Homework #3

Larry Lei Hua

Feb 24, 2019

In total there are 30 pts:

(1) 5 pts for homework policy and presentation of the homework.
- If the files are not complete or not zipped into one file (should be one zipped file with one pdf and one Rmd), deduct 2 pts
- If the overall presentation is messy and not organized, deduct 3 pts
- If the data is also submitted, deduct 1 pt

(2) 6 pts for Question 1
- The critical places to check: (a) The R functions **fracDiff** is correctly implemented; the **weights_fracDiff** can be inside of the **fracDiff** function or a separate function; (b) the two correlation coefficients calculated should be very close to the ones provided here; (c) the reason is right.
- If the function **fracDiff** has big issues making all the other calculation wrong, deduct at least 4 pts for Question 1.

(3) 6 pts for Question 2
- The critical places to check: (a) d for the closed prices and for the volumes should be **0.73** and **0.73**, respectively, or some numbers **very** close to them. (b) For the closed prices and for the volumes **respectively**, KPSS test and at least one ADF test should be conducted: at 5% level, we should fail to reject KPSS test but reject ADF test.

(4) 5 pts for Question 3
- 3 pts for CUSUM filtering and feature matrix construction; the fractionally differentiated closed prices should be used.
- 2 pts for meta labeling; the raw closed prices should be used.

(5) 8 pts for Question 4
- 2 pts for correctly applying bagging
- 4 pts for evaluating model performance based on AUC; if the testSet is not used for calculating AUC, then deduct at least 2 pts
- 2 pts for (a) the number of “1” in the test set is at least 10, and the AUC based on the test set is at least 0.7
- There are many answers for this question, and the grid search is not necessary as long as the above criterion are satisfied

---

(Please read the Homework Policy before you start)

Description: In this homework, you will practice implementing algorithms using R, generating fractional differentiated features while considering the trade off between memory and stationarity of time series, and applying bagged classification trees.

Dataset: Please download the unit bar data of bitcoin futures XBTUSD from Compass 2g, and the data has been preprocessed from tick data of the same ticker traded during the first 8 months of 2018. Use the first 2/3 as training set and the remaining 1/3 as test set.

Practices:

1. Implement the function **fracDiff** for fractionally differentiated features. Apply the function on the closed prices of the provided unit bars (training and test sets together), choose $d = 0.5$ and $\tau = 0.001$, and then print out the two correlation coefficients: one for the closed prices of the derived series and the original series, and the other for the first order difference of the closed prices and the original series. Which is larger? Why?

```r
rm(list = setdiff(ls(), lsf.str()))
library(CADFtest)
library(tseries)
library(fUnitRoots)
library(randomForest)
library(ROCR)
# source("~/Dropbox/Teaching/STAT430/R/functions.R")

#' @param d the order for fractionally differentiated features
#' @param nWei number of weights for output
#' @param tau threshold where weights are cut off; default is NULL, if not NULL then use tau and nWei is not used
#'
#' @examples
#' weights_fracDiff(0.5,tau=1e-3)
#'
#' @export
weights_fracDiff <- function(d=0.3, nWei=10, tau=NULL)
{
  wVec <- w0 <- 1
  if(is.null(tau))
  {
    for(k in 1:(nWei-1))
    {
      w1 <- (-1)*w0*(d-k+1)/k
      wVec <- c(wVec, w1)
      w0 <- w1
    }
  }else
  {
    k <- 1
    while(abs(w0) >= tau)
    {
      w1 <- (-1)*w0*(d-k+1)/k
      wVec <- c(wVec, w1)
      w0 <- w1
      k <- k+1
    }
    wVec <- wVec[-length(wVec)] # remove the last one which is already < tau
  }
  return(wVec)
}

#' @param x a vector of time series to be fractionally differentiated
#' @param d the order for fractionally differentiated features
#' @param nWei number of weights for output
#' @param tau threshold where weights are cut off; default is NULL, if not NULL then use tau and nWei is not used
#'
#' @example
#' tau <- 1e-3
#' set.seed(1)
#' x <- runif(100, 1, 3)
#' x_fracDiff <- fracDiff(x)
#'
#' @export
fracDiff <- function(x, d=0.3, nWei=10, tau=NULL)
{
  weig <- weights_fracDiff(d=d, nWei=nWei, tau=tau)
  nWei <- length(weig) # the first one in x that can use all the weights
  nx <- length(x)
  rst <- rep(NA, nx)
  rst[nWei:nx] <- sapply(nWei:nx, function(i){ sum(weig*x[i:(i-nWei+1)]) })
  return(rst)
}

dat <- read.csv("~/Dropbox/Teaching/STAT430/homework/hw03/unit_bar_XBTUSD_all.csv", header = T)
```

```r
dat$V <- as.numeric(dat$V)
dat$C <- as.numeric(dat$C)

fracD_C <- fracDiff(dat$C, d=0.5, tau=0.001)
diff_C <- c(NA, diff(dat$C))

inx1 <- !is.na(fracD_C)
inx2 <- !is.na(diff_C)

cor1 <- cor(fracD_C[inx1], dat$C[inx1], method = "pearson")
cor2 <- cor(diff_C[inx2], dat$C[inx2], method = "pearson")
cat("correlation between the fractionally differentiated closed prices and the original: ", cor1, "\n")
```

```
## correlation between the fractionally differentiated closed prices and the original:  0.809092
```

```r
cat("correlation between the first order differentiated closed prices and the original: ", cor2, "\n")
```

```
## correlation between the first order differentiated closed prices and the original:  0.02055376
```

The first one is larger because more memory is kept using fractionally differentiated series

2. Based on the training set, let $\tau = 0.0001$, and choose appropriate values of $1 > d > 0$ for the closed prices and the volumes respectively, to pass some unit root tests and the stationarity test (KPSS), while keeping as much memory as possible.

```r
S <- nrow(dat) # sample size
trainDat <- dat[1:floor(S/3*2),]
testDat <- dat[(floor(S/3*2)+1):S,]
S_train <- nrow(trainDat)
S_test <- nrow(testDat)

################################################
# use training data to find d for closed price #
################################################
d_C <- 0.73 # 0.73 is good after several trial and fail
C_fracD <- fracDiff(trainDat$C, d=d_C, tau=1e-4)
idx <- !is.na(C_fracD) # choose non NA

## Stationarity test
# KPSS test
tseries::kpss.test(C_fracD[idx], null="Trend")
```

```
##
##  KPSS Test for Trend Stationarity
##
## data:  C_fracD[idx]
## KPSS Trend = 0.14267, Truncation lag parameter = 9, p-value =
## 0.05617
```

```
## unit root tests
# Augmented Dickey-Fuller Test
tseries::adf.test(C_fracD[idx]) # only linear trend model can be specified
```

```
## Warning in tseries::adf.test(C_fracD[idx]): p-value smaller than printed p-
## value
```

```
##
##  Augmented Dickey-Fuller Test
##
## data:  C_fracD[idx]
## Dickey-Fuller = -11.868, Lag order = 14, p-value = 0.01
## alternative hypothesis: stationary
```

```
# Phillips-Perron Unit Root Test
tseries::pp.test(C_fracD[idx])
```

```
## Warning in tseries::pp.test(C_fracD[idx]): p-value smaller than printed p-
## value
```

```
##
##  Phillips-Perron Unit Root Test
##
## data:  C_fracD[idx]
## Dickey-Fuller Z(alpha) = -3057.6, Truncation lag parameter = 9,
## p-value = 0.01
## alternative hypothesis: stationary
```

```
stats::PP.test(C_fracD[idx])
```

```
##
##  Phillips-Perron Unit Root Test
##
## data:  C_fracD[idx]
## Dickey-Fuller = -46.839, Truncation lag parameter = 9, p-value =
## 0.01
```

```r
# Elliot, Rothenberg and Stock's efficient unit root test
summary(urca::ur.ers(C_fracD[idx], model = "trend"))
```

```
##
## ###############################################
## # Elliot, Rothenberg and Stock Unit Root Test #
## ###############################################
##
## Test of type DF-GLS
## detrending of series with intercept and trend
##
##
## Call:
## lm(formula = dfgls.form, data = data.dfgls)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -753.41  -63.48   -6.08   46.37  818.63
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## yd.lag       -0.33110    0.02196 -15.074  < 2e-16 ***
## yd.diff.lag1 -0.41310    0.02389 -17.292  < 2e-16 ***
## yd.diff.lag2 -0.23248    0.02342  -9.927  < 2e-16 ***
## yd.diff.lag3 -0.15874    0.02162  -7.344 2.62e-13 ***
## yd.diff.lag4 -0.07141    0.01748  -4.086 4.49e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 131.3 on 3234 degrees of freedom
## Multiple R-squared:  0.3609, Adjusted R-squared:  0.3599
## F-statistic: 365.3 on 5 and 3234 DF,  p-value: < 2.2e-16
##
##
## Value of test-statistic is: -15.0741
##
## Critical values of DF-GLS are:
##                  1pct  5pct 10pct
## critical values -3.48 -2.89 -2.57
```

```
################################################
#    use training data to find d for volumes   #
################################################
d_V <- 0.73 # 0.73 is good after several trial and fail
V_fracD <- fracDiff(trainDat$V, d=d_V, tau=1e-4)
idx <- !is.na(V_fracD) # choose non NA

## Stationarity test
# KPSS test
tseries::kpss.test(V_fracD[idx], null="Trend")
```

```
##
##  KPSS Test for Trend Stationarity
##
## data:  V_fracD[idx]
## KPSS Trend = 0.14273, Truncation lag parameter = 9, p-value =
## 0.05606
```

```
## unit root tests
# Augmented Dickey-Fuller Test
tseries::adf.test(V_fracD[idx]) # only linear trend model can be specified
```

```
## Warning in tseries::adf.test(V_fracD[idx]): p-value smaller than printed p-
## value
```

```
##
##  Augmented Dickey-Fuller Test
##
## data:  V_fracD[idx]
## Dickey-Fuller = -10.908, Lag order = 14, p-value = 0.01
## alternative hypothesis: stationary
```

```
# Phillips-Perron Unit Root Test
tseries::pp.test(V_fracD[idx])
```

```
## Warning in tseries::pp.test(V_fracD[idx]): p-value smaller than printed p-
## value
```

```
##
##  Phillips-Perron Unit Root Test
```

```
##
## data:  V_fracD[idx]
## Dickey-Fuller Z(alpha) = -1948.7, Truncation lag parameter = 9,
## p-value = 0.01
## alternative hypothesis: stationary
```

```
stats::PP.test(V_fracD[idx])
```

```
##
##  Phillips-Perron Unit Root Test
##
## data:  V_fracD[idx]
## Dickey-Fuller = -35.507, Truncation lag parameter = 9, p-value =
## 0.01
```

```
# Elliot, Rothenberg and Stock's efficient unit root test
summary(urca::ur.ers(V_fracD[idx], model = "trend"))
```

```
##
## ###############################################
## # Elliot, Rothenberg and Stock Unit Root Test #
## ###############################################
##
## Test of type DF-GLS
## detrending of series with intercept and trend
##
##
## Call:
## lm(formula = dfgls.form, data = data.dfgls)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -12927387   -501183     11466    588297   9156949
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## yd.lag       -0.42646    0.02239 -19.049  < 2e-16 ***
## yd.diff.lag1 -0.11888    0.02329  -5.104 3.51e-07 ***
## yd.diff.lag2 -0.11305    0.02207  -5.123 3.18e-07 ***
## yd.diff.lag3 -0.01403    0.02002  -0.701    0.484
## yd.diff.lag4 -0.01582    0.01758  -0.900    0.368
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1202000 on 3234 degrees of freedom
## Multiple R-squared:  0.2665, Adjusted R-squared:  0.2653
## F-statistic:   235 on 5 and 3234 DF,  p-value: < 2.2e-16
##
##
## Value of test-statistic is: -19.0491
##
## Critical values of DF-GLS are:
##                  1pct  5pct 10pct
## critical values -3.48 -2.89 -2.57
```

3. Apply **fracDiff** with the selected d's for the closed prices and the volume, respectively, based on all the data sets. Apply a symmetric CUSUM filter **on the raw closed prices series** and set up the threshold as h. For those sampled features bars, assign labels (0 for lower, and 1 for upper) by applying the triple barrier labeling method **on the raw closed prices series**, where ptSl = [1,1], target returns is $trgt$, and t1 can be set up large enough to generate enough labels (for example, vertical barriers can be 200 bars away). You can choose to use **fmlr::label_meta()** for labeling.

4. Apply bagged classification trees on the obtained features bars and labels from the training set, including at least the fractionally differentiated closed prices and the fractionally differentiated volumes as predictors, while treating h and $trgt$ as the two tuning parameters. Choose appropriate values of h and $trgt$ so that the following goals are achieved: (1) There are at least 10 labels of 1 for the features bars obtained from the test set; (2) the AUC for the **test set** predicted by the bagged trees is at least 0.6. Finaly, use **fmlr::acc_lucky()** to check whether your candidate model with the tuned parameters can overperform various guesses; using the cutoff of 0.5 for calculating the accuracy based on the candidate model.

```r
###############################################################################################
# construct feature bars using all the data after deciding on the value of d based on training data
C_fracD_all <- fracDiff(dat$C, d=d_C, tau=1e-4)
V_fracD_all <- fracDiff(dat$V, d=d_V, tau=1e-4)

# because there could be different d's for closed prices and volumes,
# we should only choose the observations that both fractionaly differentiated close prices and volumes are not NA
idx_fracD_all <- !(is.na(C_fracD_all) | is.na(V_fracD_all))
n_NA <- sum(idx_fracD_all == F) # number of NA at the begining
```

The codes inside the following loops are for Question 3. For Question 4, we can use the following grid search to find those parameters of h and $trgt$ to satisfy the performance requirement. The combinations that satisfy the requirements are highlighted at the end.

```r
hVec <- seq(10,300,length=30)
trgtVec <- seq(0.0001, 0.05, length=30)

set.seed(1)
for(ih in 1:30)
{
  for(jtrgt in 1:20)
  {
    #####################################################
    # apply cusum filter on fracDiff closed prices only #
    #####################################################

    # adding n_NA to make i_CUSUM correspond to the original data
    i_CUSUM <- fmlr::istar_CUSUM(C_fracD_all[idx_fracD_all], h=hVec[ih]) + n_NA
    n_Event <- length(i_CUSUM)

    ##############################################
    ## meta labeling based on the raw closed price
    ##############################################
    events <- data.frame(t0=i_CUSUM+1,
                         t1 = i_CUSUM+200,
                         trgt = trgtVec[jtrgt],
                         side=rep(0,n_Event))
    ptSl <- c(1,1)

    # use raw data (not the fractionally differentiated data) to create labels
    out0 <- fmlr::label_meta(dat$C, events, ptSl, n_ex=n_NA)
    # plot(out0$ret)

    ## construct X_train using various features obtained from sampled feature bars
    x <- C_fracD_all
    v <- V_fracD_all

    fMat0 <- t(sapply(1:nrow(out0),
                      function(i)
                      {
                        i_range <- out0$t0Fea[i]:out0$t1Fea[i]
                        winTmp <- x[i_range]
```

```
                            C <- tail(winTmp,1)
                            V <- sum(v[i_range])
                            return(c(C,V))
                          }
))

fMat0 <- data.frame(fMat0)
names(fMat0) <- c("C", "V")
dim(fMat0)

out0$label[out0$label==-1] <- 0

allSet <- data.frame(Y=as.factor(out0$label),
                     C=as.numeric(fMat0$C),
                     V=as.numeric(fMat0$V),
                     t1Fea=out0$t1Fea)

trainSet <- subset(allSet, t1Fea<=S_train)
testSet <- subset(allSet, t1Fea>S_train)
c(nrow(allSet), nrow(trainSet), nrow(testSet))

tb_test <- table(testSet$Y)
nTest1 <- tb_test[2] # number of "1" in the testSet
nTest0 <- tb_test[1] # number of "0" in the testSet

# bagging
bag <- randomForest(Y ~ C + V, data = trainSet, mtry = 2, importance = FALSE, ntrees = 200)

# evaluating auc based on the test set
prob_test <- predict(bag, newdata=testSet, type="prob")
pred <- prediction(prob_test[,2],testSet$Y)
auc <- performance(pred, measure = "auc")@y.values[[1]]

# evaluating acc based on the test set and the optimal cutpoint obtained from the training set
acc_perf <- performance(pred, measure = "acc")
acc_vec <- acc_perf@y.values[[1]]
acc <- acc_vec[max(which(acc_perf@x.values[[1]] >= 0.5))]
```

```r
      # monitor progress
      if(auc>=0.6 & nTest1>=10)
      {
        lucky_score <- fmlr::acc_lucky(train_class = table(trainSet$Y),
                                       test_class = tb_test,
                                       my_acc = acc)
        lucky_score <- round(unlist(lucky_score),3)
        cat(ih, jtrgt, " | ", hVec[ih], round(trgtVec[jtrgt],4), " | ", table(trainSet$Y), " | ",
            nrow(allSet), nrow(trainSet), nrow(testSet), " | ", round(auc,3), " | ", nTest1, "| lucky_score:",  lucky_score, "\n")
        cat("----------------------------------------------------------------", "\n")
      }
    }
  }
```

```
## 14 4  |  140 0.0053  |  517 568  |  1255 1085 170  |  0.602  |  76 | lucky_score: 0.553 0.095 0.079 0.499 0.498 0.447
## ----------------------------------------------------------------
## 15 4  |  150 0.0053  |  477 535  |  1157 1012 145  |  0.626  |  70 | lucky_score: 0.614 0.007 0.002 0.501 0.498 0.483
## ----------------------------------------------------------------
## 18 1  |  180 1e-04  |  397 445  |  951 842 109  |  0.608  |  52 | lucky_score: 0.514 0.441 0.4 0.502 0.499 0.477
## ----------------------------------------------------------------
## 22 1  |  220 1e-04  |  312 349  |  724 661 63  |  0.615  |  32 | lucky_score: 0.587 0.11 0.099 0.5 0.499 0.508
## ----------------------------------------------------------------
## 22 3  |  220 0.0035  |  318 343  |  724 661 63  |  0.676  |  27 | lucky_score: 0.603 0.085 0.062 0.5 0.498 0.429
## ----------------------------------------------------------------
## 25 15  |  250 0.0242  |  289 269  |  600 558 42  |  0.609  |  19 | lucky_score: 0.524 0.447 0.441 0.503 0.501 0.548
## ----------------------------------------------------------------
## 25 16  |  250 0.0259  |  291 267  |  600 558 42  |  0.606  |  20 | lucky_score: 0.595 0.135 0.151 0.503 0.5 0.524
## ----------------------------------------------------------------
## 25 18  |  250 0.0294  |  292 266  |  600 558 42  |  0.604  |  18 | lucky_score: 0.524 0.408 0.458 0.497 0.502 0.571
## ----------------------------------------------------------------
## 27 8  |  270 0.0121  |  250 253  |  536 503 33  |  0.602  |  20 | lucky_score: 0.515 0.492 0.493 0.499 0.499 0.606
## ----------------------------------------------------------------
## 28 1  |  280 1e-04  |  222 249  |  501 471 30  |  0.603  |  16 | lucky_score: 0.533 0.432 0.45 0.5 0.502 0.533
## ----------------------------------------------------------------
## 28 6  |  280 0.0087  |  231 240  |  501 471 30  |  0.625  |  16 | lucky_score: 0.5 0.575 0.572 0.504 0.499 0.533
## ----------------------------------------------------------------
## 28 7  |  280 0.0104  |  226 245  |  501 471 30  |  0.645  |  16 | lucky_score: 0.533 0.422 0.433 0.5 0.5 0.533
## ----------------------------------------------------------------
## 28 9  |  280 0.0139  |  235 236  |  501 471 30  |  0.65  |  18 | lucky_score: 0.633 0.089 0.087 0.498 0.501 0.6
## ----------------------------------------------------------------
## 29 1  |  290 1e-04  |  211 233  |  470 444 26  |  0.613  |  14 | lucky_score: 0.577 0.302 0.284 0.505 0.499 0.538
## ----------------------------------------------------------------
## 29 2  |  290 0.0018  |  216 228  |  470 444 26  |  0.692  |  13 | lucky_score: 0.5 0.578 0.572 0.502 0.499 0.5
## ----------------------------------------------------------------
## 29 3  |  290 0.0035  |  218 226  |  470 444 26  |  0.627  |  13 | lucky_score: 0.654 0.087 0.088 0.499 0.496 0.5
## ----------------------------------------------------------------
## 29 4  |  290 0.0053  |  213 231  |  470 444 26  |  0.622  |  12 | lucky_score: 0.615 0.161 0.159 0.499 0.497 0.462
## ----------------------------------------------------------------
## 29 5  |  290 0.007  |  217 227  |  470 444 26  |  0.738  |  14 | lucky_score: 0.692 0.047 0.044 0.499 0.496 0.538
## ----------------------------------------------------------------
## 29 6  |  290 0.0087  |  221 223  |  470 444 26  |  0.61  |  14 | lucky_score: 0.615 0.154 0.176 0.502 0.503 0.538
## ----------------------------------------------------------------
## 29 7  |  290 0.0104  |  215 229  |  470 444 26  |  0.72  |  14 | lucky_score: 0.577 0.293 0.254 0.5 0.497 0.538
## ----------------------------------------------------------------
## 29 8  |  290 0.0121  |  222 222  |  470 444 26  |  0.782  |  15 | lucky_score: 0.769 0.006 0.004 0.499 0.5 0.423
## ----------------------------------------------------------------
## 29 9  |  290 0.0139  |  227 217  |  470 444 26  |  0.719  |  16 | lucky_score: 0.577 0.271 0.28 0.5 0.498 0.385
## ----------------------------------------------------------------
## 29 10  |  290 0.0156  |  228 216  |  470 444 26  |  0.647  |  17 | lucky_score: 0.462 0.725 0.715 0.5 0.501 0.346
## ----------------------------------------------------------------
## 29 11  |  290 0.0173  |  224 220  |  470 444 26  |  0.638  |  16 | lucky_score: 0.538 0.391 0.432 0.494 0.5 0.385
## ----------------------------------------------------------------
## 29 12  |  290 0.019  |  226 218  |  470 444 26  |  0.654  |  17 | lucky_score: 0.538 0.437 0.405 0.504 0.5 0.346
## ----------------------------------------------------------------
## 29 13  |  290 0.0207  |  227 217  |  470 444 26  |  0.699  |  17 | lucky_score: 0.5 0.605 0.531 0.505 0.493 0.346
## ----------------------------------------------------------------
## 29 14  |  290 0.0225  |  234 210  |  470 444 26  |  0.628  |  16 | lucky_score: 0.5 0.566 0.553 0.498 0.494 0.385
## ----------------------------------------------------------------
## 30 2  |  300 0.0018  |  210 209  |  444 419 25  |  0.676  |  13 | lucky_score: 0.72 0.014 0.031 0.498 0.5 0.48
## ----------------------------------------------------------------
## 30 3  |  300 0.0035  |  210 209  |  444 419 25  |  0.604  |  14 | lucky_score: 0.56 0.369 0.372 0.504 0.503 0.44
## ----------------------------------------------------------------
## 30 5  |  300 0.007  |  209 210  |  444 419 25  |  0.688  |  14 | lucky_score: 0.56 0.346 0.359 0.5 0.502 0.56
## ----------------------------------------------------------------
## 30 6  |  300 0.0087  |  213 206  |  444 419 25  |  0.61  |  14 | lucky_score: 0.6 0.218 0.202 0.501 0.494 0.44
## ----------------------------------------------------------------
## 30 7  |  300 0.0104  |  207 212  |  444 419 25  |  0.669  |  14 | lucky_score: 0.64 0.108 0.111 0.494 0.499 0.56
## ----------------------------------------------------------------
## 30 8  |  300 0.0121  |  211 208  |  444 419 25  |  0.769  |  13 | lucky_score: 0.76 0.007 0.006 0.501 0.503 0.48
## ----------------------------------------------------------------
## 30 9  |  300 0.0139  |  214 205  |  444 419 25  |  0.708  |  14 | lucky_score: 0.6 0.228 0.22 0.505 0.497 0.44
## ----------------------------------------------------------------
## 30 10  |  300 0.0156  |  214 205  |  444 419 25  |  0.647  |  15 | lucky_score: 0.48 0.668 0.653 0.503 0.498 0.4
## ----------------------------------------------------------------
## 30 12  |  300 0.019  |  210 209  |  444 419 25  |  0.613  |  15 | lucky_score: 0.52 0.512 0.495 0.504 0.498 0.4
## ----------------------------------------------------------------
## 30 13  |  300 0.0207  |  211 208  |  444 419 25  |  0.7  |  15 | lucky_score: 0.6 0.226 0.198 0.499 0.501 0.4
## ----------------------------------------------------------------
## 30 14  |  300 0.0225  |  219 200  |  444 419 25  |  0.713  |  15 | lucky_score: 0.6 0.217 0.19 0.5 0.489 0.4
## ----------------------------------------------------------------
## 30 15  |  300 0.0242  |  222 197  |  444 419 25  |  0.688  |  14 | lucky_score: 0.56 0.357 0.309 0.501 0.497 0.44
## ----------------------------------------------------------------
## 30 16  |  300 0.0259  |  222 197  |  444 419 25  |  0.682  |  14 | lucky_score: 0.6 0.205 0.173 0.502 0.492 0.44
## ----------------------------------------------------------------
## 30 17  |  300 0.0276  |  222 197  |  444 419 25  |  0.699  |  13 | lucky_score: 0.72 0.023 0.018 0.5 0.499 0.48
## ----------------------------------------------------------------
## 30 18  |  300 0.0294  |  222 197  |  444 419 25  |  0.647  |  12 | lucky_score: 0.64 0.116 0.115 0.498 0.503 0.52
## ----------------------------------------------------------------
## 30 19  |  300 0.0311  |  223 196  |  444 419 25  |  0.622  |  12 | lucky_score: 0.68 0.056 0.051 0.504 0.502 0.52
## ----------------------------------------------------------------
```