**Background**

An attempt to produce corelation tests for 60 questions in a survey, using the corr.test() function from the psych package.

The answers from each question were coded as 5 point scales from 1 – 5. The input data frame included 8,219 observations. When she ran corr.test(), her computer did not complete the analysis after 2 hours. This led her to post a question on Stackoverflow.com, “Is there any method to estimate the R script running time?”

**An initial answer**

Although the question did not include enough data to be considered a reproducible example on Stack, one of the things that I appreciate about R is its ability to quickly generate simulated data. Three lines of code later, I have a simulated survey including 9,000 respondents of 60 questions with responses from 1 – 5.

# create 9000 rows of data w/ 60 columns

system.time(data <- as.data.frame(matrix(round(runif(9000\*60,min = 1, max = 5)),

nrow = 9000)))

id <- 1:9000

data <- cbind(id,data)

With some additional code we can calculate timings on the psych::corr.test() function. Given that the question noted that corr.test() failed to produce a result in 2 hours on a laptop with 8Gb RAM and a 2.5Ghz two core processor, I used lapply() to process a vector of numbers of observations and process them with corr.test().

observations <- c(100,200,500,1000,2000)

theTimings <- lapply(observations,function(x){

system.time(r <- corr.test(data[id <= x,2:61],method = "kendall"))

})

theNames <- paste0("timings\_",observations,"\_obs")

names(theTimings) <- theNames

theTimings

Elapsed times for the analysis ranged from 0.46 seconds with 100 observations to 106.6 seconds with 2,000 observations, as illustrated below.

> theTimings

$timings\_100\_obs

user system elapsed

0.435 0.023 0.457

$timings\_200\_obs

user system elapsed

1.154 0.019 1.174

$timings\_500\_obs

user system elapsed

5.969 0.026 5.996

$timings\_1000\_obs

user system elapsed

24.260 0.045 24.454

$timings\_2000\_obs

user system elapsed

106.465 0.109 106.603

**Generating Predictions**

We can quickly fit a linear model to these timings and use it to predict the runtime for larger data sets. We create a data frame with the timing information, fit a model, and print the model summary to check the R^2 for goodness of fit. Since this is an exploratory data analysis, I didn’t bother writing code to extract the timings returned by lapply() into a vector.

time <- c(0.457,1.174,5.996,24.454,106.603)

timeData <- data.frame(observations,time)

fit <- lm(time ~ observations, data = timeData)

summary(fit)

The summary indictes that a linear model appears to be a good fit with the data, recognizing we used a small number of observations as input to the model.

> summary(fit)

Call:

lm(formula = time ~ observations, data = timeData)

Residuals:

1 2 3 4 5

9.808 4.906 -7.130 -16.769 9.186

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept) -14.970240 8.866838 -1.688 0.18993

observations 0.056193 0.008612 6.525 0.00731 \*\*

---

Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 13.38 on 3 degrees of freedom

Multiple R-squared: 0.9342, Adjusted R-squared: 0.9122

F-statistic: 42.57 on 1 and 3 DF, p-value: 0.007315

Next, we build another data frame with additional numbers of observations and use it to generate predicted timings via the stats::predict() function.

predictions <- data.frame(observations = c(3000,4000,5000,6000,7000,8000,9000))

data.frame(observations = predictions,predicted = predict(fit,predictions))

Given this model, the 9,000 observation data frame should take about 8.2 minutes to run corr.test() on the laptop I used for this analysis, a 2015 era Macbook Pro 15 with an Intel i7-4870HQ four core processor.

> data.frame(observations = predictions,predicted = predict(fit,predictions))

observations predicted

1 3000 153.6102

2 4000 209.8037

3 5000 265.9971

4 6000 322.1906

5 7000 378.3841

6 8000 434.5776

7 9000 490.7710

> 490 / 60

[1] 8.166667

>

The original question stated that corr.test() failed to complete within 2 hours on a 2 core 2.4Ghz processor, which leads us to the hypothesis that there is a non-linear effect in runtime that becomes prominent beyond 2,000 observations. We need to generate data on runs with more observations to determine whether we can discern an effect that makes the algorithm degrade to less than linear scalability.

**Improving the Model**

One way to capture a non-linear effect is to add a quadratic effect as an independent variable in the model. We collect data for 3,000, 4,000, and 5,000 observations in order to increase the degrees of freedom in the model, as well as to provide more data from which we might detect a quadratic effect.

> theTimings

$timings\_3000\_obs

user system elapsed

259.444 0.329 260.149

$timings\_4000\_obs

user system elapsed

458.993 0.412 460.085

$timings\_5000\_obs

user system elapsed

730.178 0.839 731.915

>

Next, we run linear models with and without the quadratic effect, generate predictions, and compare the results. The summary() for the quadratic model is very interesting.

observations <- c(100,200,500,1000,2000,3000,4000,5000)

obs\_squared <- observations^2

time <- c(0.457,1.174,5.996,24.454,106.603,260.149,460.085,731.951)

timeData <- data.frame(observations,obs\_squared,time)

fitLinear <- lm(time ~ observations, data = timeData)

fitQuadratic <- lm(time ~ observations + obs\_squared, data = timeData)

summary(fitQuadratic)

> summary(fitQuadratic)

Call:

lm(formula = time ~ observations + obs\_squared, data = timeData)

Residuals:

1 2 3 4 5 6 7 8

-0.2651 0.2384 0.7455 -0.2363 -2.8974 4.5976 -2.7581 0.5753

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept) 1.121e+00 1.871e+00 0.599 0.5752

observations -7.051e-03 2.199e-03 -3.207 0.0238 \*

obs\_squared 3.062e-05 4.418e-07 69.307 1.18e-08 \*\*\*

---

Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.764 on 5 degrees of freedom

Multiple R-squared: 0.9999, Adjusted R-squared: 0.9999

F-statistic: 3.341e+04 on 2 and 5 DF, p-value: 4.841e-11

Not only has R^2 improved to .9999 with the quadratic term in the model, both the linear and quadratic terms are significantly different from zero at alpha = 0.05. Interestingly, with a quadratic term in the model, the linear effect is negative.

**Comparing prediction results**

Our penultimate step is to generate predictions for both models, combine them into a data frame and print the results.

predLinear = predict(fitLinear,predictions)

predQuadratic <- predict(fitQuadratic,predictions)

data.frame(observations = predictions$observations,

obs\_squared = predictions$obs\_squared,

predLinear,

predQuadratic)

observations obs\_squared predLinear predQuadratic

1 3000 9.0e+06 342.6230 255.5514

2 4000 1.6e+07 482.8809 462.8431

3 5000 2.5e+07 623.1388 731.3757

4 6000 3.6e+07 763.3967 1061.1490

5 7000 4.9e+07 903.6546 1452.1632

6 8000 6.4e+07 1043.9125 1904.4181

7 9000 8.1e+07 1184.1704 2417.9139

**Conclusions**

First, as we added data at larger numbers of observations in the corr.test(), the linear prediction at 9,000 observations more than doubled from 491 seconds to 1,184 seconds. As expected, adding data to the model helped improve its accuracy.

Second, the time prediction of the quadratic model was more than twice the runtime as the linear model, 2,417.9 seconds.

**And the drumroll, please…**

I ran the 9,000 observation data frame through the test, and it took 40 minutes to complete. The runtime was 5X the duration of the original linear prediction from runs up to 2,000 observations, and slightly more than 2X the prediction from runs up to 4,000 observations.

> # validate model

> system.time(r <- corr.test(data[,2:61],method = "kendall"))

user system elapsed

2398.572 2.990 2404.175

> 2404.175 / 60

[1] 40.06958

In contrast, the quadratic model is stunningly accurate in its prediction, where the predicted runtime of 2,418 seconds was within 0.6% of the actual value.

**Why didn’t the original poster’s analysis complete in 2 hours?**

In the back and forth comments posted on Stack as I developed the models, a question was raised about the relevance of multiple CPU cores in runtime performance when the corr.test() function uses a single thread to process the data.

I ran a second series of tests on an HP Spectre x-360 with an Intel i7-U6500 CPU that also runs at 2.5Ghz, but only has 2 cores. Its processing time degrades faster than that of the Intel i7-4870HQ CPU (4 cores / 2.5Ghz), as illustrated by the following table.

| **observations** | **time x360-13** | **time Macbook15** | **pct difference** |
| --- | --- | --- | --- |
| 100 | 0.56 | 0.457 | 22.53829 |
| 200 | 1.55 | 1.174 | 32.02726 |
| 500 | 8.36 | 5.996 | 39.42628 |
| 1000 | 35.33 | 24.454 | 44.47534 |
| 2000 | 151.92 | 106.603 | 42.51006 |
| 3000 | 357.45 | 260.149 | 37.40203 |
| 4000 | 646.56 | 460.085 | 40.53055 |

As we can see from the table, the i7-U6500 is 22.5% slower than the i7-4870HQ at 100 observations, and this deficit grows as the number of observations including in the timing simulations increases to 4,000.