data, l[1])$x.valid.dev)
```

```
##        V1                 V2                 V3                 V4
##  Min.   :-3.45423   Min.   :-3.46918   Min.   :-3.49127   Min.   :-3.49251
##  1st Qu.:-0.55876   1st Qu.:-0.55936   1st Qu.:-0.61115   1st Qu.:-0.61391
##  Median :-0.02924   Median :-0.03232   Median :-0.04236   Median :-0.04236
##  Mean   : 0.04105   Mean   : 0.04023   Mean   : 0.02529   Mean   : 0.02546
##  3rd Qu.: 0.65505   3rd Qu.: 0.65505   3rd Qu.: 0.67662   3rd Qu.: 0.68336
##  Max.   : 3.46893   Max.   : 3.46923   Max.   : 3.46968   Max.   : 3.46970
##        V5                 V6                 V7                 V8
##  Min.   :-4.88219   Min.   :-5.53699   Min.   :-2.86704   Min.   :-3.67965
##  1st Qu.:-0.61526   1st Qu.:-0.60515   1st Qu.:-0.50550   1st Qu.:-0.53178
##  Median :-0.04129   Median : 0.07035   Median : 0.06879   Median :-0.01446
##  Mean   : 0.01963   Mean   : 0.04784   Mean   : 0.10363   Mean   : 0.04610
##  3rd Qu.: 0.73672   3rd Qu.: 0.60152   3rd Qu.: 0.64277   3rd Qu.: 0.59397
##  Max.   : 3.46977   Max.   : 3.98537   Max.   :10.12230   Max.   : 4.83767
##        V9                V10                V11                V12
##  Min.   :-3.20048   Min.   :-8.72157   Min.   :-3.95148   Min.   :-5.309189
##  1st Qu.:-0.52749   1st Qu.:-0.39441   1st Qu.:-0.55190   1st Qu.:-0.559597
##  Median : 0.03328   Median : 0.01387   Median : 0.03688   Median :-0.004663
##  Mean   : 0.05238   Mean   : 0.08603   Mean   : 0.01099   Mean   : 0.043242
##  3rd Qu.: 0.66931   3rd Qu.: 0.57871   3rd Qu.: 0.63295   3rd Qu.: 0.622220
##  Max.   : 4.64009   Max.   : 8.35338   Max.   : 3.03732   Max.   : 3.475897
##       V13                V14                V15
##  Min.   :-3.2929   Min.   :-3.58627   Min.   :-4.37209
##  1st Qu.:-0.4455   1st Qu.:-0.59494   1st Qu.:-0.51071
##  Median : 0.1372   Median : 0.01246   Median : 0.06535
##  Mean   : 0.1344   Mean   : 0.08616   Mean   : 0.04266
##  3rd Qu.: 0.6980   3rd Qu.: 0.62741   3rd Qu.: 0.59434
##  Max.   :12.4078   Max.   : 4.61871   Max.   : 4.86511

exp_sm <- vol.exp.sm(data$x, l[1])
exp_sm$sigma2[1]

## [1] 0.402346

sum(exp_sm$sq.resid)

## [1] 22294.87

exp_sm_train <- vol.exp.sm(data$x.train, l[1])
exp_sm_train$sigma2[1]

## [1] 0.402346

sum(exp_sm_train$sq.resid)

## [1] 11078.12

exp_sm_valid <- vol.exp.sm(data$x.valid, l[1])
exp_sm_valid$sigma2[1]

## [1] 4.816277

sum(exp_sm_valid$sq.resid)

## [1] 5562.332

exp_sm_test <- vol.exp.sm(data$x.test, l[1])
exp_sm_test$sigma2[1]

## [1] 0.008497018

sum(exp_sm_test$sq.resid)
```

```
## [1] 5719.523
```

Q3

('/written/Q3')

```
### Q4 ###

thresh.reg <- function(x, y, th, x.pred = NULL) {

  # estimation of beta in y = a + x beta + epsilon (linear regression)
  # but only using those covariates in x whose marginal correlation
  # with y exceeds th
  # use th = 0 for full regression
  # note the intercept is added
  # x.pred is a new x for which we wish to make prediction

  d <- dim(x)

  ind <- (abs(cor(x, y)) > th)
  n <- sum(ind)

  new.x <- matrix(c(rep(1, d[1]), x[,ind]), d[1], n+1)    ## Adding intercept term

  gram = t(new.x) %*% new.x

  beta <- solve(gram) %*% t(new.x) %*% matrix(y, d[1], 1)

  ind.ex <- c(1, as.numeric(ind))

  ind.ex[ind.ex == 1] <- beta

  condnum = max(svd(gram)$d)/min(svd(gram)$d)

  pr <- 0

  if (!is.null(x.pred)) pr <- sum(ind.ex * c(1, x.pred))

  list(beta = ind.ex, pr=pr, condnum = condnum)

}

rolling.thresh.reg <- function(x, lambda, th, win, warmup, reg.function = thresh.reg)
{

  # performs prediction over a rolling window of size win
  # over the training set
  # x - returned by pred.footsie.prepare
  # lambda - parameter for exponential smoothing
  # th - threshold for thresh.reg
  # warmup - t_0 from the lecture notes


  xx <- first.acf.squares.train(x, lambda)

  n <- length(xx$y.train.dev)

  err <- 0
```

```r
  condnum <- predi <- truth <- rep(0, n-warmup+1)

  for (i in warmup:n) {

    y <- xx$y.train.dev[(i-win):(i-1)]
    xxx <- xx$x.train.dev[(i-win):(i-1),]

    zz <- reg.function(xxx, y, th, xx$x.train.dev[i,])

    predi[i-warmup+1] <- zz$pr
    condnum[i-warmup+1] <- zz$condnum
    truth[i-warmup+1] <- xx$y.train.dev[i]

  }

  ret <- predi * truth

  err <- sqrt(250) * mean(ret) / sqrt(var(ret))

  list(err=err, predi=predi, truth=truth, condnum=condnum)

}

rolling.thresh.reg.valid <- function(x, lambda, th, win, warmup, reg.function = thresh
.reg) {

  # The same as the previous function but for the validation set

  xx <- first.acf.squares.train(x, lambda)

  n <- length(xx$y.valid.dev)

  err <- 0

  condnum <- predi <- truth <- rep(0, n-warmup+1)

  for (i in warmup:n) {

    y <- xx$y.valid.dev[(i-win):(i-1)]
    xxx <- xx$x.valid.dev[(i-win):(i-1),]

    zz <- reg.function(xxx, y, th, xx$x.valid.dev[i,])

    predi[i-warmup+1] <- zz$pr
    condnum[i-warmup+1] <- zz$condnum
    truth[i-warmup+1] <- xx$y.valid.dev[i]

  }


  ret <- predi * truth

  err <- sqrt(250) * mean(ret) / sqrt(var(ret))

  list(err=err, predi=predi, truth=truth, condnum=condnum)

}

rolling.thresh.reg.test <- function(x, lambda, th, win, warmup, reg.function = thresh.
```

```
reg) {

  # The same as the previous function but for the test set

  xx <- first.acf.squares.train(x, lambda)

  n <- length(xx$y.test.dev)

  err <- 0

  condnum <- predi <- truth <- rep(0, n-warmup+1)

  for (i in warmup:n) {

    y <- xx$y.test.dev[(i-win):(i-1)]
    xxx <- xx$x.test.dev[(i-win):(i-1),]
    zz <- reg.function(xxx, y, th, xx$x.test.dev[i,])

    predi[i-warmup+1] <- zz$pr
    condnum[i-warmup+1] <- zz$condnum
    truth[i-warmup+1] <- xx$y.test.dev[i]

  }


  ret <- predi * truth

  err <- sqrt(250) * mean(ret) / sqrt(var(ret))

  list(err=err, predi=predi, truth=truth, condnum=condnum)

}

sharpe.curves <- function(x, lambda, th, warmup, reg.function = thresh.reg, win = seq(
from = 10, to = warmup, by = 10)) {

  # computes Sharpe ratios for a sequence of rolling windows (D in the lecture notes)
  # for the training, validation and test sets

  w <- length(win)

  train.curve <- valid.curve <- test.curve <- rep(0, w)

  n <- length(x$y.train)

  i=1
  rreg <- rolling.thresh.reg(x, lambda, th, win[i], warmup, reg.function)
  rreg.valid <- rolling.thresh.reg.valid(x, lambda, th, win[i], warmup, reg.function)
  rreg.test <- rolling.thresh.reg.test(x, lambda, th, win[i], warmup, reg.function)

  condnum = matrix(0,w,length(rreg$condnum))
  condnum.valid = matrix(0,w,length(rreg.valid$condnum))
  condnum.test = matrix(0,w,length(rreg.test$condnum))

  train.curve[i] <- rreg$err
  valid.curve[i] <- rreg.valid$err
  test.curve[i] <- rreg.test$err

  condnum[i,] <- rreg$condnum
```

```r
  condnum.valid[i,] <- rreg.valid$condnum
  condnum.test[i,] <- rreg.test$condnum

  for (i in 2:w) {
    rreg <- rolling.thresh.reg(x, lambda, th, win[i], warmup, reg.function)
    rreg.valid <- rolling.thresh.reg.valid(x, lambda, th, win[i], warmup, reg.function
)
    rreg.test <- rolling.thresh.reg.test(x, lambda, th, win[i], warmup, reg.function)

    train.curve[i] <- rreg$err
    valid.curve[i] <- rreg.valid$err
    test.curve[i] <- rreg.test$err

    condnum[i,] <- rreg$condnum
    condnum.valid[i,] <- rreg.valid$condnum
    condnum.test[i,] <- rreg.test$condnum
  }

  list(train.curve = train.curve, valid.curve = valid.curve, test.curve = test.curve,
condnum=condnum, condnum.valid = condnum.valid, condnum.test = condnum.test)
}


sc <- sharpe.curves(data, l[1], 0, 250, win = seq(from = 50, to = 250, by = 20))

q=0
data.q0 = pred.footsie.prepare(q)
first.acf.squares.train(data.q0, l[1])$ss

## [1] 0.5328232

q=1
data.q1 = pred.footsie.prepare(q)
first.acf.squares.train(data.q1, l[1])$ss

## [1] 0.8618017

vroll.valid.q0 = rolling.thresh.reg.valid(data.q0, l[1], 0, 250, 250)
vroll.valid.q0$predi

##  [1]  0.1597079959 -0.0402177189 -0.4805836338 -0.3718759645  0.1088853153
##  [6] -0.0874019352  0.2357422040  0.2248919643  0.5316821348 -0.1039701020
## [11] -0.3385942316  0.1404196867 -0.1857659236 -0.1974213492  0.2048665046
## [16] -0.0652181023 -0.1389832481  0.2000685142 -0.1848522642  0.0085046748
## [21] -0.0842184390  0.4053823192  0.0215455309  0.1966942361  0.2551004716
## [26]  0.0140154264 -0.0417534334  0.1620105243 -0.1314700731  1.3176074740
## [31] -0.3073934215  0.5033200890  0.0245259864  0.1930373900  0.2585605861
## [36] -1.3930763538 -0.3801696833 -0.1686909178 -0.0944942183  0.3408677063
## [41]  0.0599924192  0.2524110437  1.4341033531  0.3324057758 -0.6230846303
## [46] -0.1342987556  0.0385815840 -0.0134176612  0.0375007592 -0.2114069757
## [51]  0.1578152699 -0.1124456956  0.1200962641  0.1247862139 -0.3801827213
## [56] -0.1659913223  0.0809517270  0.0003774759 -0.1005715027 -0.1986116008
## [61] -0.2913413441 -0.1621919151  0.0580157201  0.2083287960  0.7353356292
## [66]  0.0057788295

vroll.valid.q1 = rolling.thresh.reg.valid(data.q1, l[1], 0, 250, 250)
vroll.valid.q1$predi

##  [1]  0.121634798 -0.131327917 -0.563504321 -0.319917473  0.102012490
##  [6] -0.081624500  0.184649227  0.229611931  0.476733360 -0.121438465
## [11] -0.284968369  0.145587527 -0.145548296 -0.137329022  0.193064861
```

```
## [16] -0.083948068 -0.119935109  0.220475110 -0.175589258 -0.037515372
## [21] -0.102541868  0.473657865  0.025903419  0.120861024  0.224451003
## [26]  0.018638885 -0.035112750  0.149926970 -0.100896573  1.258516657
## [31] -0.292031956  0.518631108 -0.016660359  0.191729164  0.253601783
## [36] -1.376387966 -0.359462299 -0.210323272 -0.086934774  0.348150479
## [41]  0.101956159  0.246599849  1.286301004  0.225391931 -0.529924372
## [46] -0.141468064  0.025487099 -0.002606828  0.028452336 -0.163382997
## [51]  0.203689961 -0.104009393  0.128216327  0.108919711 -0.367904354
## [56] -0.168132483  0.100199775 -0.037369695 -0.132863099 -0.225275443
## [61] -0.275274186 -0.145289113  0.058192143  0.225762177  0.682242516
## [66] -0.033743361
```

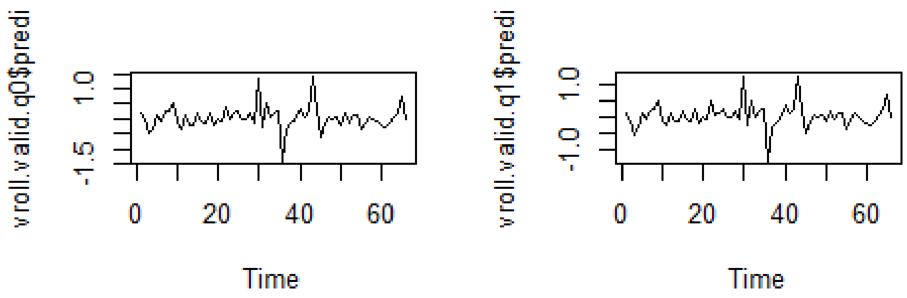
```
vroll.test.q0 = rolling.thresh.reg.test(data.q0, l[1], 0, 250, 250)
vroll.test.q0$predi
```

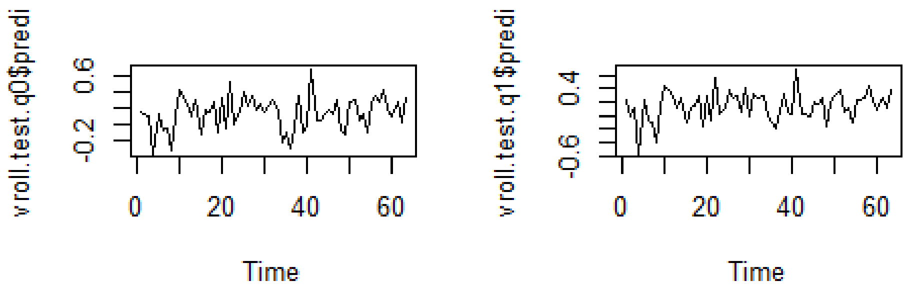
```
##  [1]  0.164143731  0.136985962  0.108698929 -0.356977377  0.138389134
##  [6] -0.071851031 -0.052060765 -0.326489103  0.107425544  0.426759396
## [11]  0.364947472  0.265135789  0.119305229  0.299793058 -0.124221867
## [16]  0.189877028  0.164983796  0.276481955 -0.099279330  0.334726605
## [21] -0.053601172  0.528276146  0.003543128  0.150009710  0.413695786
## [26]  0.235435421  0.362169919  0.177053507  0.249071520  0.154843448
## [31]  0.238680157  0.300314396  0.188485386 -0.207605850 -0.102447671
## [36] -0.290789378 -0.042400210  0.360762244 -0.096701781 -0.028255427
## [41]  0.694919243  0.052695113  0.055396120  0.139976724  0.184531645
## [46]  0.141068653  0.313567189 -0.070446222 -0.105749711  0.286778960
## [51]  0.315649176  0.056460562  0.134328402 -0.100686205  0.254621402
## [56]  0.347386921  0.286911657  0.445084080  0.204827344  0.105730838
## [61]  0.271213304  0.020536119  0.331073165
```

```
vroll.test.q1 = rolling.thresh.reg.test(data.q1, l[1], 0, 250, 250)
vroll.test.q1$predi
```

```
##  [1]  0.241910998 -0.010724653  0.102758272 -0.560321805  0.218986900
##  [6] -0.073084855 -0.107923069 -0.384382096  0.149264190  0.453011515
## [11]  0.379912422  0.294601121  0.100228600  0.273162551 -0.085585489
## [16]  0.124425295  0.139812441  0.284639474 -0.153953990  0.298401417
## [21] -0.068094579  0.567227191  0.019449646  0.125098087  0.369508532
## [26]  0.256724567  0.282070359  0.058249187  0.418567885  0.004793541
## [31]  0.312411177  0.256371978  0.279646180 -0.010800545 -0.108134221
## [36] -0.181657647  0.059574993  0.337562431  0.039964782  0.026414199
## [41]  0.696544799  0.022285311  0.010720155  0.007216858  0.197521109
## [46]  0.174022290  0.263428031 -0.150471984  0.171147234  0.282836449
## [51]  0.395593741  0.058557871  0.119569123 -0.105944094  0.222489767
## [56]  0.237891022  0.270299150  0.449220455  0.224373366  0.078351256
## [61]  0.255470779  0.101068765  0.391155023
```

```
par(mfrow=c(2,2))
plot.ts(vroll.valid.q0$predi)
plot.ts(vroll.valid.q1$predi)
plot.ts(vroll.test.q0$predi)
plot.ts(vroll.test.q1$predi)
```






```
sc$train.curve
```

```
##  [1] -0.9515432 -1.6000513 -2.4268744 -1.1492116 -0.7950692 -0.2301892
##  [7] -0.3783488 -0.4200400 -0.8086515 -0.9950237 -1.2727641
```

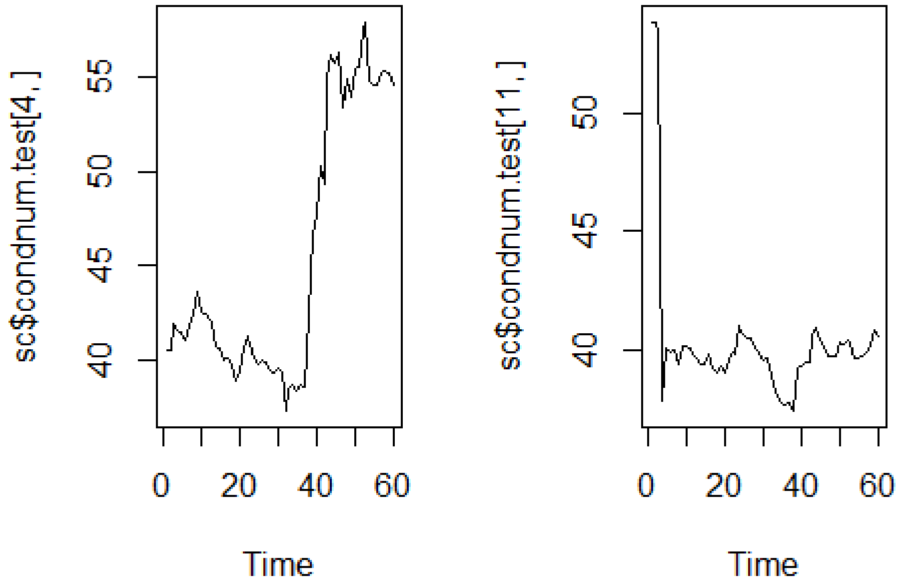
```
sc$valid.curve
```

```
##  [1] -0.1304532  1.2402546  1.7898408  1.2163159  0.9134427  1.0604952
##  [7]  1.6627320  1.9441245  1.8269458  1.9882775  2.3058055
```

```
sc$test.curve
```

```
##  [1]  0.79694289 -1.58342018  0.23411483 -0.33468884 -1.30694422 -1.13498139
##  [7] -0.98854820 -0.46134462 -0.37753009 -0.07550039  0.27712573
```

```
par(mfrow=c(1,2))
plot.ts(sc$condnum.test[4,])
plot.ts(sc$condnum.test[11,]) # very different
```




```
sim.grid <- function(x, lambda, win, warmup = 250, reg.function = thresh.reg, th.grid
= seq(from = 0, to = 1, by = .01)) {

  # Which threshold th best over training set?

  tt <- length(th.grid)

  res <- rep(0, tt)

  for (i in 1:tt) res[i] <- rolling.thresh.reg(x, lambda, th.grid[i], win, warmup, reg
.function)$err

  res

}

sim.grid.valid <- function(x, lambda, win, warmup = 250, reg.function = thresh.reg, th
.grid = seq(from = 0, to = 1, by = .01)) {

  # The same over the validation set.

  tt <- length(th.grid)

  res <- rep(0, tt)

  for (i in 1:tt) res[i] <- rolling.thresh.reg.valid(x, lambda, th.grid[i], win, warmu
p, reg.function)$err

  res

}


sim.grid(data, l[1], 500, warmup = 500) # choose theta=0.1 or higher
```

```
##   [1] -1.5180141 -1.5261645 -1.8123940 -1.3370585 -1.8880686 -1.4991259
##   [7] -1.3158252 -1.1271947 -0.6673724 -0.2326335 -0.1783129 -0.1783129
##  [13] -0.1783129  0.6493373  0.6493373  0.6493373  0.6493373  0.6493373
##  [19]  0.6493373 -0.7520519 -0.7520519 -0.7520519 -0.7520519 -0.7520519
##  [25] -0.7520519 -0.7520519 -0.7520519 -0.7520519 -0.7520519 -0.7520519
##  [31] -0.7520519 -0.7520519 -0.7520519 -0.7520519 -0.7520519 -0.7520519
##  [37] -0.7520519 -0.7520519 -0.7520519 -0.7520519 -0.7520519 -0.7520519
##  [43] -0.7520519 -0.7520519 -0.7520519 -0.7520519 -0.7520519 -0.7520519
##  [49] -0.7520519 -0.7520519 -0.7520519 -0.7520519 -0.7520519 -0.7520519
##  [55] -0.7520519 -0.7520519 -0.7520519 -0.7520519 -0.7520519 -0.7520519
##  [61] -0.7520519 -0.7520519 -0.7520519 -0.7520519 -0.7520519 -0.7520519
##  [67] -0.7520519 -0.7520519 -0.7520519 -0.7520519 -0.7520519 -0.7520519
##  [73] -0.7520519 -0.7520519 -0.7520519 -0.7520519 -0.7520519 -0.7520519
##  [79] -0.7520519 -0.7520519 -0.7520519 -0.7520519 -0.7520519 -0.7520519
##  [85] -0.7520519 -0.7520519 -0.7520519 -0.7520519 -0.7520519 -0.7520519
##  [91] -0.7520519 -0.7520519 -0.7520519 -0.7520519 -0.7520519 -0.7520519
##  [97] -0.7520519 -0.7520519 -0.7520519 -0.7520519 -0.7520519

#sim.grid.valid(data, l[1], 500, warmup = 500)

### Q5 ###

q=1
data = pred.footsie.prepare(q)
data.dev = first.acf.squares.train(data, l[1])

train.pca = prcomp(data.dev$x.train.dev)
valid.pca = prcomp(data.dev$x.valid.dev)
test.pca = prcomp(data.dev$x.test.dev)

train.pca

## Standard deviations (1, .., p=11):
##  [1] 2.6908214 1.2174236 1.1484013 1.0035868 0.9026703 0.8387433 0.7369523
##  [8] 0.6679145 0.5858903 0.5395038 0.3566507
##
## Rotation (n x k) = (11 x 11):
##             PC1         PC2         PC3         PC4         PC5         PC6
##  [1,] 0.3753099 -0.17476786  0.13997796  0.15008437 -0.01785347 -0.05448676
##  [2,] 0.3256779  0.03633543  0.01682959 -0.02919087 -0.06131647 -0.12639467
##  [3,] 0.3030198  0.08335755  0.02945837 -0.24088625  0.02066430 -0.36103594
##  [4,] 0.3415676 -0.01180969 -0.01863526 -0.18072864 -0.19994393 -0.26233468
##  [5,] 0.3261432  0.26111517 -0.09601532 -0.14703285 -0.56573548  0.62608737
##  [6,] 0.3155894  0.40638695 -0.44383102  0.55125717  0.42858574  0.04809338
##  [7,] 0.2317582  0.10944207  0.29896198 -0.49315181  0.64849916  0.40253988
##  [8,] 0.3474847  0.05691754 -0.01009353  0.12248203 -0.12638538 -0.03192938
##  [9,] 0.1719237 -0.69700951 -0.63175173 -0.20165428  0.11946485  0.13842494
## [10,] 0.3209056 -0.06992218  0.20643828 -0.06810300  0.02217960 -0.35668350
## [11,] 0.1778349 -0.47094369  0.49142222  0.50688006  0.02205393  0.27955027
##               PC7         PC8          PC9         PC10         PC11
##  [1,]  0.07047082  0.05565961  0.029039808  0.003947237  0.879925530
##  [2,]  0.11784580  0.86163837  0.013915492 -0.273099978 -0.201636809
##  [3,] -0.34963642 -0.21490311 -0.682178777 -0.276920601 -0.032874494
##  [4,]  0.45342153 -0.06485999 -0.140491540  0.696092622 -0.165239844
##  [5,] -0.27983839 -0.02804439  0.004321006  0.058632145  0.004176171
##  [6,] -0.11273970 -0.01285871 -0.042990008  0.174977897 -0.055162017
##  [7,]  0.12663389 -0.02199568  0.041676797  0.026932809 -0.012720575
##  [8,]  0.50987234 -0.43403189  0.239321688 -0.553226218 -0.179529089
##  [9,] -0.06705435 -0.02109752  0.027986219 -0.045829814 -0.059891467
```

```
## [10,] -0.52559288 -0.11454608  0.621523449  0.119024134 -0.165329397
## [11,] -0.07442882 -0.02656911 -0.256717800  0.084048382 -0.300524754
```

```
valid.pca
```

```
## Standard deviations (1, .., p=11):
##  [1] 2.7962445 1.2052491 1.1282235 0.9965355 0.9153459 0.7941590 0.7590724
##  [8] 0.6642127 0.5763015 0.4974903 0.3125186
## 
## Rotation (n x k) = (11 x 11):
##               PC1         PC2         PC3          PC4          PC5         PC6
##  [1,] -0.34044857 -0.05228586 -0.14427618 -0.076705485 -0.020173162  0.11535540
##  [2,] -0.30543954 -0.06515488 -0.03064002  0.073544934  0.049410874  0.04004276
##  [3,] -0.34568329  0.18778533 -0.29931208  0.051354484  0.377043154 -0.28492551
##  [4,] -0.31095246 -0.02791065  0.07273851 -0.003689994  0.368606659  0.34909323
##  [5,] -0.29480470  0.03101616  0.23536133 -0.008166582 -0.000594984  0.50716224
##  [6,] -0.34758526  0.01741112  0.61043439 -0.270472764 -0.158727551 -0.59536820
##  [7,] -0.29466641  0.15504357 -0.07148369  0.775531658 -0.489275056 -0.07805909
##  [8,] -0.35709702 -0.01529527  0.29831273 -0.065096530 -0.023481391  0.25277004
##  [9,] -0.08466373 -0.95973128 -0.04739494  0.164195163  0.045376608 -0.11490678
## [10,] -0.28600590  0.08046122 -0.35961744 -0.007916493  0.339686743 -0.28467252
## [11,] -0.25213859 -0.06231334 -0.47923947 -0.529268050 -0.580117949  0.07833464
##               PC7         PC8          PC9        PC10        PC11
##  [1,]  0.23601861  0.09785632  0.004009709 -0.05763632 -0.87882258
##  [2,]  0.74446687 -0.27476894  0.149382843  0.41008295  0.26806128
##  [3,] -0.49013425 -0.24450372  0.269219661  0.39158386 -0.06196649
##  [4,] -0.00377407 -0.08632469  0.438890346 -0.63766338  0.18105965
##  [5,] -0.19249788  0.60816621  0.023641518  0.41850954  0.12969932
##  [6,]  0.01892070  0.14421496  0.157267230 -0.08630531  0.01002040
##  [7,] -0.07139005  0.03204175  0.038991533 -0.16549129  0.04538400
##  [8,] -0.20518523 -0.52945301 -0.625751444 -0.01979260  0.01405591
##  [9,] -0.15159723  0.04127665 -0.001937287  0.04227896  0.02832385
## [10,]  0.17100324  0.41519263 -0.532330100 -0.22423757  0.22500859
## [11,] -0.11394601 -0.03990258  0.103964062 -0.07816376  0.22041071
```

```
test.pca
```

```
## Standard deviations (1, .., p=11):
##  [1] 2.7200725 1.6199612 1.2310197 1.1402806 1.1033865 1.0171014 0.8686622
##  [8] 0.8105881 0.7596497 0.6969744 0.3978733
## 
## Rotation (n x k) = (11 x 11):
##              PC1           PC2         PC3          PC4         PC5         PC6
##  [1,] -0.3714561  0.1032561919 -0.14224133 -0.179968988  0.07963333  0.02601431
##  [2,] -0.2554644 -0.1359056854 -0.07422914  0.006179086  0.49451182  0.30340451
##  [3,] -0.2560279 -0.0720879777  0.08151337 -0.341509637 -0.02606410  0.12371099
##  [4,] -0.2905068  0.0065481151  0.17415117 -0.100367387  0.25882992  0.02733597
##  [5,] -0.3789372  0.2006241301  0.21036179  0.296489695 -0.13133992 -0.14656722
##  [6,] -0.3964002  0.1765219247  0.40259795  0.384756781 -0.34751664 -0.09243374
##  [7,] -0.2191163  0.1678017056 -0.54514818  0.478347761  0.03176049  0.49659689
##  [8,] -0.2799943  0.0715094869  0.17573133 -0.088025353  0.56434253 -0.28440661
##  [9,] -0.1479953 -0.0008106139  0.30569709 -0.428713095 -0.31199318  0.62406611
## [10,] -0.4299631 -0.6230078085 -0.39381962 -0.123180322 -0.32230544 -0.30247406
## [11,] -0.1136122  0.6876102181 -0.39285435 -0.411574130 -0.15071355 -0.22966648
##               PC7         PC8         PC9         PC10        PC11
##  [1,] -0.04613715  0.08232039 -0.10155663 -0.024178450 -0.87858707
##  [2,] -0.61123011  0.09510561  0.15716211  0.371679283  0.16920538
##  [3,]  0.52978004  0.54409259  0.34231515  0.279696776  0.13372520
##  [4,] -0.04298906  0.37746563 -0.48599568 -0.603393433  0.25063256
##  [5,]  0.13215786 -0.07131993 -0.54410846  0.561517691  0.10657334
```

```
##  [6,] -0.25542569  0.08874959  0.49189557 -0.238435570 -0.01845731
##  [7,]  0.31817492 -0.08686572  0.05158952 -0.172918452  0.09416535
##  [8,]  0.32752279 -0.54642209  0.24080454 -0.101704005  0.06565985
##  [9,] -0.02825568 -0.44509211 -0.09321735 -0.049976471  0.06293131
## [10,] -0.07454577 -0.16789616 -0.02230439 -0.063547847  0.15189599
## [11,] -0.21020053 -0.01671359  0.06135406  0.004677607  0.25854512
```

```r
# scree plots to check
par(mfrow=c(1,3))
plot(train.pca$sdev^2)
plot(valid.pca$sdev^2)
plot(test.pca$sdev^2)
```