Vectorization of Dataframe Write for Filling Missing Values

Accessing large dataframes can be computationally expensive. What follows are two methods for vectorizing the process of writing to a dataframe when filling in missing values. We often want to fill in missing values when pre-processing a dataset for model construction and analysis.

The first method set out below, what I will call the "**named vector**" approach, is about 4 to 8 times faster than sequentially walking through the "cells" indexed by (row, column) pairs. The second method, what I will call the "**transposed columns**" approach, can be more than 100 times faster than the "**sequential walk-through**" approach. The comparative measures given here (4-8X, 100X) are very rough estimates and depend on the size of the dataframe, the number of columns having missing values, the number of missing values, and possibly how we are imputing values for each of the different column variables. In the tests that follow, I use the same imputation approach for all variables, replacing a missing value for a variable with the median of its existing values. In the tests that follow I *only vary the number of missing values* in the dataframe I am working with.

Given that the Transposed Columns approach is so much faster, why even bother with the Named Vector approach? I do so because the latter oftentimes has value in other contexts, and the former is merely a refinement of the first method. Also, comparing the two methods has pedagogic value exactly because they are so closely related. Until we run some tests, we do not know how the two methods will compare, nor do we know the degree to which each is an improvement over sequentially walking through a dataframe.

No doubt, with a little more investment of time, one can also improve upon the Transposed Columns approach. It may be hard to beat, though, for its simplicity relative to its effectiveness; one does not have to be an expert in R to make sense of how it works.

* * * * *

Preliminaries

In [1]:
```
# I make use of the dataset that I work with in my CA_housing project.


modDat <- read.csv("/home/greg/Documents/stat/Geron_ML/datasets/housing/hhRAW_with_HHdens.c
                   header = T,
                   colClasses= c(rep("numeric", 9), rep("character", 2), rep("numeric", 4))
print(dim(modDat))
# [1] 20423    15

print(colnames(modDat))
```

```
[1] 20423    15
 [1] "longitude"          "latitude"           "housing_median_age"
 [4] "total_rooms"        "total_bedrooms"     "population"
 [7] "households"         "median_income"      "median_house_value"
[10] "ocean_proximity"    "HHdensity"          "HHdens_ln"
[13] "rooms_per_hh"       "bdrms_per_room"     "pop_per_hh"
```

In [2]:
```
# Here I am not interested in working with the categorical data.

df <- modDat[, c(1:9, 12:15)]
colnames(df)
```

'longitude' 'latitude' 'housing_median_age' 'total_rooms' 'total_bedrooms' 'population' 'households' 'median_income' 'median_house_value' 'HHdens_ln' 'rooms_per_hh' 'bdrms_per_room' 'pop_per_hh'

In [3]:
```
# There are 13 columns in df.
```

```
dim(df)
```

20423 13

Sequential Walk-Through Approach

If our dataset is small, we can simply walk through the cells of the dataframe and replace missing values as we find them. With larger datasets, this method is very inefficient!

In [ ]:
```
# We must first create some missing values in df.

# Insert NA in 10,000 randomly selected cells of the dataframe.
# df has 265,499 cells.

set.seed(4331)
n <- 10000
yy <- sample(1:13, n, replace= TRUE)
xx <- sample(1:nrow(df), n, replace= FALSE)

for(i in 1:n) {
  df[xx[i], yy[i]] <- NA
}
```

In [6]:
```
# Compute the median of each variable.  Note that this computation has
# to be done every time we create a new set of missing values in df.

imputed_vals <- rep(NA, dim(df)[2])
for(i in 1:length(imputed_vals)) {
  imputed_vals[i] <- median(df[, i], na.rm= TRUE)
}
named_imputed_vals <- imputed_vals
names(named_imputed_vals) <- 1:13
print(round(named_imputed_vals, 2))
```

```
        1         2         3         4         5         6         7         8
  -118.49     34.26     29.00   2127.00    435.00   1166.00    409.00      3.54
        9        10        11        12        13
179700.00      7.72      5.23      0.20      2.82
```

In [7]:
```
# Measure the time it takes to fill in these 10K missing values.

start <- Sys.time()
for(i in 1:dim(df)[1]) {
  for(j in 1:dim(df)[2]) {
    if(is.na(df[i, j])) {
      df[i, j] <- imputed_vals[j]
    }
  }
}
stop <- Sys.time()
round(stop - start, 4)
# Time difference of 1.963 secs
```

```
Time difference of 1.963 secs
```

Sequential Walk-Through single test results for 10K NAs: 1.963 seconds

In [ ]:
```
### COMMENT:

# For this approach, I am not going to verify that the missing
# values have been filled in as expected.
```

Sequential Walk-Through Results for 100 trials: *smaller* sets of NAs

In [4]:
```
# Create 100 trial sizes.  Each element of smp is for a single
# trial and is the number of missing values that will be
# inserted into our dataframe.

set.seed(99871)
smp <- sample(2000:20000, 100, replace= FALSE)
```

In [5]:
```
# This function constructs a dataframe with missing values
# and then sequentially fills in the missing values.

method_direct <- function(n_vals, seed) {

  df <- modDat[, c(1:9, 12:15)]
  set.seed(seed)
  yy <- sample(1:13, n_vals, replace= TRUE)
  xx <- sample(1:nrow(df), n_vals, replace= TRUE)

  for(i in 1:n_vals) {
    df[xx[i], yy[i]] <- NA
  }

  imputed_vals <- rep(NA, dim(df)[2])
  for(i in 1:length(imputed_vals)) {
    imputed_vals[i] <- median(df[, i], na.rm= TRUE)
  }

  # This is the part for which we need a time.
  start <- Sys.time()
  for(i in 1:dim(df)[1]) {
    for(j in 1:dim(df)[2]) {
      if(is.na(df[i, j])) {
        df[i, j] <- imputed_vals[j]
      }
    }
  }
  stop <- Sys.time()
  delta <- as.numeric(stop - start)

  return(delta)
}
```

In [6]:
```
# Run the test.

result <- 0
for(i in 1:length(smp)) {
  seed <- 8871 * i
  swt_time <- method_direct(smp[i], seed)
  result <- result + swt_time
}
(swt_avg_time_tst01 <- round(result/length(smp), 4))
# 2.2681 seconds
```

2.2681

Sequential Walk-Through 100 trials, test01 average time: 2.2681 seconds

Sequential Walk-Through Results for 100 trials: *larger* sets of NAs

In [7]:
```
# Get trial sizes.  Keep in mind that df has 265,499 cells.  If
# 50K cells have missing values, that equates to almost 19% of
# the dataframe cells having missing values.

set.seed(99871)
smp <- sample(20000:50000, 100, replace= FALSE)
```

In [8]:
```
# Run the test.

result <- 0
for(i in 1:length(smp)) {
  seed <- 8871 * i
  swt_time <- method_direct(smp[i], seed)
  result <- result + swt_time
}
(swt_avg_time_tst02 <- round(result/length(smp), 4))
# 3.0131
```

3.0131

Sequential Walk-Through 100 trials, test02 average time: 3.0131 seconds

Named Vector Approach

Here is one way to vectorize the above process.

In [13]:
```
df <- modDat[, c(1:9, 12:15)]
```

In [14]:
```
# We must first create some missing values in df.

# As above, initially insert NA in 10,000 randomly
# selected cells of the dataframe.  The same
# seed is used from the above, corresponding test.

set.seed(4331)
n <- 10000
yy <- sample(1:13, n, replace= TRUE)
xx <- sample(1:nrow(df), n, replace= FALSE)

for(i in 1:n) {
  df[xx[i], yy[i]] <- NA
}
```

In [15]:
```
# Compute the median of each variable.

imputed_vals <- rep(NA, dim(df)[2])
for(i in 1:length(imputed_vals)) {
  imputed_vals[i] <- median(df[, i], na.rm= TRUE)
}
named_imputed_vals <- imputed_vals
names(named_imputed_vals) <- 1:13
print(round(named_imputed_vals, 2))
```

```
        1         2         3         4         5         6         7         8
  -118.50     34.26     29.00   2125.00    436.00   1165.50    409.00      3.53
        9        10        11        12        13
179800.00      7.72      5.23      0.20      2.82
```

In [ ]:
```
require(stringr)
```

In [16]:
```
# Two functions needed for this approach:
```

```
get_column <- function(cell_name) {
  index <- as.numeric(str_locate(cell_name, "_")[1]) + 1
  return(as.numeric(substr(cell_name, index, 100)))
}

get_row <- function(cell_name) {
  index <- as.numeric(str_locate(cell_name, "_")[1]) - 1
  return(as.numeric(substr(cell_name, 1, index)))
}
```

In [17]:
```
# Measure the time it takes to fill in the 10K missing values.

start <- Sys.time()
# Create cell names.
suffixes <- paste(rep("_", dim(df)[2]), as.character(1:dim(df)[2]), sep="")
prefixes <- as.character(1:dim(df)[1])

first_arg <- as.vector(mapply(rep, prefixes, rep(dim(df)[2], length(prefixes))))
vnames <- paste(first_arg, rep(suffixes, length(prefixes)), sep="")

# Here is df in flattened form:
df_flat <- as.list(t(df))
names(df_flat) <- vnames


# Identify the cells in df with missing values.
missing <- names(which(is.na(df_flat)))
length(missing)
# [1] 10000
head(missing)
# [1] "1_5"   "2_12"  "8_11"  "10_11" "15_12" "18_7"


miss_cols <- mapply(get_column, missing)
names(miss_cols) <- missing
miss_rows <- mapply(get_row, missing)
names(miss_rows) <- missing

# Fill the missing values in each column.
for(i in 1:dim(df)[2]) {
  cells <- names(miss_cols[as.numeric(miss_cols)== i])
  rows <-as.numeric(miss_rows[cells])
  df[rows, i] <- imputed_vals[i]
}

stop <- Sys.time()
round(stop - start, 4)
# Time difference of 0.4521 secs
```

10000

'1_5' '2_12' '8_11' '10_11' '15_12' '18_7'

```
Time difference of 0.4226 secs
```

Named vector single test results for 10K NAs: 0.4226 seconds

In [18]:
```
# Check that the missing values were correctly filled in.

head(missing)
```

'1_5' '2_12' '8_11' '10_11' '15_12' '18_7'

In [19]:
```
print(head(miss_cols))
print(head(miss_rows))
```

```
 1_5  2_12  8_11 10_11 15_12  18_7
   5    12    11    11    12     7
 1_5  2_12  8_11 10_11 15_12  18_7
```

In [20]:
```
# We see from the following test that all missing values have been
# correctly filled in.

all_good <- FALSE
incorrect <- 0
for(i in 1:length(missing)) {
    val <- round(df[as.numeric(miss_rows[i]), as.numeric(miss_cols[i])], 2)
    true_val <- round(imputed_vals[miss_cols[i]], 2)
    if(val != true_val) { incorrect <- incorrect + 1 }
}
if(incorrect == 0) { all_good <- TRUE }
ifelse(all_good, "All is good!", "Fill did not work!")
```

'All is good!'

Named Vector Results for 100 trials: *smaller* sets of NAs

In [21]:
```
# Create trial sizes exactly as in the corresponding test above:

set.seed(99871)
smp <- sample(2000:20000, 100, replace= FALSE)
```

In [22]:
```
# This function constructs a dataframe with missing values
# and then fills in the missing values using the above Named
# Vector approach.

method_vector <- function(n_vals, seed) {

  df <- modDat[, c(1:9, 12:15)]
  set.seed(seed)
  yy <- sample(1:13, n_vals, replace= TRUE)
  xx <- sample(1:nrow(df), n_vals, replace= TRUE)

  for(i in 1:n_vals) {
    df[xx[i], yy[i]] <- NA
  }

  imputed_vals <- rep(NA, dim(df)[2])
  for(i in 1:length(imputed_vals)) {
    imputed_vals[i] <- median(df[, i], na.rm= TRUE)
  }

  start <- Sys.time()
  # Create cell names.
  suffixes <- paste(rep("_", dim(df)[2]), as.character(1:dim(df)[2]), sep="")
  prefixes <- as.character(1:dim(df)[1])
  first_arg <- as.vector(mapply(rep, prefixes, rep(dim(df)[2], length(prefixes))))
  vnames <- paste(first_arg, rep(suffixes, length(prefixes)), sep="")

  # df in flattened form:
  df_flat <- as.list(t(df))
  names(df_flat) <- vnames

  # Identify the cells in df with missing values.
  missing <- names(which(is.na(df_flat)))
  miss_cols <- mapply(get_column, missing)
  miss_rows <- mapply(get_row, missing)
  names(miss_rows) <- names(miss_cols) <- missing

  # Fill the missing values in each column.
  ### NOTE: We can increase the speed of this loop by walking through
  ### only those columns with missing values.  This is easy to do, and
  ### becomes more important as the number of variables in the dataset
```

```
  ### increases.  (This is implemented in the Transposed Columns approach.)
  for(i in 1:dim(df)[2]) {
    cells <- names(miss_cols[as.numeric(miss_cols)== i])
    rows <-as.numeric(miss_rows[cells])
    df[rows, i] <- imputed_vals[i]
  }

  stop <- Sys.time()
  delta <- as.numeric(stop - start)
  return(delta)
}
```

In [23]:
```
# Run the test.

result <- 0
for(i in 1:length(smp)) {
  seed <- 8871 * i
  nv_time <- method_vector(smp[i], seed)
  result <- result + nv_time
}
(nv_avg_time_tst01 <- round(result/length(smp), 4))
# 0.3013 second
```

0.3013

Named Vector 100 trials, test01 average time: 0.3013 seconds (= 301ms)

For the Sequential Walk-Through method, this test result was 2.2681 seconds. In other words, the Named Vector approach is **7.5 times faster** on this test set.

Named Vector Results for 100 trials: *larger* sets of NAs

In [24]:
```
# Get exactly the same test set for the test02 test above.

set.seed(99871)
smp <- sample(20000:50000, 100, replace= FALSE)
```

In [25]:
```
# Run the test.

result <- 0
for(i in 1:length(smp)) {
  seed <- 8871 * i
  nv_time <- method_vector(smp[i], seed)
  result <- result + nv_time
}
(nv_avg_time_tst02 <- round(result/length(smp), 4))
# 0.7867 second
```

0.7867

Named Vector 100 trials, test02 average time: 0.7867 seconds (= 787ms)

For the Sequential Walk-Through method, this test result was 3.0131 seconds. So the Named Vector approach is **3.8 times faster** on this test set.

Comments

On average, across 200 trials, the Named Vector approach is about **5.65** times faster than the Sequential Walk-Through approach. The performance gain is much less than I expected.

Transposed Columns Approach

Here is a better way to vectorize the process. Rather than constructing names for the row-column locations of each cell---names which later have to be re-interpreted into numeric values---we can make use of R's implicit indices for the rows and columns.

In [26]:
```
df <- modDat[, c(1:9, 12:15)]
```

In [27]:
```
# Again create some missing values in df for an initial test.

# As above (in the Sequential Walk-Trhough section) insert NA in
# 10,000 randomly selected cells of the dataframe.  The same
# seed is used from the above, corresponding test.

set.seed(4331)
n <- 10000
yy <- sample(1:13, n, replace= TRUE)
xx <- sample(1:nrow(df), n, replace= FALSE)

for(i in 1:n) {
  df[xx[i], yy[i]] <- NA
}
```

In [28]:
```
# Compute the median of each variable.

imputed_vals <- rep(NA, dim(df)[2])
for(i in 1:length(imputed_vals)) {
  imputed_vals[i] <- median(df[, i], na.rm= TRUE)
}
named_imputed_vals <- imputed_vals
names(named_imputed_vals) <- 1:13
print(round(named_imputed_vals, 2))
```

```
        1         2         3         4         5         6         7         8 
  -118.50     34.26     29.00   2125.00    436.00   1165.50    409.00      3.53 
        9        10        11        12        13 
179800.00      7.72      5.23      0.20      2.82 
```

In [29]:
```
# Identify the cells which have missing values.
# Note the transposing of the dataframe.

ans <- which(is.na(t(df)))
print(head(ans))
# [1]   5    25    102    128    194    228
```

```
[1]   5  25 102 128 194 228
```

In [30]:
```
# Extracting the modulo gives us the column number; 0 = 13.

(remainder <- head(ans) %% 13)
```

```
5 12 11 11 12 7
```

In [31]:
```
# Integer dividing by 13 (= dim(df)[2]) helps us
# obtain the row number:

head(ans) %/% 13 + ceiling(remainder/dim(df)[2])
```

```
1 2 8 10 15 18
```

In [32]:
```
# Get columns with missing values.

remainder <- ans %% 13
columns <- remainder
columns[which(columns==0)] <- 13
```

```
cols_uniq <- sort(unique(columns)); cols_uniq

length(columns)
# 10,000
```

1 2 3 4 5 6 7 8 9 10 11 12 13

10000

In [33]:
```
# Get rows with missing values.

rows <- ans %/% 13 + ceiling(remainder/dim(df)[2])
length(rows)
# 10,000
```

10000

In [34]:
```
# Name each column entry by its row:

names(columns) <- as.character(rows)
print(head(columns))
```

```
 1  2  8 10 15 18
 5 12 11 11 12  7
```

In [35]:
```
# Fill the cells which have missing values.

for(col in cols_uniq) {
  rowsTofill <- as.numeric(names(columns[as.numeric(columns)==col]))
  df[rowsTofill, col] <- imputed_vals[col]
}
```

In [36]:
```
# Check that the fill worked as expected.
# NOTE: 'miss_rows' and 'miss_cols' were created above,
# in the Named Vector section.

all_good <- FALSE
incorrect <- 0
for(i in 1:length(columns)) {
    val <- round(df[as.numeric(miss_rows[i]), as.numeric(miss_cols[i])], 2)
    true_val <- round(imputed_vals[miss_cols[i]], 2)
    if(val != true_val) { incorrect <- incorrect + 1 }
}
if(incorrect == 0) { all_good <- TRUE }
ifelse(all_good, "All is good!", "Fill did not work!")
```

'All is good!'

Transposed Columns Results for 100 trials: *smaller* sets of NAs

In [9]:
```
# This function constructs a dataframe with missing values and
# then fills in the missing values using the above Transposed
# Columns approach.

tc02 <- function(n_vals, seed) {

  df <- modDat[, c(1:9, 12:15)]
  set.seed(seed)
  yy <- sample(1:13, n_vals, replace= TRUE)
  xx <- sample(1:nrow(df), n_vals, replace= TRUE)

  for(i in 1:n_vals) {
    df[xx[i], yy[i]] <- NA
  }

  imputed_vals <- rep(NA, dim(df)[2])
```

```
  for(i in 1:length(imputed_vals)) {
    imputed_vals[i] <- median(df[, i], na.rm= TRUE)
  }

  start <- Sys.time()
  # Identify the cells which have missing values.
  df_t <- which(is.na(t(df)))

  ## get columns with missing values
  remainder <- df_t %% 13
  columns <- remainder
  columns[which(columns==0)] <- 13
  cols_uniq <- sort(unique(columns))

  ## get rows with missing values
  rows <- df_t %/% 13 + ceiling(remainder/dim(df)[2])
  # Name each column entry by its row:
  names(columns) <- as.character(rows)

  for(col in cols_uniq) {
    rowsTofill <- as.numeric(names(columns[as.numeric(columns)==col]))
    df[rowsTofill, col] <- imputed_vals[col]
  }
  stop <- Sys.time()
  delta <- as.numeric(stop - start)
  return(delta)
}
```

In [66]:
```
# Create trial sizes exactly as in the corresponding test above:

set.seed(99871)
smp <- sample(2000:20000, 100, replace= FALSE)
```

In [67]:
```
# Run the test.

result <- 0
for(i in 1:length(smp)) {
  seed <- 8871 * i
  tc02_time <- tc02(smp[i], seed)
  result <- result + tc02_time
}
(tc_avg_time <- round(result/length(smp), 4))
# 0.0139 second
```

0.0139

Transposed Columns 100 trials, test01 average time: 0.0139 seconds (= 14ms)

For the Sequential Walk-Through method, this test result was 2.2681 seconds. In other words, the Transposed Columns approach is **163 times faster** on this test set.

Transposed Columns Results for 100 trials: *larger* sets of NAs

In [10]:
```
# Get exactly the same test set for the test02 test above.

set.seed(99871)
smp <- sample(20000:50000, 100, replace= FALSE)
```

In [11]:
```
# Run the test.

result <- 0
for(i in 1:length(smp)) {
  seed <- 8871 * i
  tc02_time <- tc02(smp[i], seed)
  result <- result + tc02_time
```

```
}
(tc_avg_time <- round(result/length(smp), 4))
# 0.0325 second
```

0.0325

Transposed Columns 100 trials, test02 average time: 0.0325 seconds (= 32.5ms)

For the Sequential Walk-Through method, this test result was 3.0131 seconds. So the Transposed Columns approach is **93 times faster** on this test set.

Final Comments

Across 200 trials and a wide range of percent missing values (as a percent of the number of cells in the dataframe), we see that the Transposed Columns method is, on average, **128 times faster** than the Sequential Walk-Through approach.

The above results suggest that this performance gain increases the smaller the number of missing values that have to be filled in. It is also reasonable to infer that this performance gain is proportional to the size of the dataframe that we have to write to. However, the relative performance gain will decrease the more we need to customize what we do for each of the variables with missing values. For example, if we want to forward-fill for one variable, interpolate for another, and use the median (as we did above) for a third, the Transposed Columns approach will not yield as great a gain over sequentially walking through the dataframe as when we simply use the same method for each variable when filling in missing values.

* * * * *

In [ ]: