Assignment5

Filip Wilhelm Sjostrand

2023-06-15

1 Introduction

This report presents the process of optimizing the URL decoding function in R. We start by an explanation of the modifications made to the original function. We then test the modified functions and debug any issues that arise. Further, we create a model to predict the run time of the function based on the string length and visualize the results. Finally, we conclude our observations and discuss any possible pitfalls.

```r
library(rvest)
library(httr)
library(ggpubr)
theme_set(theme_pubr())
library(tidyverse)
source("funs5.R")
source("debugger.R")
```

2 Data Acquisition and Preprocessing

We begin by acquiring the UTF-8 percent-encoding table from the W3Schools website. This table is essential for decoding percent-encoded strings. We parse the website, select the table using an appropriate XPath, and extract the relevant columns. The resulting data frame, utf_df, is saved for future use.

```r
# Get UTF-8 Percent-Encoding ----------
response <- GET("https://www.w3schools.com/tags/ref_urlencode.ASP")
parsed_page <- read_html(response)

utf_df <- parsed_page %>%
  html_elements(xpath = '//*[@id="main"]/table[1]') %>%
  html_table() %>%
  as.data.frame() %>%
  select(-From.Windows.1252)

saveRDS(utf_df, "utf_df.rds")
```

```r
# Get the Simulations ----------
sample_strings <- readRDS("sample_strings.rds")
URL_real <- readLines("PercentEncodedString.txt")
```

```
## Warning in readLines("PercentEncodedString.txt"): incomplete final line found
## on 'PercentEncodedString.txt'
```

3 Function Optimization

3.1 Preallocation

The original URL decoding function was optimized for improved performance, particularly for larger inputs. The initial function was expanding the output vector each time it encountered a character to decode, which is inefficient for larger data. To address this, we modified the function to preallocate a large output vector at the start, and then fill it as it processed the URL. An additional step was added to trim any unused space in the vector at the end, ensuring the output is correctly formatted. This optimization resulted in a more efficient function that performs faster and uses less memory.

3.2 vectorized

In the vectorized version of the URL decoding function, we've broken down the process into several smaller, more manageable functions, each performing a specific task. The `find_percent()` function identifies all the percent-encoded characters in the input string. The `index_percent()` function locates the start and end indices of these encoded characters. The `convert_percent()` function then converts these encoded characters into their equivalent characters using a lookup table. These three functions are then combined in the `merge_outputs()` function to create a comprehensive matrix of all the relevant information needed for string manipulation.

The `replace_percent()` function uses this matrix to replace all the percent-encoded characters in the original string. Instead of iterating over each encoded character one by one, it uses the `stringi::stri_replace_all_fixed()` function to replace all matches at once, making it significantly faster and more efficient. This function is then applied to all URLs in a list using the `sapply()` function in `URLdecode_vectorized()`, allowing for efficient processing of multiple URLs at once.

This vectorized approach is more efficient than the original function, particularly for larger inputs, as it avoids the need for iterative loops and takes advantage of R's vectorized operations for faster computation.

4 Function Testing and Debugging

By first comparing each output for each UTF encoding, by using `mismatcher()` from `debugger.R`, we can see any discrepancies in the transformations of characters. We can see a clear pattern in the mismatches: the functions based upon `utils::URLdecode()` produces another output than teh vectorized function that gets its transformations from the scraped table `utf_df`.

```r
res1 <- mismatcher(utf_df)
res1
```

```
##       UTF       table     original    preallocated vectorized
##  [1,] "%20"     "space"   " "         " "          "space"
##  [2,] "%7F"     ""        "\177"      "\177"       ""
##  [3,] "%81"     "\u0081" "\x81"      "\x81"       "\u0081"
##  [4,] "%C5%8D"  "\u008d" "ō"         "ō"          "\u008d"
##  [5,] "%8F"     "\u008f" "\x8f"      "\x8f"       "\u008f"
##  [6,] "%E2%84"  " "      "\xe2\x84" "\xe2\x84"   " "
##  [7,] "%E2%80"  ">"       "\xe2\x80" "\xe2\x80"   ">"
##  [8,] "%9D"     "\u009d" "\x9d"      "\x9d"       "\u009d"
##  [9,] "%C2%A0"  ""        " "         " "          ""
```

A simple approach to this is to write a function, `remapper()`, that takes in `utf_df` and iterates through the whole data frame. It should take each element of `From.UTF.8`, give it to `URLdecode()`, compate to the output given in `Character`, if different, replace it with `URLdecode()` output. Then, when the vectorized version is called, we should get the same output as the original function.

```r
utf_df <- remapper(utf_df)
saveRDS(utf_df, "utf_df.rds")
```

The function clearly reduced our mismatches. Left there seems to be a consistent output among the functions. We should understand why R call them inequality.

```r
res2 <- mismatcher(utf_df)
res2
```

```
##      UTF      table      original   preallocated vectorized
## [1,] "%81"    "\x81"     "\x81"     "\x81"       "\x81"
## [2,] "%8F"    "\x8f"     "\x8f"     "\x8f"       "\x8f"
## [3,] "%E2%84" "\xe2\x84" "\xe2\x84" "\xe2\x84"   "\xe2\x84"
## [4,] "%E2%80" "\xe2\x80" "\xe2\x80" "\xe2\x80"   "\xe2\x80"
## [5,] "%9D"    "\x9d"     "\x9d"     "\x9d"       "\x9d"
```

For the sake of the assignment, we shall see if any of the remaingin mismatching characters appear in our goal string, the `URL_real`. We begin by filtering `utf_df` on `res2` such that we may pass it on to `find_percent()`. Since we cannot find any of the non-matching characters in our desired URL, we shall proceed with the assignment as they will not interfere with the result.

```r
subset_df <- utf_df %>% filter(utf_df$From.UTF.8 %in% res2[,1])
find_percent(URL_real, subset_df)
```

```
##      Match
```

Remove the irrelevant characters. We have no mismatches now

```r
utf_df <- utf_df %>% filter(!(From.UTF.8 %in% subset_df$From.UTF.8))
saveRDS(utf_df, "utf_df.rds")
mismatcher(utf_df)
```

```
##      UTF table original preallocated vectorized
```

The `tester.R` script is a comprehensive testing suite for URL decoding functions in R. The script defines a set of test inputs and expected outputs. These include both known inputs and outputs, as well as randomly generated inputs. The known inputs include specific URL-encoded strings and a large input string created by repeating a specific URL-encoded string 10,000 times.

The script defines three testing functions: `test_known_outputs()`, `test_random_comparison()`, and `test_large_inputs()`. The `test_known_outputs()` function tests whether a given decoding function produces the expected output for each known input. The `test_random_comparison()` function tests whether the output of a given decoding function matches the output of the original `URLdecode()` function for each random input. The `test_large_inputs()` function tests whether a given decoding function can correctly decode the large input string.

Finally, the script runs these testing functions on our two different URL decoding functions: `URLdecode_preallocated()` and `URLdecode_vectorized()`. Since running the script throws no error, we shall confirm that the functions are performing correctly.

```r
source("tester.R")
```

5 Modeling

```r
# Loading run times ----------
r_time <- readRDS("r_time.rds")
r_time_prealoc <- readRDS("r_time_prealoc.rds")
r_time_vector <- readRDS("r_time_vector.rds")

true_original <- readRDS("r_time_real.rds")
true_preallocated <- readRDS("r_time_prealoc_real.rds")
true_vectorized <- readRDS("r_time_vector_real.rds")

#
real_URL <- readLines("PercentEncodedString.txt")
```

```
## Warning in readLines("PercentEncodedString.txt"): incomplete final line found
## on 'PercentEncodedString.txt'
```

The chosen model to predict the run time of `utils::URLdecode()` as a function of string length is a quadratic model, as indicated by the `I(num_char^2)` term. This suggests that the relationship between run time and string length is not linear, but rather increases at an accelerating rate as string length increases. The statistical evidence strongly supports this model. The coefficient for the squared term of string length is highly significant, indicating a substantial effect on the run time. The residuals of the model are relatively small, suggesting accurate predictions. The model explains all of the variability in the run time, as indicated by the R-squared values of .99. The F-statistic is extremely high, further supporting the model's validity.

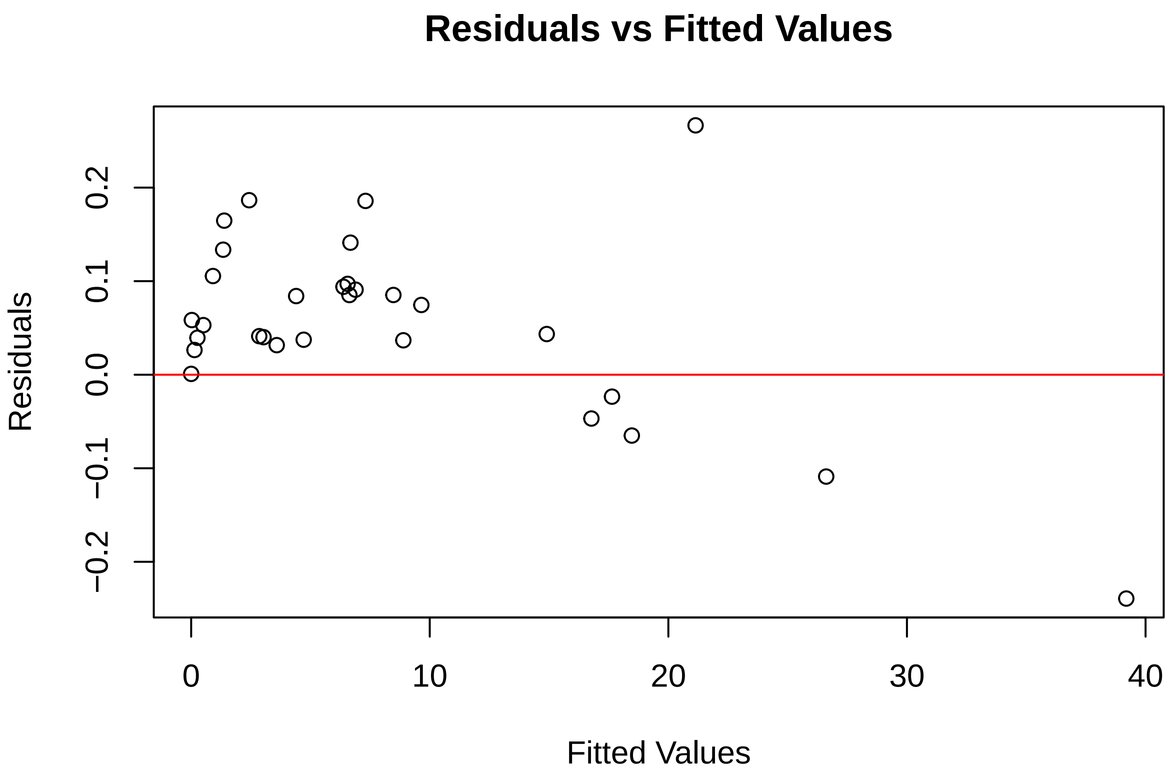
Interestingly, the model does not include an intercept term. This is because the intercept was found to be statistically insignificant, suggesting that when the string length is zero, the run time is also zero. This makes sense in the context of the `utils::URLdecode()` function, as there would be nothing to decode in an empty string, and thus the run time should be zero.

```r
string_runtime <-
  sample_strings %>%
  cbind(r_time) %>%
  select(-string)

model <- lm(run_time ~ I(num_char^2) - 1, data = string_runtime)
summary(model)
```

```
##
## Call:
## lm(formula = run_time ~ I(num_char^2) - 1, data = string_runtime)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.23929  0.03289  0.05570  0.09631  0.26654
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## I(num_char^2) 5.58e-10   9.37e-13   595.6   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.112 on 29 degrees of freedom
## Multiple R-squared:  0.9999, Adjusted R-squared:  0.9999
## F-statistic: 3.547e+05 on 1 and 29 DF,  p-value: < 2.2e-16
```

To further validate this model, we can test the assumptions of linear regression. These assumptions include linearity, independence, homoscedasticity (constant variance), and normality of residuals. The Durbin-Watson test shows no significant autocorrelation in the residuals, which is a good sign. The residuals of the model do not appears to barely follow a normal distribution, as indicated by the Shapiro-Wilk test and the QQ plot. This almost violation of the normality assumption could affect the reliability of the model's predictions. Similarly, there appears to be some pattern in the residual plot. Therefore, while the model seems to have a good fit according to some metrics, the violation of the normality assumption suggests that caution should be exercised when using this model for prediction.

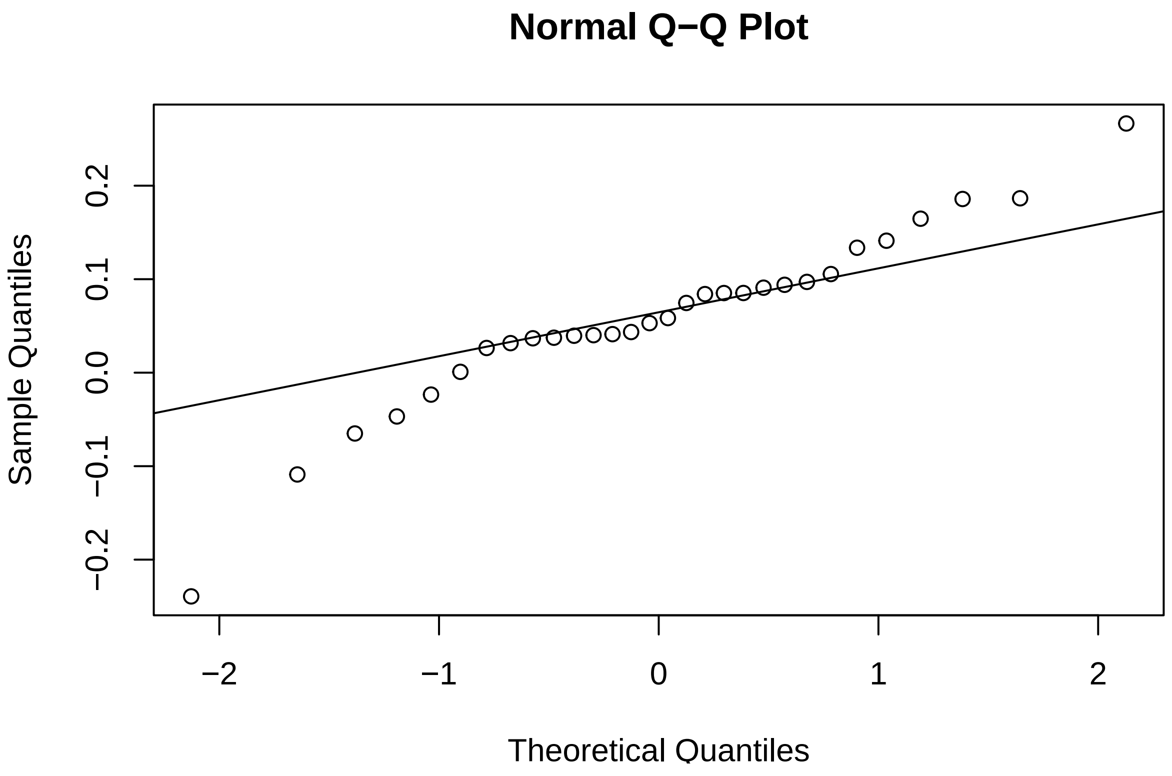
```r
residuals <- resid(model)

plot(predict(model), residuals,
     xlab = "Fitted Values", ylab = "Residuals",
     main = "Residuals vs Fitted Values")
abline(h = 0, col = "red")
```



```r
car::durbinWatsonTest(model)
```

```
##  lag Autocorrelation D-W Statistic p-value
##    1         0.33207      1.178365    0.16
##  Alternative hypothesis: rho != 0
```

```
qqnorm(residuals)
qqline(residuals)
```



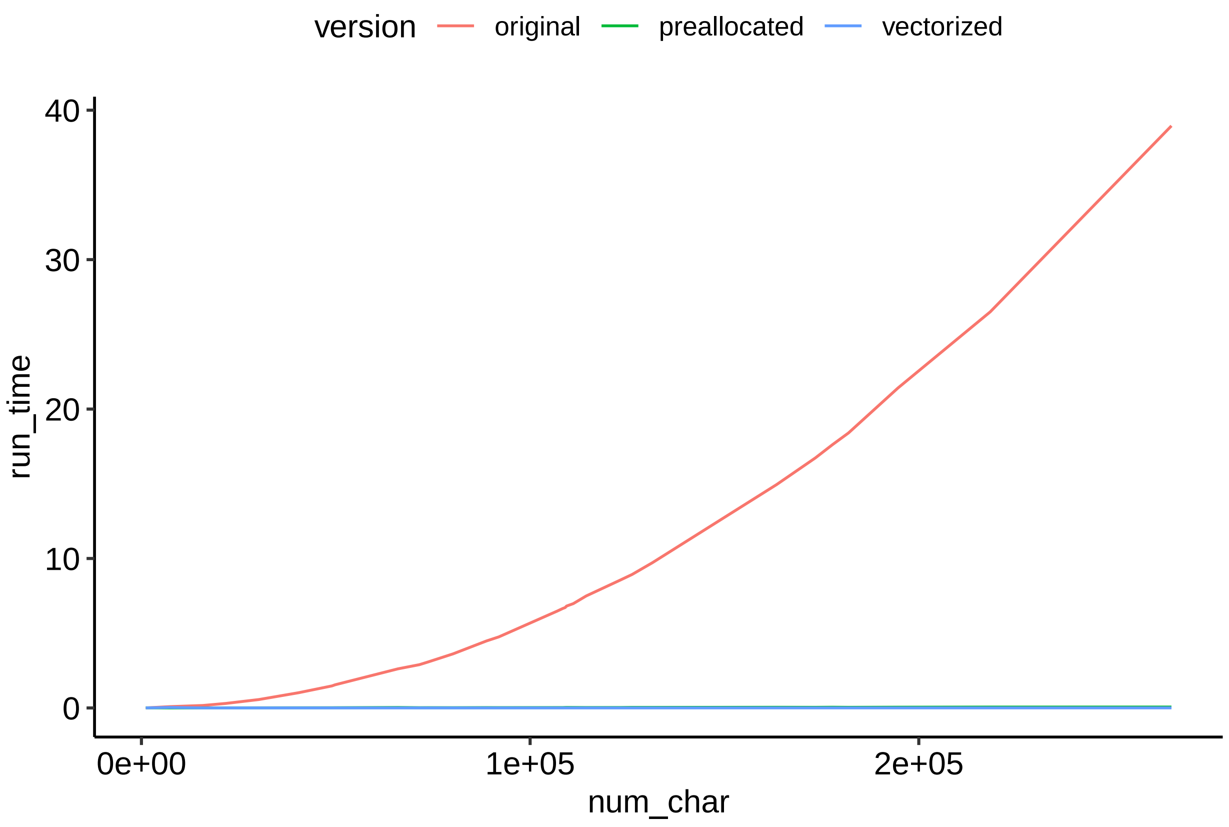
```
shapiro.test(residuals)
```

```
##
##  Shapiro-Wilk normality test
##
## data:  residuals
## W = 0.93994, p-value = 0.09065
```

6 Data Visualization

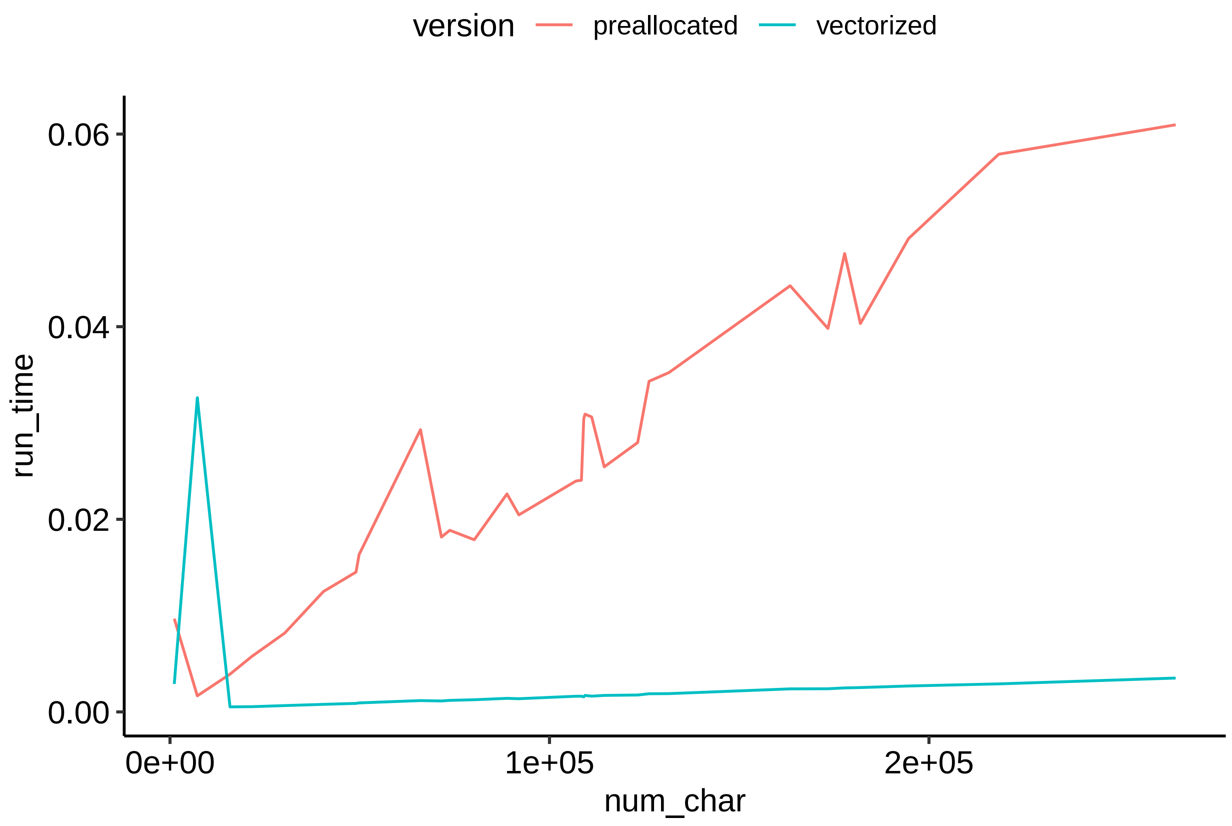
In the first plot, all three versions are compared. It's evident that the modified versions (preallocated and vectorized) outperform the original function, as their run times are significantly lower. This demonstrates the effectiveness of the modifications in improving the function's performance.

```
full_df <- data.frame(
  num_char = sample_strings$num_char,
  "original" = string_runtime$run_time,
  "preallocated" = r_time_prealoc[[1]],
  "vectorized" = r_time_vector[[1]]
) %>%
  pivot_longer(
    cols = c(original, preallocated, vectorized),
    names_to = "version",
    values_to = "run_time"
  )

full_df %>%
  ggplot() +
  aes(x = num_char, y = run_time, colour = version) +
  geom_line() +
  scale_color_hue(direction = 1)
```



The second plot focuses on the comparison between the two modified versions: preallocated and vectorized. It was initially expected that the vectorized version would perform the best due to the inherent efficiency of vectorized operations in R. However, the plot reveals that the preallocated version actually has the shortest run time, which was a surprising result.

```
full_df %>%
  filter(version != "original") %>%
  ggplot() +
  aes(x = num_char, y = run_time, colour = version) +
  geom_line() +
  scale_color_hue(direction = 1)
```

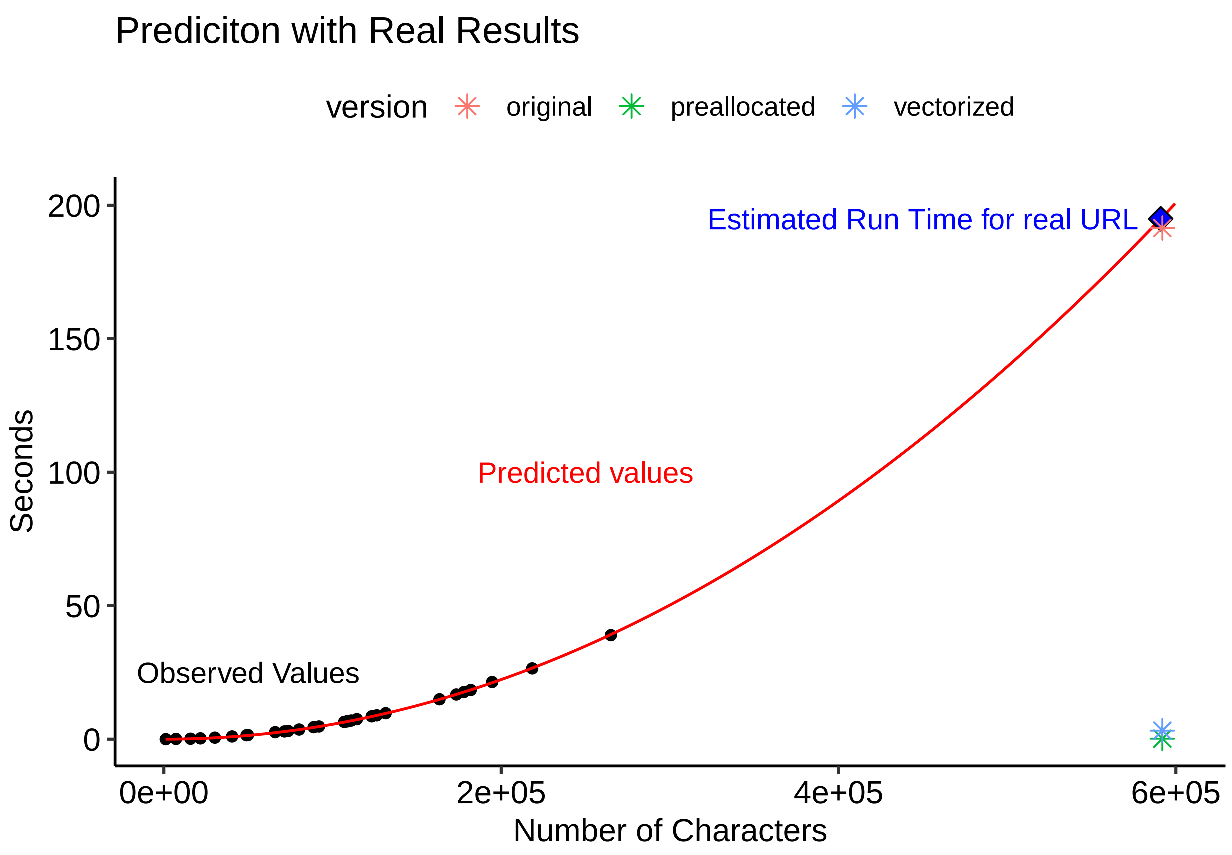


The final series of plots involve a prediction model for the run time of the original function based on the number of characters. The model's predictions are plotted alongside the observed values, providing a visual comparison of the model's accuracy. The model's predictions are quite close to the actual results, indicating that it is a good fit for the data. However, the plot also reveals that the preallocated version of the function outperforms the original function even at high numbers of characters, further emphasizing its superior performance.

The asterisks on the plots represent the run times for a real URL, providing a practical example of the function's performance. These points align well with the model's predictions and the observed values, further validating the model and the effectiveness of the function modifications.

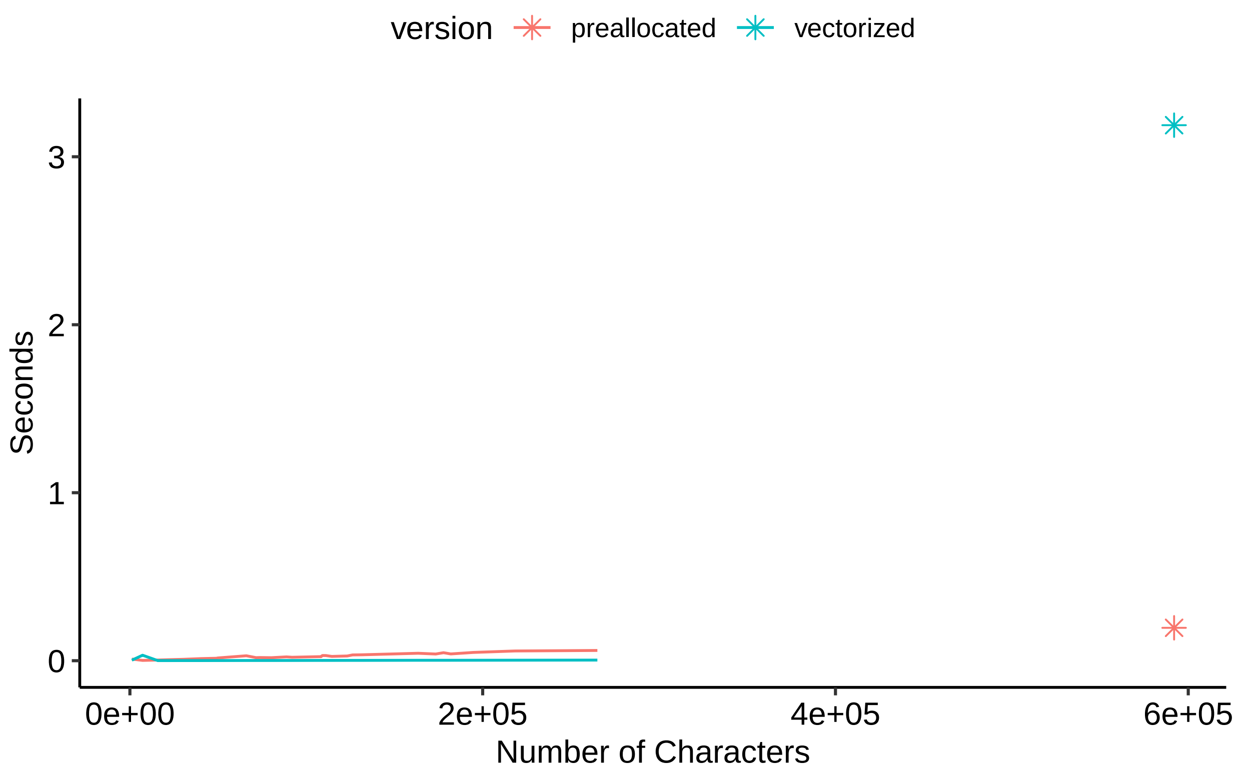
```
estimate_points <- data.frame(num_char = c(seq(
  1000,
  600000,
  by = 1500
)))
predictions <- predict(model, newdata = estimate_points)
predicted_df <- data.frame(run_time = predictions, estimate_points)
goal_point <- data.frame(
  run_time = predict(model, newdata = data.frame(num_char = 591000)),
  num_char = 591000
)

true_char <- nchar(real_URL)

real_df <- data.frame(
  num_char = true_char,
  "original" = true_original[[1]],
  "preallocated" = true_preallocated[[1]],
```

```r
  "vectorized" = true_vectorized[[1]]
) %>%
  pivot_longer(
    cols = c(original, preallocated, vectorized),
    names_to = "version",
    values_to = "run_time"
  )

ggplot() +
  geom_point(
    aes(x = num_char, y = run_time),
    data = string_runtime
  ) +
  geom_line(
    aes(x = num_char, y = run_time),
    data = predicted_df,
    colour = "red"
  ) +
  geom_point(
    aes(x = num_char, y = run_time),
    data = goal_point,
    fill = "blue",
    shape = 23,
    size = 3
  ) +
  annotate(
    geom = "text",
    x = 250000,
    y = 100,
    label = "Predicted values",
    colour = "red"
  ) +
  annotate(
    geom = "text",
    x = 450000,
    y = goal_point[[1]],
    label = "Estimated Run Time for real URL",
    colour = "blue"
  ) +
  annotate(
    geom = "text",
    x = 50000,
    y = 25,
    label = "Observed Values"
  ) +
  labs(title = "prediction of run time for utils::URLdecode()") +
  ylab("Seconds") +
  xlab("Number of Characters") +
  geom_point(
    aes(x = num_char, y = run_time, colour = version),
    data = real_df,
    shape = "asterisk",
    size = 2.6
  ) +
  scale_color_hue(direction = 1) +
  labs(title = "Prediciton with Real Results")
```



```
full_df %>%
  filter(version != "original") %>%
  ggplot() +
  aes(x = num_char, y = run_time, colour = version) +
  geom_line() +
  geom_point(
    aes(x = num_char, y = run_time, colour = version),
    data = real_df %>% filter(version != "original"),
    shape = "asterisk",
    size = 2.6
  ) +
  scale_color_hue(direction = 1) +
  labs(title = "observed with Final Results") +
  ylab("Seconds") +
  xlab("Number of Characters")
```

observed with Final Results



7 Discussion

In summary, we saw suggestions that implementing preallocation and vectorization can speed up a function call significantly. From our generated strings, we anticipated a slightly quicker version using preallocation plus vectorization than just simply preallocation. However, from implementing the functions on a real world data, we observed the opposite.

The preallocated version uses a while loop to iterate through each character in the URL string. This approach is efficient when the URL contains a high proportion of encoded characters, as it processes each character only once.

On the other hand, the vectorized version uses regular expressions to identify and replace encoded characters. This approach is highly efficient when the URL contains a low proportion of encoded characters, as it can replace all occurrences of an encoded character at once. However, it may be less efficient when the URL contains a high proportion of encoded characters, as it needs to perform multiple replacements.

```r
n_utf_real <- find_percent(real_URL, utf_df)
real_ratio <- length(n_utf_real)/nchar(real_URL)

big_sample_string <- tail(sample_strings, 1)$string
n_utf_sample <- find_percent(big_sample_string, utf_df)
sample_ration <- length(n_utf_sample)/nchar(big_sample_string)

real_ratio/sample_ration
```

```
## [1] 2635.768
```

To confirm this hypothesis, we have gathered the proportions of a generated string and the real URL. Below we can confirm that the real URL has a significantly larger proportion of UTF encoding (2636 times more).

```r
test <- paste0(
  sample(utf_df$From.UTF.8, 250000, replace = TRUE),
  collapse = ""
)

res1 <- run_time(test, FUN = URLdecode_preallocated)
res2 <- run_time(test, FUN = URLdecode_vectorized, utf_df)

res2/res1
```

```
##   run_time
## 1 48.25072
```

Similarily, If we tried to generate a string consisting only of UTF encodings and compared with would expect the preallocated function to perform better. We can observe that for a random string consisting of 250000 UTF encodings, the preallocated version performed 48 times faster than the vectorized version.

Hence, for future refernece, an approach to vectorized function would likely to be to modify the existent function rather than implementing regular expressions.