Project Five: STA 141B

Benjamin Jewell

General Approach:

For this project I wrote a pre-allocated and vectorized version of the URLdecode function to increase run time. Data was generated by sampling UTF-8 characters and standard letters and numbers to test the run times. After verifying the results were correct this data was graphed to look for a general trend. Linear regression was then applied to try to predict how long it would take for the special 591k character string. For my approach I chose 1000 samples of size 1 to around 100,000, as this already took me around 30 minutes to run and larger values did not seem time efficient for me.

Errors:

When working on the v2 preallocated version of URLdecode I mostly ran into issues with using the raw type and using a vector of it's length. Originally I was inserted the transformed UTF-8 characters into the ith position which left various NA's throughout the string which did not allow it to be converted back into characters. After watching the transformation step by step to see what was happening I changed the code to add the new character at the jth index, that is, right after the previous character and avoid this.

Writing the vectorization formula was plagued by lots of issues as I tried to shape it into a function that was actually vectorized. Originally I had planned to use matrices to try to solve the problem but ran into issues of dimensions often and had to constantly check them to make sure they lined up. Here the main issue was indexing, as while it was easy to find the percent signs in the input strings via vector operations, it was a bit harder to selected the following two characters and also delete the percent signs. This was another case of using lots of print statements to watch the process and move through it step by step. This whole section really made me think and I often resorted to writing out on paper what I wanted the computer to do vs what was actually happening in each step and trying to align these two.

When comparing data I had originally been using matrices to store the run times and results, but I kept running into type issues. Eventually I remembered from class that matrices are a single type and had to add some tweaks to account for it being strings.

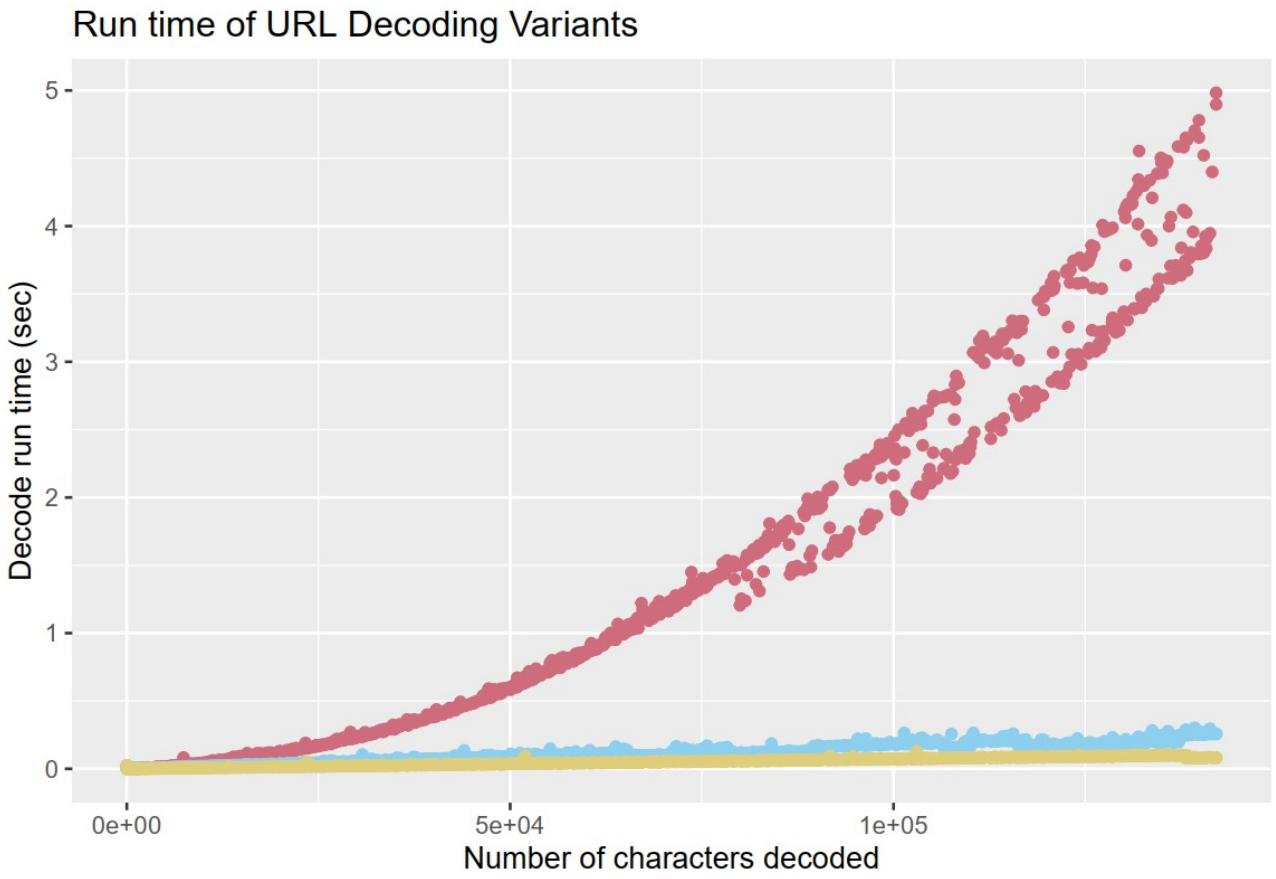
My only other place with major errors was trying to get ggplot to graph my plots properly with the right colors. Here I had to turn to the internet for a bit of help getting ggplot to work, as well as going back through my old ggplot exercises from STA 141A.

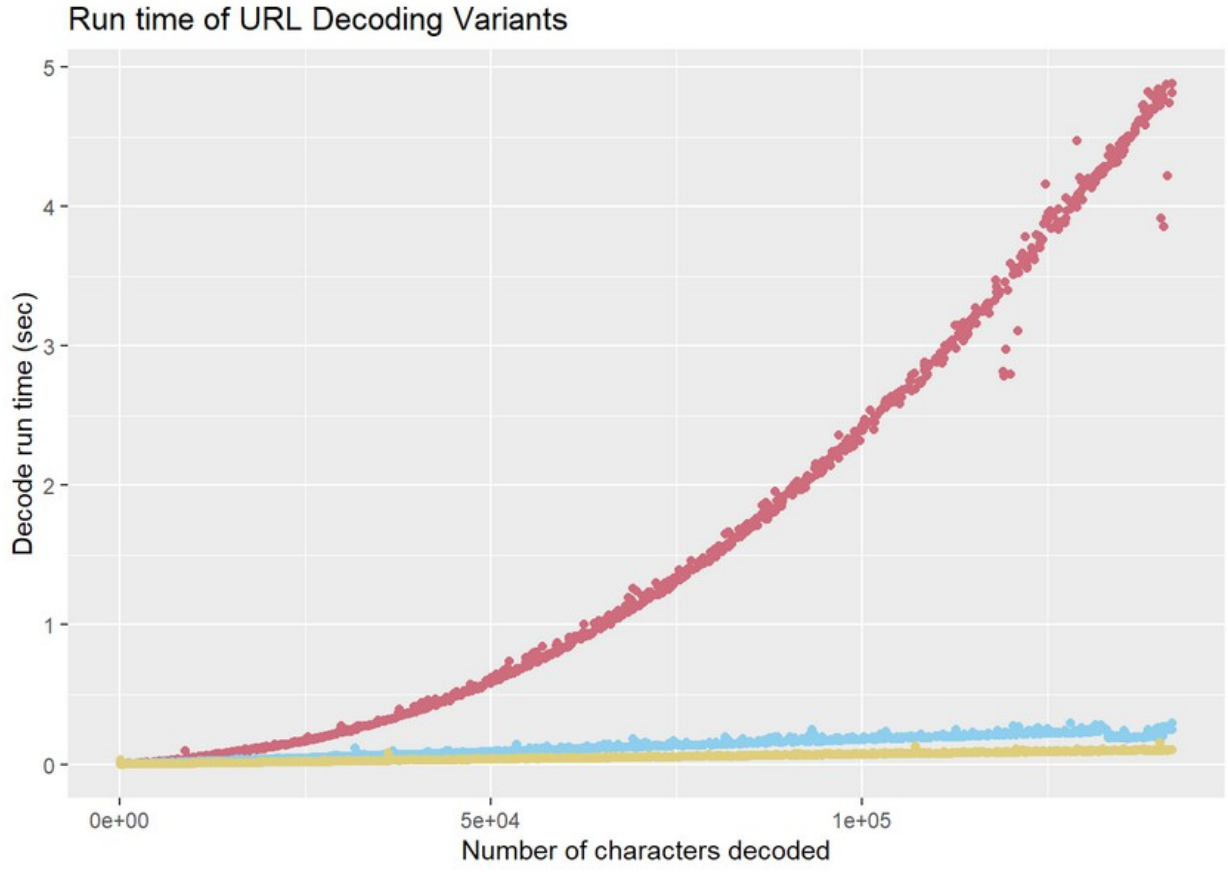
Verification:

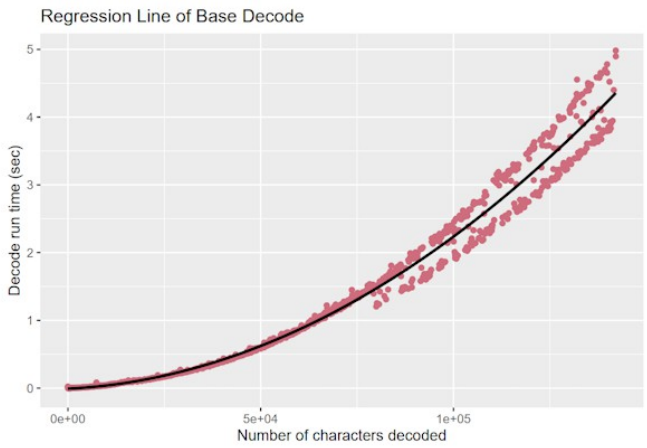
Obviously while speed is what we are trying to improve here, we also care a lot about if our results are correct. For this I chose to simply compare my results to the results of URLdecode and ensure that they are the same, as if the official R URLdecode function is wrong then I think we have bigger fish to fry. Since we already have to compute the run times and results of the base URLdecode for our graphs, I chose to reuse those same results in my comparisons with the other two functions to avoid having to run URLdecode twice.

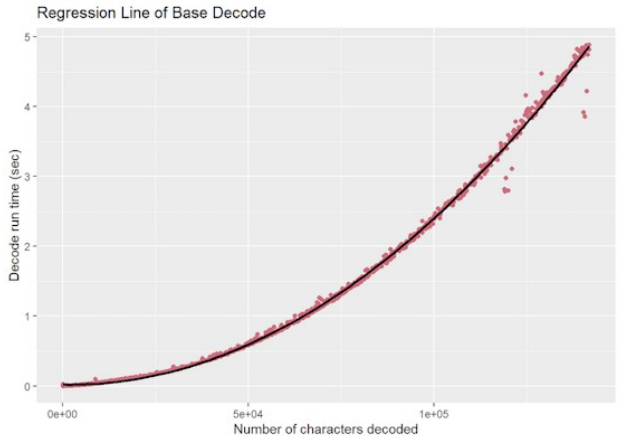
Run Time Curves:

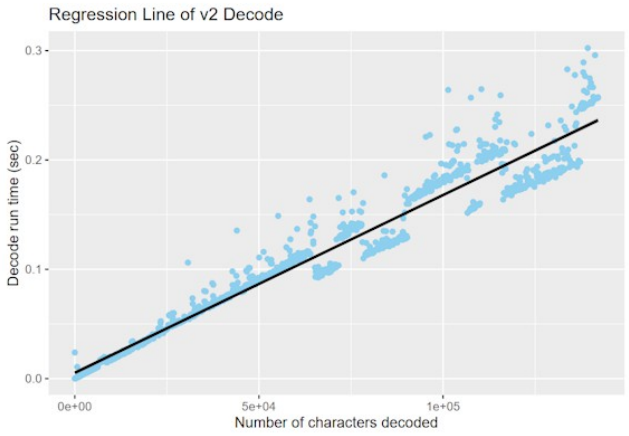
For this project I made a graph of each model's run time and attempted to fit a linear regression to them. I ran this process twice, giving us two graphs to compare as seen below. Our first graph shows the run times of all three functions, followed by a second graph of the same process. You'll notice that there is an odd spread to the first graph, I believe this might have been due to the knitting process of saving my code. I also saved the graphs from my earlier practice run and will include them below.

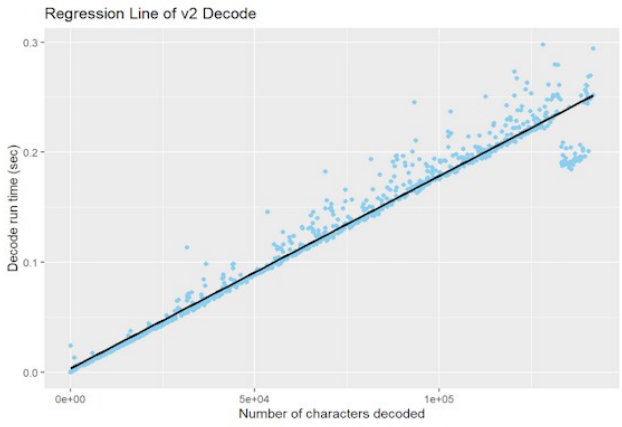

Run time of URL Decoding Variants
Decode run time (sec)
0
1
2
3
4
5
0e+00
5e+04
1e+05
Number of characters decoded

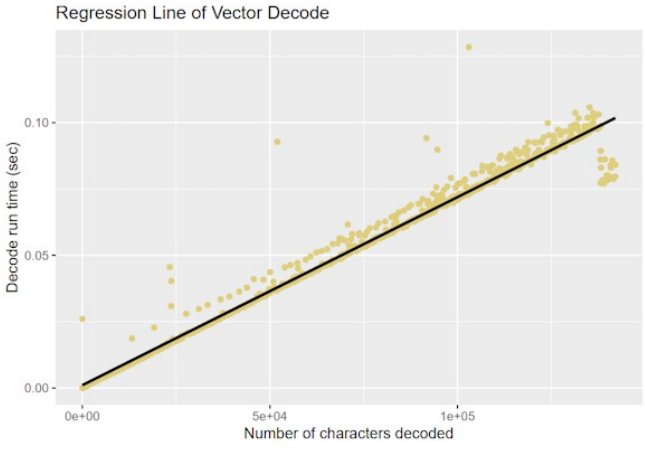

Run time of URL Decoding Variants
Decode run time (sec)
0
1
2
3
4
5
0e+00
5e+04
1e+05
Number of characters decoded

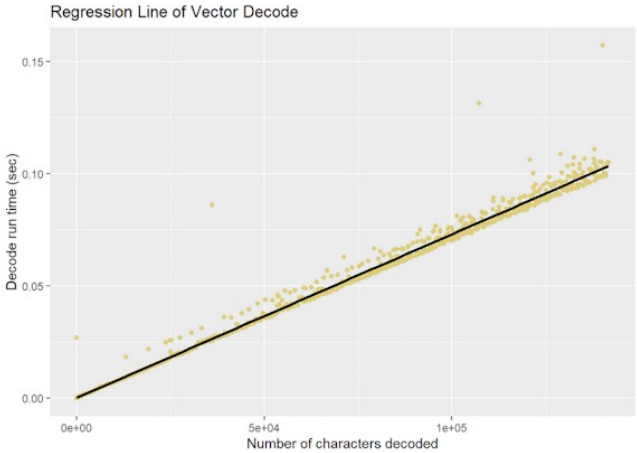












Above the left most graphs are from the kniting process, while the right graphs are from an earlier run. As we can see, our two new models are both improvements of compared to URLdecode. Our vectorized performed the best in most cases, though from experience I know in some cases (as well will see at the end) it seems like the preallocations can beat out the vectorized approach.

Prediction for 591k Characters:

Using our sample data points we attempted to fit linear models to predict how long it would take each function. As seen below our estimates were all off, under estimating for the Base and Vector

decodes, while over estimating for the v2 decode. The largest mistake was for our base prediction as the prediction was 70.56 seconds, while the actual run time was 299.23 seconds, giving us the very large MSE of 52289.87. It seems that unfortunately our model is not the best predictor for our run time as the data gets larger, though I suspect because of the unusual shape of the graph this may have led to a larger error.

Interestingly it seems that the preallocation model is actually faster on our large file when I knit this code, rather than the vectorized version as we might expect. I'm not particularly sure why this is, but perhaps it could simply be that my computer suddenly started to do a second task at the same time as the URL_vec run.

Model	Predicted Run Times (sec)	Real Run Times (sec)
URLdecode	70.5604513	299.239938
URLdecode_v2	0.9677605	0.440223
URL_vec	0.4205448	0.869216

Functions:

URLdecode_v2: This is my version of the preallocation function of URLdecode. Here I set the output vector to the number of characters of the input URL, as for as far as I'm aware the output should never be bigger then the input. The issue is that if you only do this then there will be blank spots in the output vector because you're adding at the ith index but the UTF-8 characters are three characters long originally. Thus we add a second index j to keep track of the position that we add on, thus basically appending it but not having to rebuild the output vector each time. The rawToChar will trim off the empty spaces at the end of our function when we return it.

URLdecode_tp: This is a failed attempted to vectorize URLdecode using the tapply function. I ended up not using it for this project but decided to keep the code here since it was a good learning experience for me on how to use tapply.

URL_vec: This is my vectorized approach to the problem, which runs faster than the other two URLdecode methods most of the time. It works by using a tad bit of concatenating to “select” where the two digits following a % sign are. Those numbers are then pulled out and the normal URLdecode arithmetic is performed to transform them. However the hard part here is adding them pairwise and putting them back in the right spots. At the suggestion of Professor Duncan I change the dimension of this vector to two, then column sum them together before inserting them back in. We use logical arguments to then drop the % signs from our vector and return it.

gen_data: This function generates our sample data for testing. We generate our test URLs by sampling a combination of the UTF-8 characters (downloaded and saved off of the W3 webpage), the lower case and upper case letters and the numbers 0 to 9. All non UTF-8 characters are then duplicated twice so that they have a larger proportion in our population, as otherwise our test URLs are mostly UTF-8 characters due to how many UTF-8 characters there are.

The function takes a number of samples to take, then begins generating strings, each 5 larger than the previous. Once it has generated 100 strings, the size steps up by 50 each string.

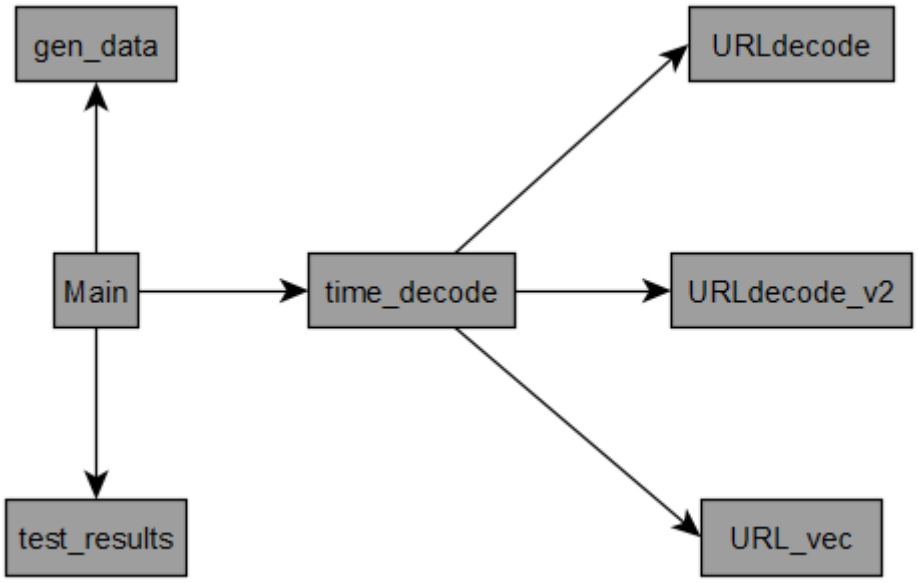
time_decode: This function takes a URL decoding function, a number of data points and test data. It

then begins to run the decoding function on each test string. The results, run time and test data are all saved in a matrix and then returned.

test_results: This function takes a matrix of truth values and test values. It then compares them to ensure they are the same, returning a warning if they do not. The truth value matrix should originate from the test results of URLdecode, as we are trusting that the official R package is correct. We do this instead of regenerating these results to avoid a higher run time (as we'd be calling URLcode twice on each string then).

Main: The main function simply takes a number of tests you want to performed. It first generates that many test strings, then passes them onto our three functions: URLdecode, URLdecode_v2, and URL_vec. The results are made in time_decode, then verified in test_results, then saved and returned (so that we can graph them later).

Here is a graph of how the functions are all called from Main. Note that the graphing is done outside of any function after main is finished being called to generate our data.



Works Cited:

[1] https://statisticsglobe.com/count-occurrence-of-certain-character-in-string-in-r

[2] https://statisticsglobe.com/apply-functions-in-r/#example-6-mapply-function

[3] https://stackoverflow.com/questions/57153428/r-plot-color-combinations-that-are-colorblind-accessible

[4] https://www.statology.org/r-create-vector-of-zeros/

[5] https://stackoverflow.com/questions/23635662/editing-legend-text-labels-in-ggplot

[6] https://stackoverflow.com/questions/42764028/fitting-a-quadratic-curve-in-ggplot

[7] https://thomasadventure.blog/posts/ggplot-regression-line/

HW5 - Optimize - Jewell

Benjamin Jewell

2023-06-08

Libraries

```
library(ggplot2)
library(knitr)
library(XML)
utf_chars = readRDS('E:\\College\\UC Davis\\STA141B\\HW5\\utf_list')
mtxt = read.table('E:\\College\\UC Davis\\STA141B\\HW5\\PercentEncoded_mini.txt')
full_txt = read.table('E:\\College\\UC Davis\\STA141B\\HW5\\PercentEncodedString.txt')
```

```
## Warning in read.table("E:\\College\\UC
## Davis\\STA141B\\HW5\\PercentEncodedString.txt"): incomplete final line found by
## readTableHeader on 'E:\College\UC Davis\STA141B\HW5\PercentEncodedString.txt'
```

The Original URLdecode function

```
start_URLbase = Sys.time()
URLdecode <- function (URL)
{
    vapply(URL, function(URL) {
        x <- charToRaw(URL)
        pc <- charToRaw("%")
        out <- raw(0L)
        i <- 1L

        while (i <= length(x)) {
            if (x[i] != pc) {
                out <- c(out, x[i])
                i <- i + 1L
            }
            else {
                y <- as.integer(x[i + 1L:2L])
                y[y > 96L] <- y[y > 96L] - 32L
                y[y > 57L] <- y[y > 57L] - 7L
                y <- sum((y - 48L) * c(16L, 1L))
                out <- c(out, as.raw(as.character(y)))
                i <- i + 3L
            }
        }
        rawToChar(out)
    }, character(1), USE.NAMES = FALSE)
}
```

```
#URLdecode(read.table('E:\\College\\UC Davis\\STA141B\\HW5\\PercentEncoded_mini.txt'))
#URLdecode('Hello%20there%20')
#URLdecode(mtxt)
#URLdecode('%58%E2%80%93A')
#print(Sys.time() - start_URLbase)
```

URLbase Conversion

```
start_URLbase = Sys.time()
URLdecode_v2 <- function (URL)
{
    vapply(URL, function(URL) {
        x <- charToRaw(URL)
        pc <- charToRaw("%")
        out <- raw(length = nchar(URL))
        i <- 1L
        j <- 1L
        while (i <= length(x)) {
            if (x[i] != pc) {
                out[j] = x[i]
                i <- i + 1L
            }
            else {
                y <- as.integer(x[i + 1L:2L])
                y[y > 96L] <- y[y > 96L] - 32L
                y[y > 57L] <- y[y > 57L] - 7L
                y <- sum((y - 48L) * c(16L, 1L))
                out[[j]] <- as.raw(as.character(y))
                i <- i + 3L
            }
            j <- j + 1L
        }
        rawToChar(out)
    }, character(1), USE.NAMES = FALSE)
}
#URLdecode_v2(read.table('E:\\College\\UC Davis\\STA141B\\HW5\\PercentEncodedString.txt'))
#URLdecode_v2('20%20k%20j')
#URLdecode_v2(full_txt)
#print(Sys.time() - start_URLbase)
```

TApply Approach - Failed Vectorization

```
start_URLbase = Sys.time()
URLdecode_tp <- function(URL){
        r <- charToRaw(URL)
        pc <- charToRaw("%")

        mt = matrix(data = FALSE, nrow = 3, ncol = (nchar(URL)+2))
        mt[1,3:(nchar(URL)+2)] = c(r == pc)
        mt[2,2:(nchar(URL)+1)] = c(r == pc)
        mt[3,1:(nchar(URL)+0)] = c(r == pc)
        mt = colSums(mt)
```

```
        m = cumsum(as.numeric(!as.logical(mt)) + c((r == pc), 0, 0))

        k = tapply(r, m[1:(length(r))], function(x)
            if (length(x) > 1){
                y <- as.integer(x[2:3])
                y[y > 96L] <- y[y > 96L] - 32L
                y[y > 57L] <- y[y > 57L] - 7L
                x <- as.raw(as.character(sum((y - 48L) * c(16L, 1L))))
            } else {x = x})

        #print(k)
        #print('---')

        rawToChar(unlist(k))
}

#URLdecode_tp(mtxt)
#URLdecode_tp('mk%20k%24j')
#print(Sys.time() - start_URLbase)
```

Vectorized Approach

```
#mtxt = read.table('E:\\College\\UC Davis\\STA141B\\HW5\\PercentEncoded_mini.txt')[[1]]
start_URLbase = Sys.time()
URL_vec <- function(URL){vapply(URL, function(URL) {
        r <- as.integer(charToRaw(URL))
        pc <- charToRaw("%")

        pnums <- (c(0,(r==pc),0) + c(0,0,(r==pc)))[1:length(r)]

        y = r[as.logical(pnums)]
        y[y > 96L] <- y[y > 96L] - 32L
        y[y > 57L] <- y[y > 57L] - 7L
        y = (y-48L) * c(16L, 1L)
        attr(y, 'dim') = c(2,sum(pnums)/2)
        y = colSums(y)

        r = r[!as.logical(pnums)]
        r[r == pc] = y


        rawToChar(as.raw(as.character(r)))
    }, character(1), USE.NAMES = FALSE)
}

#URL_vec(mtxt)
#URL_vec('mk%20k%24j')
#URL_vec('%58%E2%80%93A')
#URL_vec(full_txt)
#print(Sys.time() - start_URLbase)
```

===== TESTING OUR RESULTS =====

Generating some data

```
#All utf_chars encoded
utf_chars = readRDS('E:\\College\\UC Davis\\STA141B\\HW5\\utf_list')

#Random letters
words1 = sample(letters, 1000, replace = TRUE)

#Random words of length 2
words2 = list()
for (i in 1:1000){
  words2[i] = paste(unlist(sample(letters, 2, replace = TRUE)), collapse='')
}
```

Generate Data

```
gen_data <-function(n, stepfactor = 1L){
  data = list()
  all_chars = append(rep(letters, 3), append(rep(LETTERS,3), append(rep(c(0:9),3) , utf_chars)))

  j = 1L
  for (i in seq(1, n)){
    data[i] = paste(sample(all_chars, j, replace = TRUE), collapse ='')

    if (i < 100){j = j + 5*stepfactor}
    else {j = j + 50*stepfactor}
    #else if (n < 1000){j = j + 10*stepfactor}
  }

  data
}

#lapply(gen_data(102), nchar)
```

The timing function

```
time_decode <- function(func, #the Decoder function used
                        n,     #Num of data points (can get a str of len 3*n to 9*n long)
                        test_data #the data we use
                        )
  {

  run_time = matrix(data = rep(-1, n*3), ncol=n, nrow=3)

  for (i in 1:length(test_data)){
    #ttdd <<- test_data[i]
    start_time = Sys.time()
    run_time[1, i] = func(test_data[i])       #Results
    run_time[2, i] = Sys.time() - start_time  #Run Time
    run_time[3, i] = test_data[[i]]           #Test data
  }

  run_time
}
```

```
#time_decode(URL_vec, 10, list('a%20b'))
# run_time = time_decode(URLdecode_v2, 1000)
# plot(lapply(run_time[3,], nchar), run_time[2,])
```

Compare results with URLdecode results We do so by using the results from our URLdecode runtime to avoid running it twice

```
test_results <- function(results_mtrx, truth_mtrx){
  #truth_val = lapply(results_mtrx[3,], URLdecode)

  if (sum(truth_mtrx[1, ] == results_mtrx[1, ]) != length(truth_mtrx[1, ])){
    TRUTH_VAL <<- trtuth_mtrx[1,]
    PRED_VAL <<- results_mtrx[1,]
    warning('Values do not match! Saving variables as TRUTH_VAL & PRED_VAL')
  }
}

#test_results(time_decode(URLdecode_v2, 2000))
```

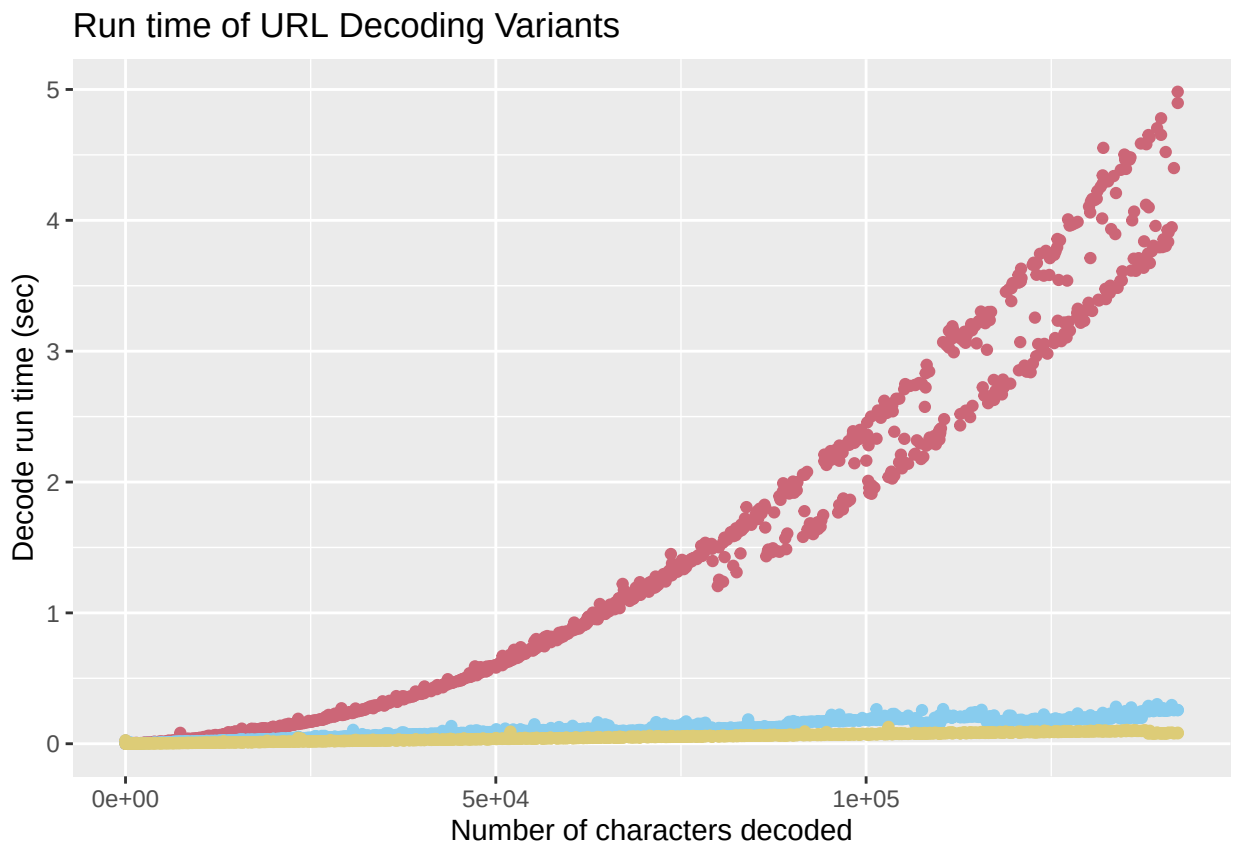
Main Control Unit

```
main <- function(n){

  runtimes <- matrix(rep(-1, n*4), nrow = 4, ncol = n)
  #runtimes[4, ] <- c(1:n)

  #Generate the data
  data2test = gen_data(n)
  runtimes[4, ] = as.numeric(lapply(data2test, nchar))

  #print(runtimes)

  #URLdecode basetime
  base_de <- time_decode(URLdecode, n, data2test)
      #We don't test results because URLdecode is our 'Truth Value(s)'
  runtimes[1, ] <- as.numeric(base_de[2, ])

  #URLdecode_v2 values
  redo_de <- time_decode(URLdecode_v2, n, data2test)
  test_results(redo_de, base_de)
  runtimes[2, ] <- as.numeric(redo_de[2, ])

  #URL_vec values
  vect_de <- time_decode(URL_vec, n, data2test)
  test_results(vect_de, base_de)
  runtimes[3, ] <- as.numeric(vect_de[2, ])

  return(runtimes)
}

t_data = main(1000)

t_data = as.data.frame(t(t_data)) #= as.numeric(t_data)
```

```
colnames(t_data) <- c('base_t', 'v2_t', 'vector_t', 'nchar')
head(t_data)
```

```
##         base_t         v2_t     vector_t nchar
## 1 2.370811e-02 2.388597e-02 2.605796e-02     1
## 2 8.702278e-05 8.296967e-05 6.794930e-05    24
## 3 9.012222e-05 8.511543e-05 7.796288e-05    36
## 4 1.358986e-04 1.311302e-04 8.487701e-05    62
## 5 1.389980e-04 1.330376e-04 9.083748e-05    67
## 6 1.630783e-04 1.568794e-04 1.020432e-04    82
```

```
#[3] color blind palette
ggplot(data = t_data, aes(x = nchar, y = base_t)) +
  geom_point(color = "#CC6677") + #UrlDecode
  geom_point(aes(y = v2_t), color = "#88CCEE") + #URLdecode_v2
  geom_point(aes(y = vector_t), color = "#DDCC77") + #URL_vec
  labs(title = 'Run time of URL Decoding Variants',
       x= 'Number of characters decoded',
       y= 'Decode run time (sec)',
       color = "Function") +
  scale_color_manual(name = "Function",
                     breaks = c("URLdecode", "URLdecode_v2", "URL_vec"),
                     values = c("#CC6677", "#88CCEE", "#DDCC77")) #[5] Adding legend
```



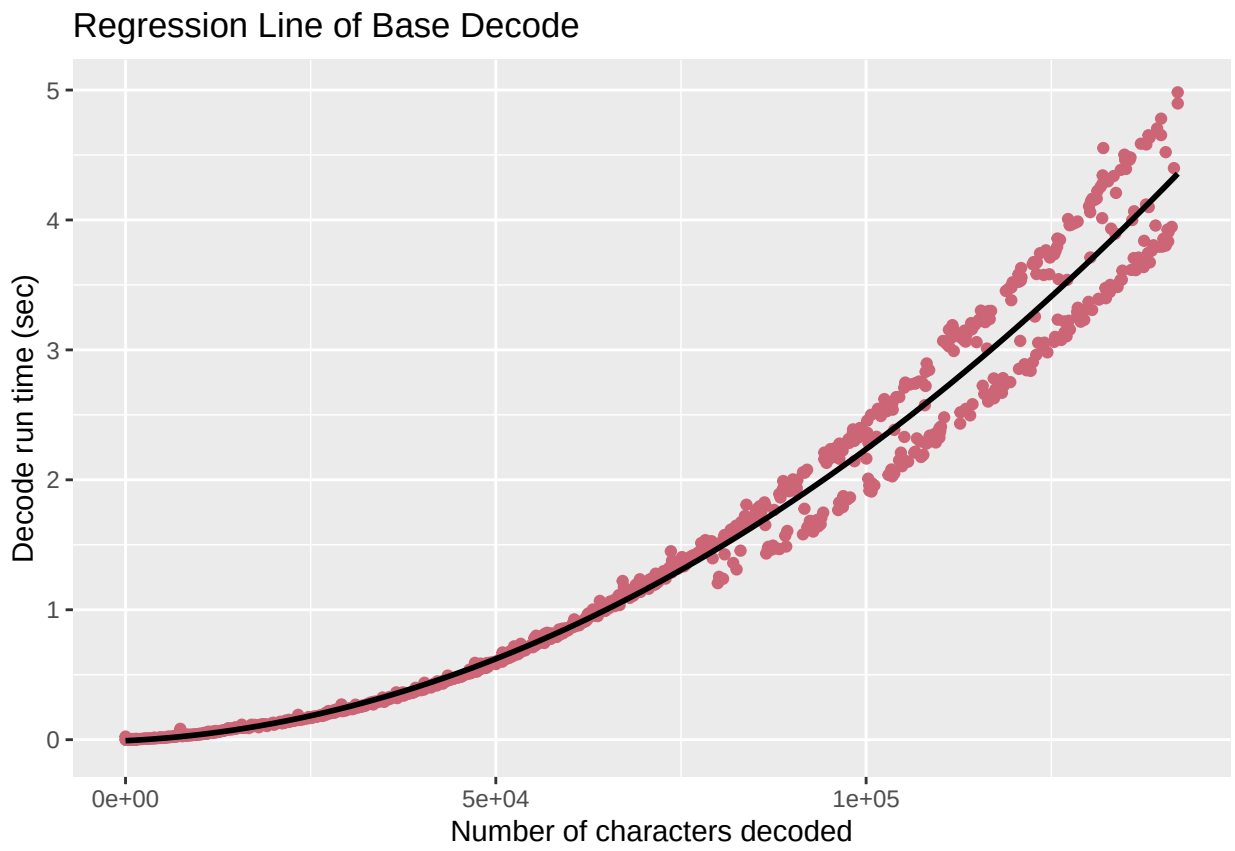
Regression / Prediction:

```r
#Base Model
t_data$nchar_sqr = t_data$nchar ^ 2
base_model_lm <- lm(base_t ~ nchar + nchar_sqr, data=t_data)
summary(base_model_lm)
```

```
##
## Call:
## lm(formula = base_t ~ nchar + nchar_sqr, data = t_data)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.46401 -0.02689  0.00555  0.04852  0.76766
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -8.129e-03  1.405e-02  -0.579    0.563
## nchar        2.769e-06  5.056e-07   5.477 5.47e-08 ***
## nchar_sqr    1.967e-10  3.678e-12  53.475  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.192 on 997 degrees of freedom
## Multiple R-squared:  0.9789, Adjusted R-squared:  0.9788
## F-statistic: 2.309e+04 on 2 and 997 DF,  p-value: < 2.2e-16
```

```r
#[6][7] quadratic curve
ggplot(data = base_model_lm, aes(x = nchar)) +
     geom_point(aes(y = base_t), color = "#CC6677") +
     stat_smooth(aes(y = base_t),
                 method = "lm",
                 formula = y ~ poly(x, 2),
                 size = 1,
                 color = 1) +
    labs(title = 'Regression Line of Base Decode',
       x= 'Number of characters decoded',
       y= 'Decode run time (sec)')
```

```
## Warning: Using 'size' aesthetic for lines was deprecated in ggplot2 3.4.0.
## i Please use 'linewidth' instead.
## This warning is displayed once every 8 hours.
## Call 'lifecycle::last_lifecycle_warnings()' to see where this warning was
## generated.
```



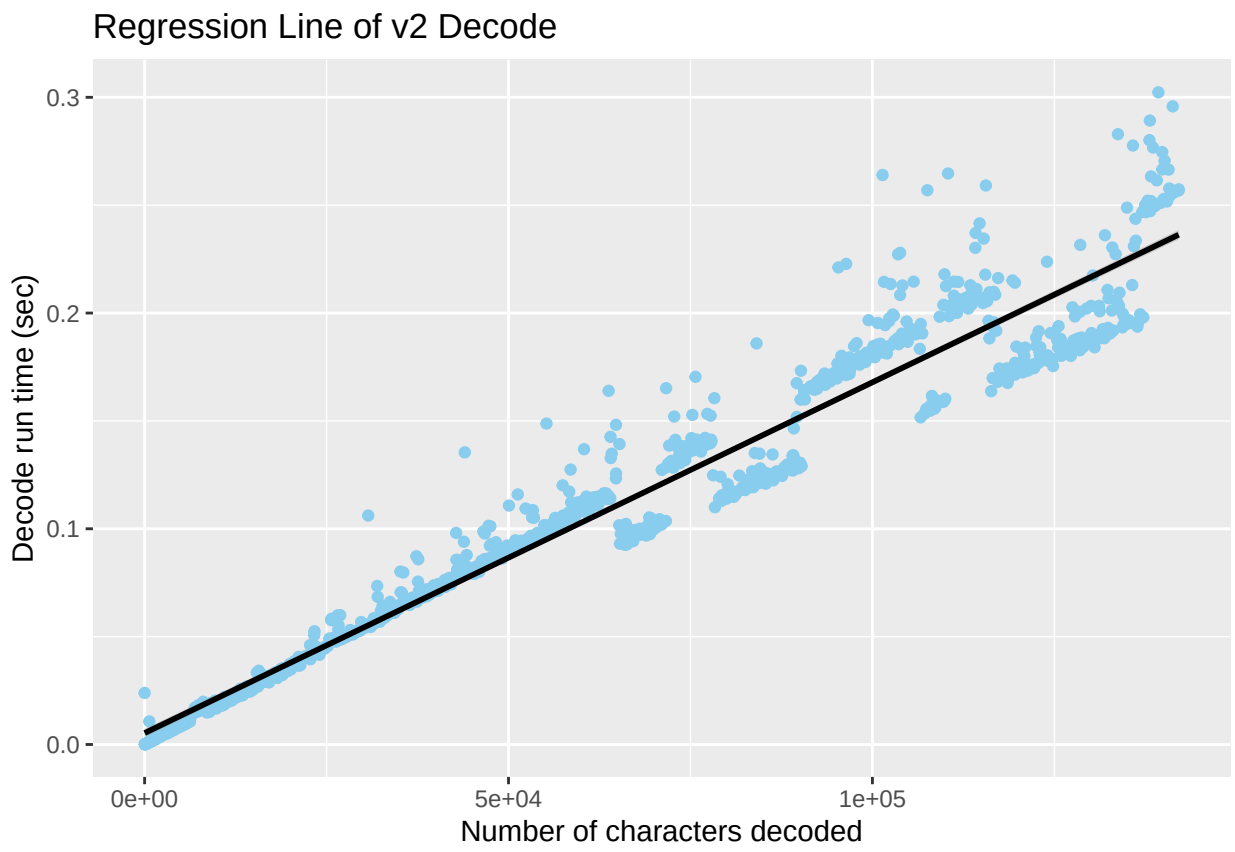

```
#v2 Model
v2_lm <- lm(v2_t ~ nchar, data = t_data)
summary(v2_lm)
```

```
##
## Call:
## lm(formula = v2_t ~ nchar, data = t_data)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.033416 -0.005286 -0.000284  0.009409  0.093842
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 5.305e-03  9.845e-04   5.389 8.86e-08 ***
## nchar       1.626e-06  1.260e-08 129.074  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.01749 on 998 degrees of freedom
## Multiple R-squared:  0.9435, Adjusted R-squared:  0.9434
## F-statistic: 1.666e+04 on 1 and 998 DF,  p-value: < 2.2e-16
```

```
ggplot(data = v2_lm$model, aes(x = nchar, y = v2_t)) +
    geom_point(color = "#88CCEE") +
```

```
    stat_smooth(method = "lm", size = 1, color = 1) +
    labs(title = 'Regression Line of v2 Decode',
       x= 'Number of characters decoded',
       y= 'Decode run time (sec)')
```

```
## `geom_smooth()` using formula = 'y ~ x'
```



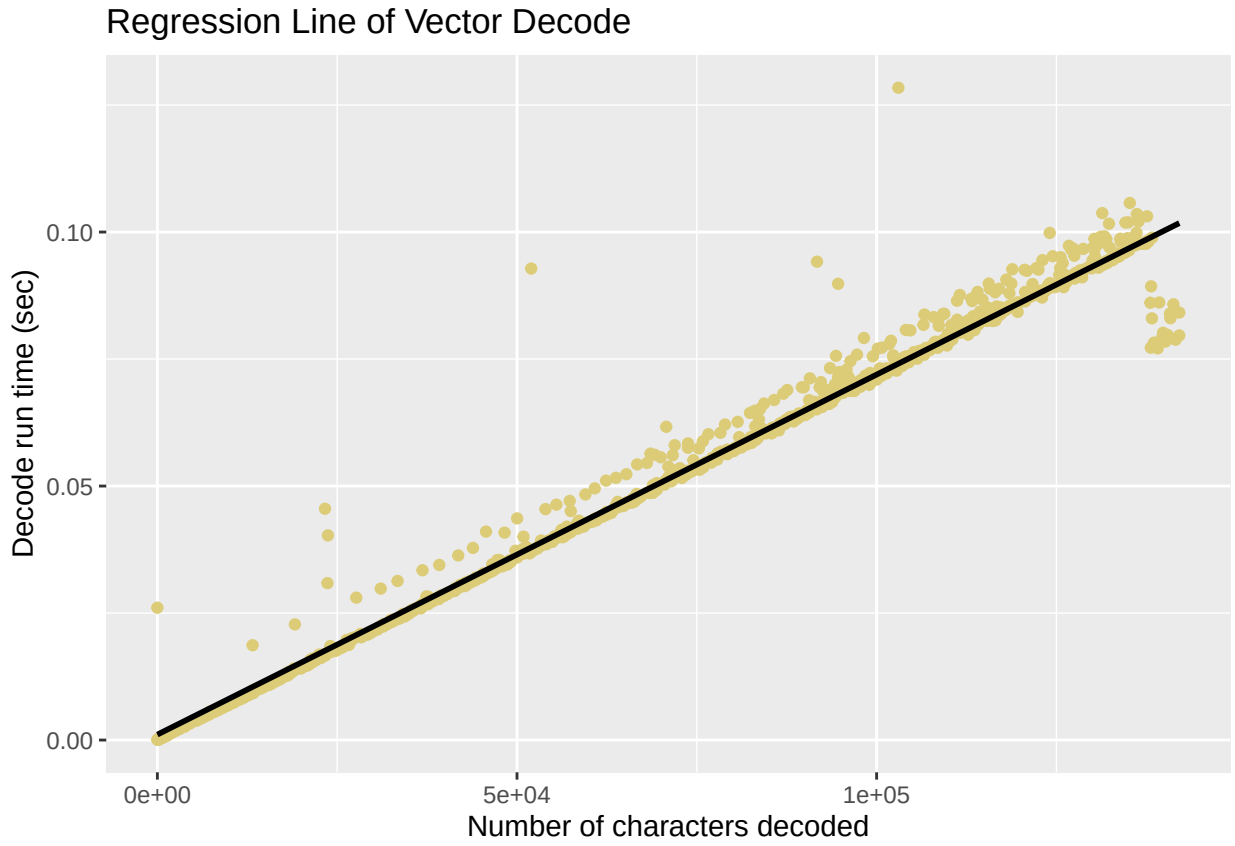

```
#vector Model
vec_lm <- lm(vector_t ~ nchar, data = t_data)
summary(vec_lm)
```

```
##
## Call:
## lm(formula = vector_t ~ nchar, data = t_data)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.022585 -0.000997 -0.000666  0.000197  0.054901
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 1.079e-03  2.696e-04   4.002 6.74e-05 ***
## nchar       7.086e-07  3.450e-09 205.392  < 2e-16 ***
## ---
```

```
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.00479 on 998 degrees of freedom
## Multiple R-squared:  0.9769, Adjusted R-squared:  0.9769
## F-statistic: 4.219e+04 on 1 and 998 DF,  p-value: < 2.2e-16
```

```
ggplot(data = vec_lm, aes(x = nchar, y = vector_t)) +
    geom_point(color = "#DDCC77") +
    stat_smooth(method = "lm", size = 1, color = 1) +
    labs(title = 'Regression Line of Vector Decode',
       x= 'Number of characters decoded',
       y= 'Decode run time (sec)')
```

```
## `geom_smooth()` using formula = 'y ~ x'
```



Mega String Time Comparing:

```
#full_txt = read.table('E:\\College\\UC Davis\\STA141B\\HW5\\PercentEncodedString.txt')
#mini_txt = read.table('E:\\College\\UC Davis\\STA141B\\HW5\\PercentEncoded_mini.txt')

#Making our Regression Prediciton
nchar(full_txt) #591977 characters
```

```
##     V1
## 591977
```

```
t_titles = c('URLdecode', 'URLdecode_v2', 'URL_vec')
decode_fcts = c(URLdecode, URLdecode_v2, URL_vec)

pred_times = data.frame(matrix(data = NA, nrow = 3, ncol = 3))
colnames(pred_times) <- c("Model", "Predicted Run Times (sec)", "Real Run Times (sec)")
for (i in 1:3){pred_times[i,1] = t_titles[i]}
pred_times[1,2] = predict(base_model_lm, data.frame(nchar = 591977, nchar_sqr = 591977^2 ))[[1]]
pred_times[2,2] = predict(v2_lm, data.frame(nchar = 591977), nchar_sqr = 591977^2)[[1]]
pred_times[3,2] = predict(vec_lm, data.frame(nchar = 591977))[[1]]

for (i in 1:3){
  t0 = Sys.time()
  decode_fcts[[i]](full_txt)
  pred_times[i, 3] = difftime(Sys.time(), t0, units = "secs")[[1]]
}

kable(pred_times)
```

Model	Predicted Run Times (sec)	Real Run Times (sec)
URLdecode	70.5604513	299.239938
URLdecode_v2	0.9677605	0.440223
URL_vec	0.4205448	0.869216

[1] https://statisticsglobe.com/count-occurrence-of-certain-character-in-string-in-r

[2] https://statisticsglobe.com/apply-functions-in-r/#example-6-mapply-function

[3] https://stackoverflow.com/questions/57153428/r-plot-color-combinations-that-are-colorblind-accessible

[4] https://www.statology.org/r-create-vector-of-zeros/

[5] https://stackoverflow.com/questions/23635662/editing-legend-text-labels-in-ggplot

[6] https://stackoverflow.com/questions/42764028/fitting-a-quadratic-curve-in-ggplot

[7] https://thomasadventure.blog/posts/ggplot-regression-line/