Stats 199 Final Report - Webscraping and Dashboarding with Grailed.com Data

Naren Akurati

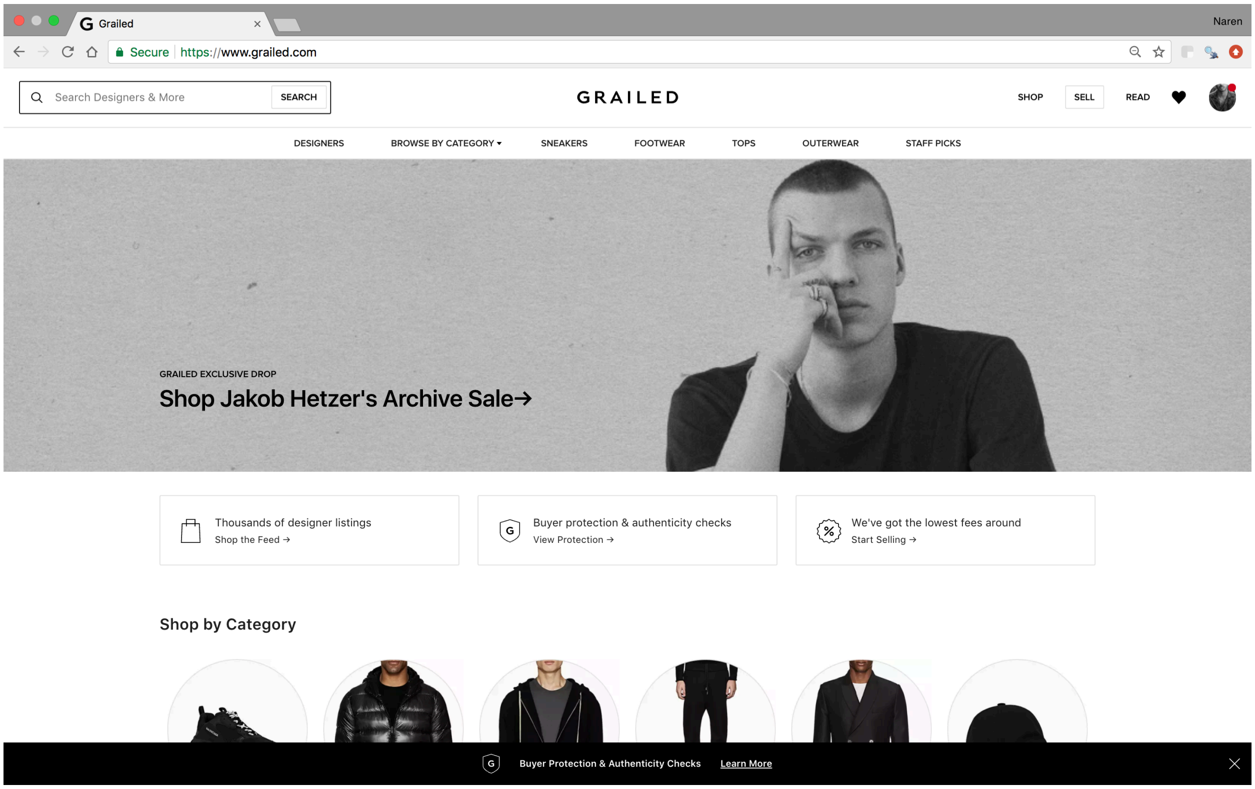
3/22/2019

Contents

1 Intro/Abstract

1.1 What I was trying to achieve

Grailed.com is a marketplace for high fashion items. It runs roughly 1,000 transactions every 24 hours ranging from rare/vintage Helmut Lang pieces to the latest Supreme drops. My goal was to harness this data and create a potential solution for users of the site that would benefit them in their shopping experience and offer them further insight into the marketplace.



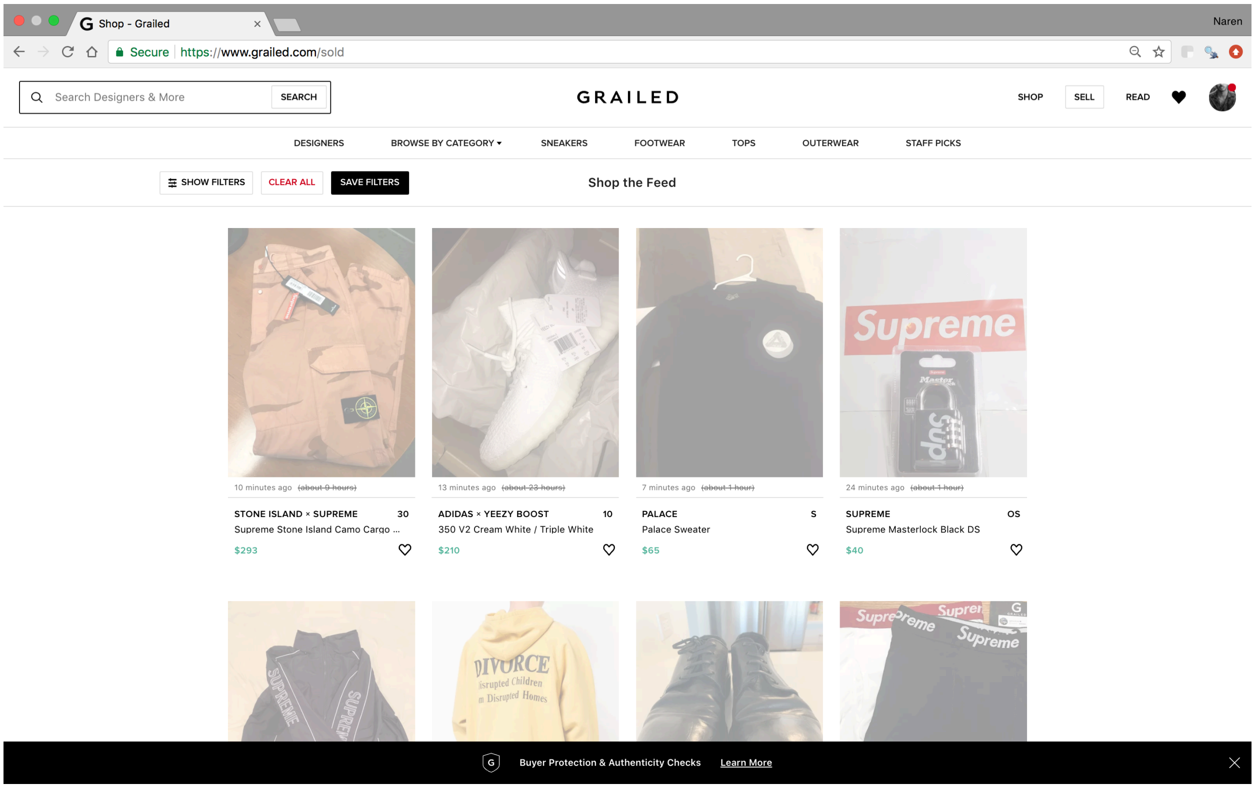
1.2 What I ended with

I ended with a live Shiny dashboard that automatically pulls and updates itself with new transactions. It displays the top 10 brands at any time and their respective sales counts.

2 Web Scraping

2.1 Rvest - Slow Method

My first way to get the data was to render the *Grailed.com/sold* page and scrape 80 items at a time.



10 minutes ago ~~(about 9 hours)~~
STONE ISLAND × SUPREME 30
Supreme Stone Island Camo Cargo ...
$293

13 minutes ago ~~(about 23 hours)~~
ADIDAS × YEEZY BOOST 10
350 V2 Cream White / Triple White
$210

7 minutes ago ~~(about 1 hour)~~
PALACE S
Palace Sweater
$65

24 minutes ago ~~(about 1 hour)~~
SUPREME OS
Supreme Masterlock Black DS
$40

Below is the preliminary function using PhantomJS to render each grailed.com/sold page in the background and pull 80 items at a time. To pull 80 items, the function takes roughly **1.7 minutes** to run. The reason the time is so lengthy, is because this is almost equivalent to loading the Grailed.com/sold page yourself on an internet browser. This method includes regular expressions and searching through page elements for key words to pull data. The relavant code is included below.

```
load_data <- function(){

  #create temp dataframe
  temp_frame <- setNames(data.frame(matrix(ncol = 6, nrow = 80)), c("title", "price", "brand", "size",

  pull_and_store_internal <- function(url){

    temp <- matrix(ncol = 6, nrow = 1)

    sold_item_test <- read_html(url)

    title <- sold_item_test %>% html_nodes('title') %>% html_text()
    price <- sold_item_test %>% html_nodes('._sold') %>% html_text()
    brand <- sold_item_test %>% html_nodes('.jumbo') %>% html_text()
    size <- sold_item_test %>% html_nodes('.listing-size') %>% html_text()

    script <- sold_item_test %>% html_nodes('script') %>% html_text()

    time <- script[6]
    time <- as.character(time)
    time <- regmatches(time, gregexpr("(?<=sold_at).*(?=sold_price.)", time, perl = TRUE))
    time <- gsub("[\"]","", time); time <- gsub(",","", time); time <- str_sub(time, 2)

    user <- script[6]
    user <- as.character(user)
    user <- regmatches(user, gregexpr("(?<=username).*(?=avatar_url)", user, perl = TRUE))
    user <- gsub("[\"]","", user); user <- gsub(",","", user); user <- str_sub(user, 2)

    temp[1,1] <- title
    temp[1,2] <- price
    temp[1,3] <- brand
    temp[1,4] <- size
    temp[1,5] <- time
    temp[1,6] <- user

    return(temp)
  }

  #load url
  url <- paste0('https://www.grailed.com/sold')
  lines <- readLines("scrape_sold.js")
  lines[1] <- paste0("var url ='", url , "';")
  writeLines(lines, "scrape_sold.js")
  system("./phantomjs scrape_sold.js")
  sold_page <- read_html("1.html")

  #iterate through list of 80
  for (i in 1:80){
    created_url <- paste("//*[@id=\"homepage-v2\"]/div/div/div[2]/div/div[2]/div/div[2]/div[", i , "]/a
    #print(created_url)
    temp_url <- sold_page %>% html_nodes(xpath = created_url) %>% xml_attr("href")
    temp_url <- paste("https://www.grailed.com", temp_url, sep = "")
    print(temp_url)
    temp_frame[i,] <- pull_and_store_internal(temp_url)
  }
  return(temp_frame)
}
```

Now, 1.7 minutes does not seem like a very long time to pull 80 items, however, we have to consider how often we would have to pull 80 items. In order to make sure we don't lose any data, we would have to run this every 1 hour or every 30 minutes just to be safe. That means to pull 1000 items (about a day's worth of sales on Grailed.com), we would have to spend about **22 minutes** of compute time over at least 13 regularly scheduled intervals through the day in order to get our relavant data, and this would not account for hours of fluctation where there might be an unusually large number of sales.

2.2 Algolia and CURL - Fast Method

Instead of loading each the *Grailed.com/sold* page each time to get the data, we can get the data straight from the source using CURL. With just a few lines of code, we can now have the information for 999 items. Approximate runtime for this method is less than **1 second** □ a huge improvement in time over the previous method.

```
#get 999 items
beans <- as.character('{"params":"query=&filters=(strata%3A\'grailed\'%20OR%20strata%3A\'hype\')%20AND%

#pull request
r <- POST("https://mnrwefss2q-dsn.algolia.net/1/indexes/Listing_sold_production/query?x-algolia-agent=A
```

Now we have to take the important variables from this huge dataset. So we make a function that sends the CURL request, pulls the 999 items, parses the sold times, and assigns intervals to each item based on its sold time. The most important part here is to set a proper interval that will capture a sufficient number of sales. Too low and the variation becomes too prevalant and it becomes hard to see trends. Too high and you will not be able to see trends in the data. A 12 hour interval fit well, especially since it represents exactly half the day and will show the same time in the AM and PM every time.

```
pull_and_merge_data <- function(){
  #get 999 hits
  beans <- as.character('{"params":"query=&filters=(strata%3A\'grailed\'%20OR%20strata%3A\'hype\')%20AND

  #pull request
  r <- POST("https://mnrwefss2q-dsn.algolia.net/1/indexes/Listing_sold_production/query?x-algolia-agent=

  #parse the hits
  object <- content(r, "parsed")$hits

  collected_data <- data.frame("title" = c(1:999), "price" = c(1:999), "sold_at" = c(1:999), "username"

  #fill dataframe
  for (i in 1:999){
    collected_data$title[i] <- object[[i]]$title
    collected_data$price[i] <- object[[i]]$price
    collected_data$sold_at[i] <- object[[i]]$sold_at
    collected_data$username[i] <- object[[i]]$user$username
    collected_data$bought_and_sold[i] <- object[[i]]$user$total_bought_and_sold
    collected_data$designer[i] <- object[[i]]$designer$name
    collected_data$sold_price[i] <- object[[i]]$sold_price
    collected_data$interval <- 1
  }

  c <- anti_join(backup, collected_data, by = "title")
  backup <- rbind(collected_data,c)

  # c <- anti_join(df, collected_data, by = "title")
  # collected_data <- rbind(collected_data,c)

  #interval assignment
  x <- 10

  collected_data_reorder <- backup[order(backup$sold_at),]
  collected_data_reorder$sold_at <- parse_date(collected_data_reorder$sold_at)

  i <- 1

  collected_data_reorder$interval <- 1

  max <- as.numeric(trunc((max(collected_data_reorder$sold_at) - min(collected_data_reorder$sold_at)) /

  for (i in 1:(max+1)){
    collected_data_reorder$interval[which(collected_data_reorder$sold_at <= (collected_data_reorder$sold
  }

  write.csv(collected_data_reorder, file = "cleaned_agg_data.csv")
  print("CSV has been written")
  return(collected_data_reorder)
}
```

```
backup <- pull_and_merge_data()
```

3 Dashboard

3.1 Setting up Shiny interface

The Shiny interface runs separately from the data retrieval script. Thus we can set a task scheduler in our main script to continuously update our dataframe with new data every 12 hours. The dashboard shows the top 10 brands in terms of sales count on Grailed.com.

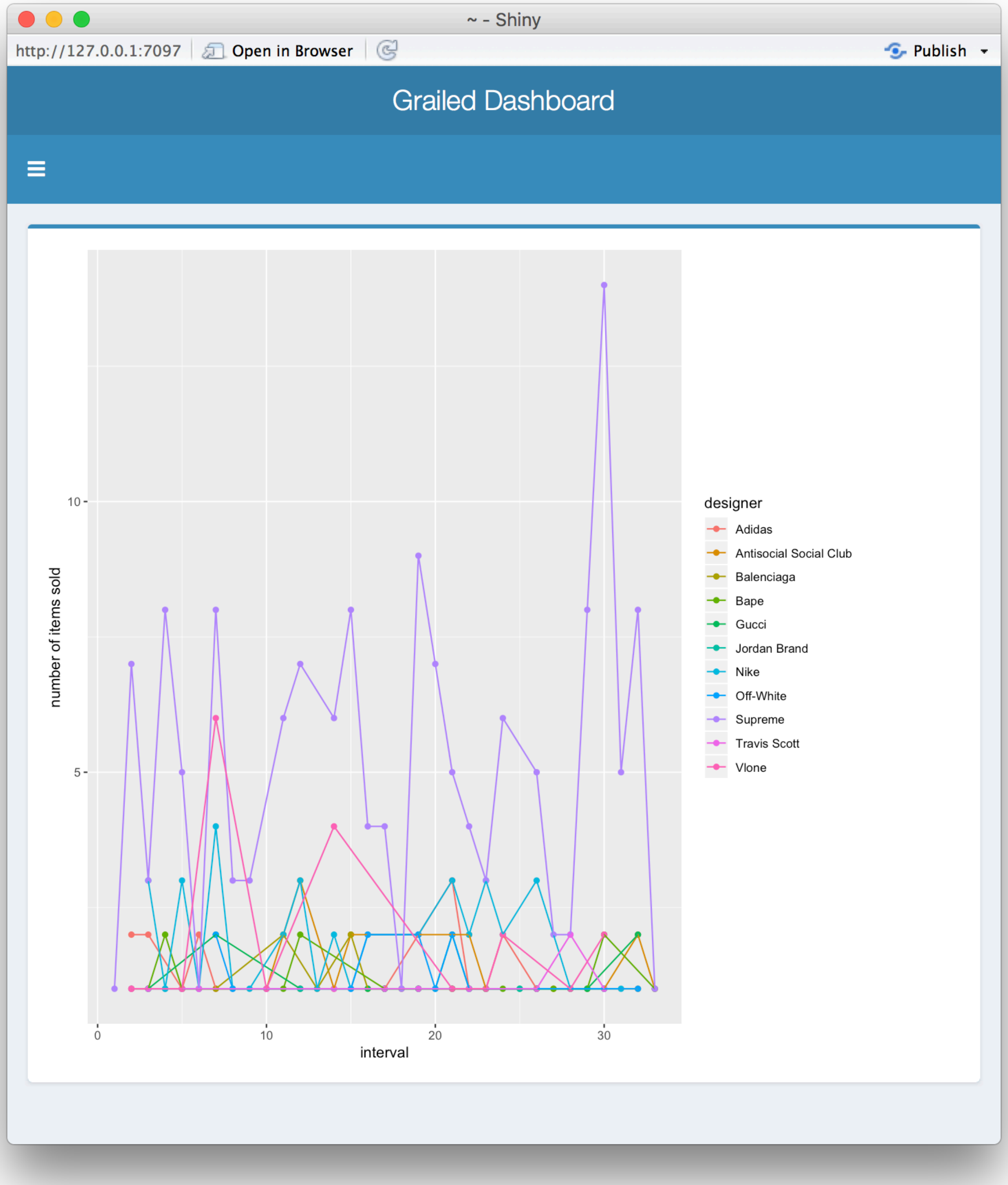
```
tclTaskSchedule(43200, pull_and_merge_data(), id = "test", redo = TRUE)
```

Shiny code:

```
ui <- dashboardPage(
  dashboardHeader(title = "Grailed Dashboard"),
  dashboardSidebar(),
  dashboardBody(
    # Boxes need to be put in a row (or column)
    fluidRow(
      box(titel = "Grailed", status = "primary", plotOutput("plot1", height = 600, width = 600))
    )
  )
)

server <- function(input, output) {
  output$plot1 <- renderPlot({
    ggplot(data = test, aes(x = interval, y = n, color = designer)) + geom_point() + geom_line() + ylab
  })
}

shinyApp(ui, server)
```



4 Where to go from here

4.1 Poisson regression and time series analysis

Since we are working with count data, we could use our current data and the pattern of sales to make predictions on future brand sales count and what kind of pattern they could exhibit.

4.2 Filters and options

The dashboard can be made more user friendly and customizable. One way is to include filters that allow users to pick which brands they want to see. Another option would be for a user customizable time interval, so users can take a closer or broader look into sales count data.

4.3 Set up on a live website

The Shiny app runs as an extension of RStudio as of now, but the eventual goal is to set it up on its own server to have it be truly live.

5 What I learned

I PhantomJS - How to render a webpage in the background for your to be able to access elements. This is very helpful to reduce clutter in your program.

II Rvest - The simplest and most straightforward way to webscrape in R, however it is certainly not the most efficient. This helped this project get started in the first place proving that almost anything you can see on a site can be scraped in some way or another.

III Curl, HTTR, Alogolia - Getting information from the source is always a better way than surface-level web scraping. This was a much deeper way of accessing the data.

IV The most efficient way often takes the longest to figure out, but in the long run it's worth doing. It helped a lot to spend a 1-2 week deviation from the project in order to make the method of doing the project much cleaner and simpler.