CSCI 385 - Second Deliverable

Kris Selvidge

12/03/2020

Introduction

The domain of this project is network data and how we can use R to help visualize performance metrics. For this second deliverable I have narrowed down the more interesting website root captures from the previous deliverable and reran the experiment to collect more data.

Using Wireshark, a network packet capture program, I collected traffic data for each experiment session for analysis. Results were exported to Javascript Object Notation format (JSON) and read into this R knit.

A brief understanding of how my experiment was setup is indicated in the data set section from deliverable 1. Background information on Wireshark is recommended, but specifics used for this capture are detailed below.

Revised Data Set 1

After reviewing the data set for the previous first deliverable my instructor and classmates pointed out that some of the sites surveyed had either a much greater amount of data returned and/or greater variability seen throughout visualization. When rerunning the experiment for Wireshark capture, we limited the scope of captures to this subset selection of websites. The size of the JSON Wireshark files were significantly large due to traffic analysis additions (423 different column field types are recorded during a Wireshark capture). I decided rather to run the experiment on all sites in one bash script that I would instead break it down into one script per site. This resulted in a separately generated comma separated file (csv) for output fromd data set 1. While I could have manually merged these files together prior to importing into R, I have chosen to load each separately and practice combining them within R Studio.

Although it wasn't mentioned in deliverable 1, this first dataset is considered "tidy". Each column represents only one variable value and each row represents a single observation.

Loading Data Set 1 Subset Site 1: Amazon

```
amazon <- read_csv('D:\\fall2020\\TTH\\CSCI385 Data Science\\a2\\amazon.csv')
```

```
## Parsed with column specification:
## cols(
##   source = col_character(),
##   destination = col_character(),
##   protocol = col_character(),
##   chunkSize = col_double(),
##   tCount = col_double(),
##   fileSize = col_double(),
##   retransmits = col_double(),
```

```
##   connectTime = col_double(),
##   requestTime = col_double(),
##   receiveTime = col_double()
## )
```

Loading Data Set 1 Subset Site 1: Facebook

```
facebook <- read_csv('D:\\fall2020\\TTH\\CSCI385 Data Science\\a2\\facebook.csv')
```

```
## Parsed with column specification:
## cols(
##   source = col_character(),
##   destination = col_character(),
##   protocol = col_character(),
##   chunkSize = col_double(),
##   tCount = col_double(),
##   fileSize = col_double(),
##   retransmits = col_double(),
##   connectTime = col_double(),
##   requestTime = col_double(),
##   receiveTime = col_double()
## )
```

Loading Data Set 1 Subset Site 1: Google

```
google <- read_csv('D:\\fall2020\\TTH\\CSCI385 Data Science\\a2\\google.csv')
```

```
## Parsed with column specification:
## cols(
##   source = col_character(),
##   destination = col_character(),
##   protocol = col_character(),
##   chunkSize = col_double(),
##   tCount = col_double(),
##   fileSize = col_double(),
##   retransmits = col_double(),
##   connectTime = col_double(),
##   requestTime = col_double(),
##   receiveTime = col_double()
## )
```

Loading Data Set 1 Subset Site 1: Yahoo

```
yahoo <- read_csv('D:\\fall2020\\TTH\\CSCI385 Data Science\\a2\\yahoo.csv')
```

```
## Parsed with column specification:
## cols(
##   source = col_character(),
##   destination = col_character(),
##   protocol = col_character(),
##   chunkSize = col_double(),
```

```
##   tCount = col_double(),
##   fileSize = col_double(),
##   retransmits = col_double(),
##   connectTime = col_double(),
##   requestTime = col_double(),
##   receiveTime = col_double()
## )
```

Loading Data Set 1 Subset Site 1:Youtube

```
youtube <- read_csv('D:\\fall2020\\TTH\\CSCI385 Data Science\\a2\\youtube.csv')
```

```
## Parsed with column specification:
## cols(
##   source = col_character(),
##   destination = col_character(),
##   protocol = col_character(),
##   chunkSize = col_double(),
##   tCount = col_double(),
##   fileSize = col_double(),
##   retransmits = col_double(),
##   connectTime = col_double(),
##   requestTime = col_double(),
##   receiveTime = col_double()
## )
```

Combining all Data Set 1 Subset Sites

```
netdat <- bind_rows(amazon, facebook, google, yahoo, youtube)
head(netdat)
```

```
## # A tibble: 6 x 10
##   source destination protocol chunkSize tCount fileSize retransmits connectTime
##   <chr>  <chr>       <chr>        <dbl>  <dbl>    <dbl>       <dbl>       <dbl>
## 1 192.1~ www.amazon~ HTTP/1.0         4      7     1263           2    0.00084
## 2 192.1~ www.amazon~ HTTP/1.1         4      7     1263           2    0.000445
## 3 192.1~ www.amazon~ HTTP/1.0         5     12     1263           2    0.000437
## 4 192.1~ www.amazon~ HTTP/1.1         5     12     1263           2    0.000463
## 5 192.1~ www.amazon~ HTTP/1.0         6     13     1263           2    0.000448
## 6 192.1~ www.amazon~ HTTP/1.1         6     13     1263           2    0.000520
## # ... with 2 more variables: requestTime <dbl>, receiveTime <dbl>
```

Categorical Variables:

- 'source' - 'character' - Origin location requesting root file from destination. Note: In this phase of the experiement there is only one location (localhost) requesting data. This computer is my local HP Omen 17 laptop running Ubuntu 18.
- 'destination' - 'character' - Web server sending root file. These hosts are ten top and popular websites.
- 'protocol' - 'character' - Transfer protocol version HTTP (Hypertext Transfer Protocol) 1.0 or 1.1. Note that twitter.com does not respond to HTTP 1.1 requests.

Continuous Variables

- 'chunkSize' - 'double' - Size of each chunk of packet data in bytes. This value is the only continuous variable adjusted during the experiment. Values range from 4 to 1000.

(Note all variable values below were observed as a part of the experiment.)

- 'tCount' - 'double' - Number of completed HTML markup tags parsed within chunks. Tags markup content within documents and are opened with character '<' and closed with character '>'.
- 'fileSize' - 'double' - Total root file size in bytes.
- 'retransmits' - 'double' - Number of packets smaller than chunkSize indicating lost bytes that required a retransmit of data.
- 'connectTime' - 'double' - Time in seconds to establish connection from source to destination.
- 'requestTime' - 'double' - Time in seconds to submit request from source to destination.
- 'receiveTime' - 'double' - Time in seconds to receive root file from destination to source.

Manipulated/Independent Variables

For our program experiment, we manually chose the following three variables as input to determine the resulting observations:

destination (amazon.com, facebook.com, google.com, yahoo.com, youtube.com) protocol (HTTP/1, HTTP/1.1) chunkSize (4:120)

Exploratory Data Analysis

In our first deliverable we were primarily focused on seeing what impact the size of each chunk packet processed in an HTTP transfer and whether protocols (HTTP/1 or HTTP/1.1) made a difference. Using our 10 websites I observed that the only variables effected by chunkSize were tag counts (tCount), retransmits and receiving time. After narrowing the observations and doing a rerun, I reviewed the relationship between these variables to confirm this still existed.

Later we will try to model the impact of of our previously denoted manipulated/independent variables to predict their impact on one or more of these variables, so I am setting aside some data for testing.

```
rest_rows <- as.vector(createDataPartition(netdat$chunkSize, p = 0.8, list = FALSE))
test <- netdat[-rest_rows, ]
rest <- netdat[rest_rows, ]
```

```
summary(rest)
```

```
##     source          destination          protocol          chunkSize
##  Length:936         Length:936         Length:936         Min.   :  4.00
##  Class :character   Class :character   Class :character   1st Qu.: 33.00
##  Mode  :character   Mode  :character   Mode  :character   Median : 62.00
##                                                           Mean   : 62.15
##                                                           3rd Qu.: 91.00
##                                                           Max.   :120.00
##      tCount         fileSize      retransmits      connectTime
```

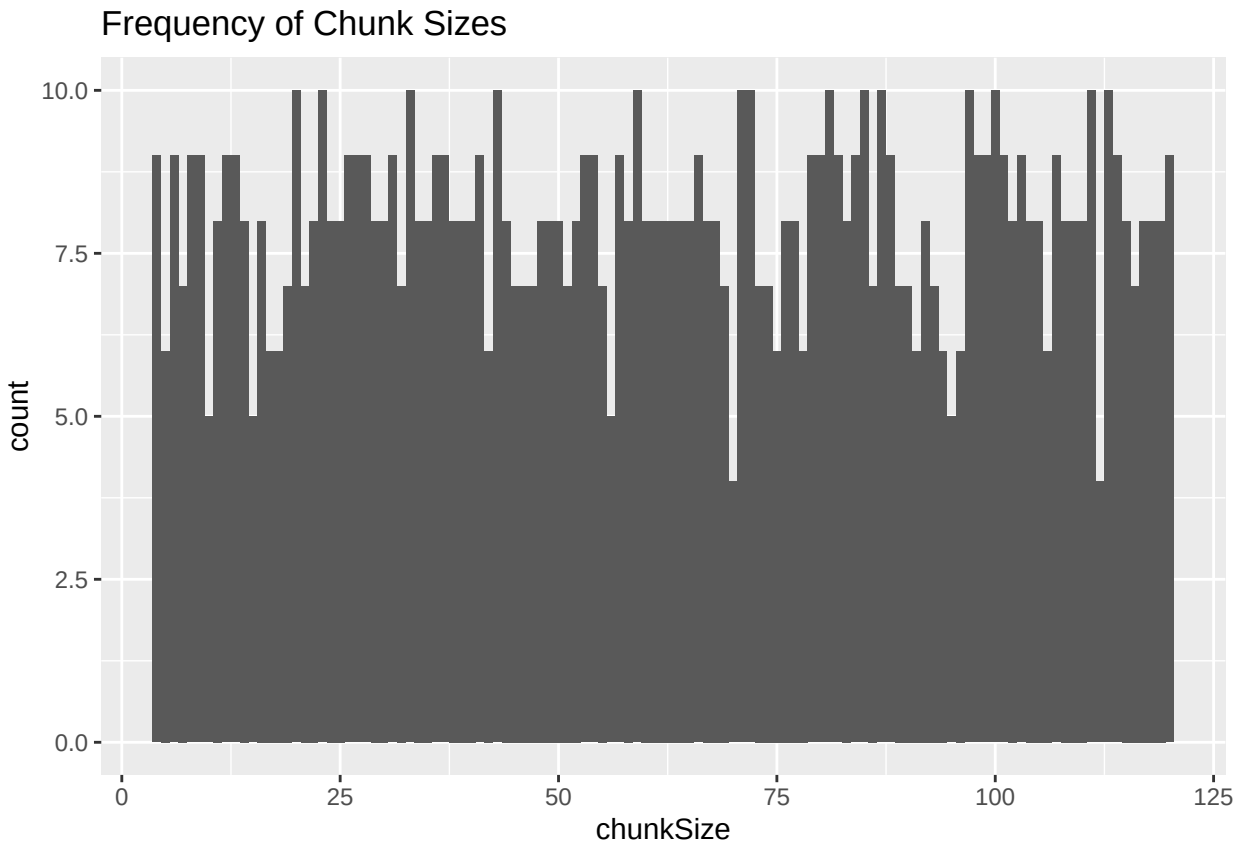
```
##  Min.   :  0.0   Min.   :  366   Min.   : 1.000   Min.   :0.0003900
##  1st Qu.: 19.0   1st Qu.: 1263   1st Qu.: 2.000   1st Qu.:0.0004410
##  Median : 53.0   Median : 4612   Median : 2.000   Median :0.0004725
##  Mean   :129.9   Mean   :21279   Mean   : 6.191   Mean   :0.0004886
##  3rd Qu.:283.0   3rd Qu.:48634   3rd Qu.:12.000   3rd Qu.:0.0005040
##  Max.   :299.0   Max.   :49448   Max.   :22.000   Max.   :0.0018950
##   requestTime         receiveTime
##  Min.   :3.200e-05   Min.   :0.0000080
##  1st Qu.:4.900e-05   1st Qu.:0.0000710
##  Median :5.300e-05   Median :0.0001345
##  Mean   :6.682e-05   Mean   :0.0005330
##  3rd Qu.:6.000e-05   3rd Qu.:0.0007630
##  Max.   :5.340e-04   Max.   :0.0067160
```

Summary Observations: While in the first deliverable we looked at 18943 observations, in this case we have drastically fewer at 936 observations in deliverable 2. Part of this is explained by a smaller set of websites. However, I also reduced the number of the chunkSize variations because we saw very few changes to other variables when any chunkSize was over about 120, thus while the chunkSize minimum is still 4 we now have a new maximum of 120 rather than 1000. The old range was 4-1000 or 996 total before and is now 4-120 or 116, thus we reduced by 880 which as a percentage of 996 is roughly 88% leaving 12% left. As a few sites do not support http/1.1, removing half of the sites was not a perfect 50% but very close. Finally we also reduced the results by 20% to set aside as test data so the remainder is 80%. A comparison calculation closely matches the number of actual observations. (18943 * 50% * 12% * 80% or 909 vs 936 actual)

Every request resulted in a response as it did in the first deliverable. These sites were chosen because their root files were significantly larger than the other site root responses in the deliverable 1 observations. From the summary we can confirm this by seeing the minimum file size returned is now 366 bytes whereas it was 150 in deliverable 1. The median fileSize increased from 418 to and the mean from 1192 to. The max remains about the same (49712 to 49669).

Chunk Size:

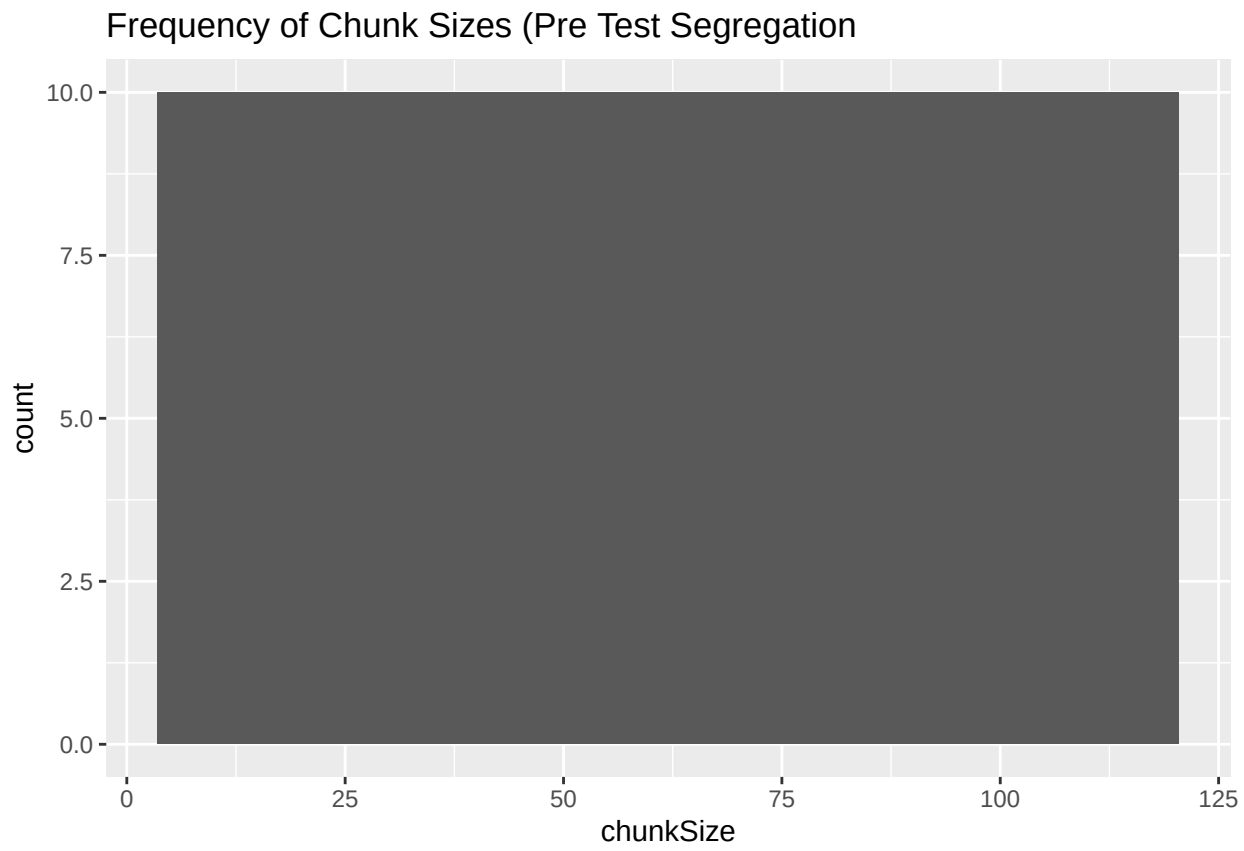
```
ggplot(data = rest) +
  geom_histogram(mapping = aes(x = chunkSize), binwidth = 1)+
  ggtitle("Frequency of Chunk Sizes")
```



Chunk Size is the variable we manually chose in our experiment to alter, so we should know exactly what the frequency is for each. I made a range of chunkSize 4 to 120 in a regular integer interval (4,5,6...118,199,120). There is one observation per chunk Size interval different per website (5) per protocol (2). What we should in the histogram above is a flat line of a count of 10 for every chunk size.

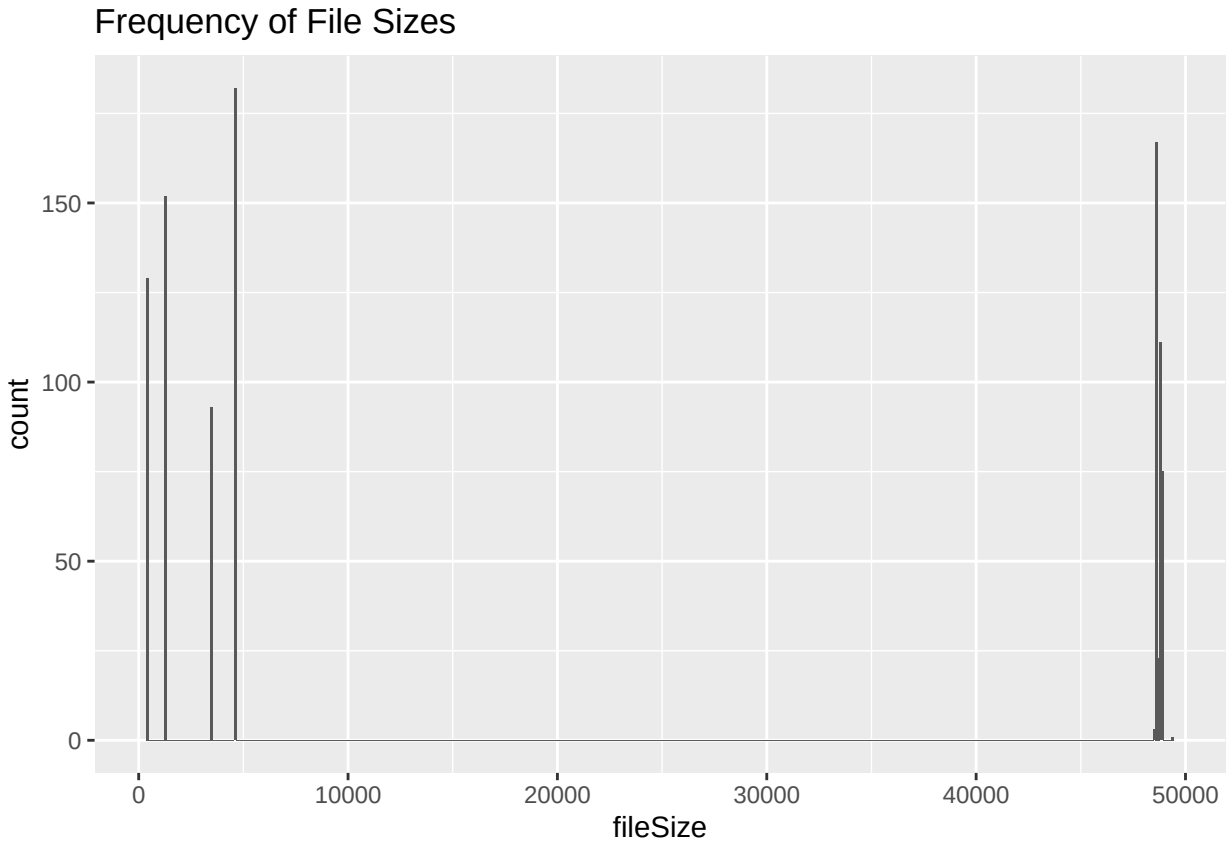
So why is it that this is not the case above? While we do see a flat peak of 10 thoroughout the graph, we also see counts dipping randomly. This is because we set aside 20% of the observation for future testing! Just to confirm this, we can see what we would expect to see from the original unaltered data.

```
ggplot(data = netdat) +
  geom_histogram(mapping = aes(x = chunkSize), binwidth = 1)+
  ggtitle("Frequency of Chunk Sizes (Pre Test Segregation")
```



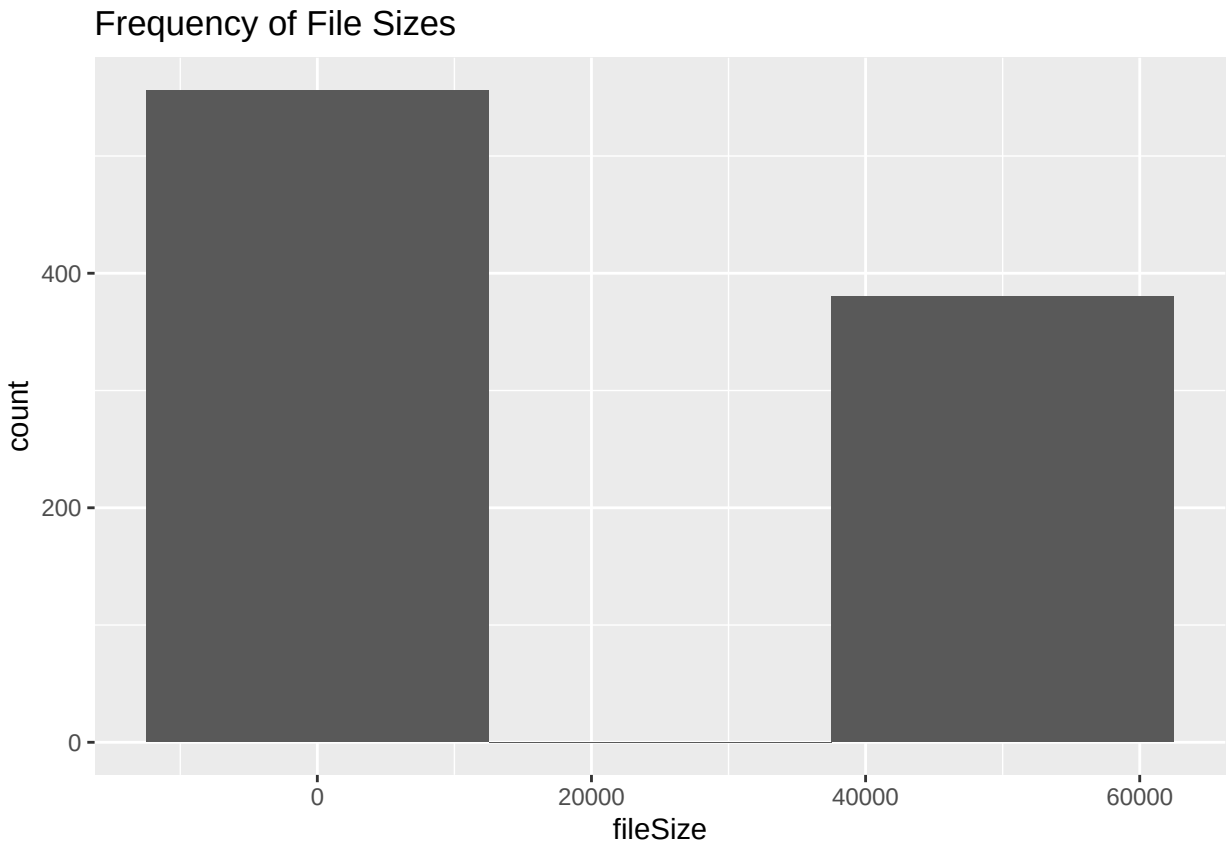
File Size:

```
ggplot(data = rest) +
  geom_histogram(mapping = aes(x = fileSize), binwidth = 100)+
  ggtitle("Frequency of File Sizes")
```



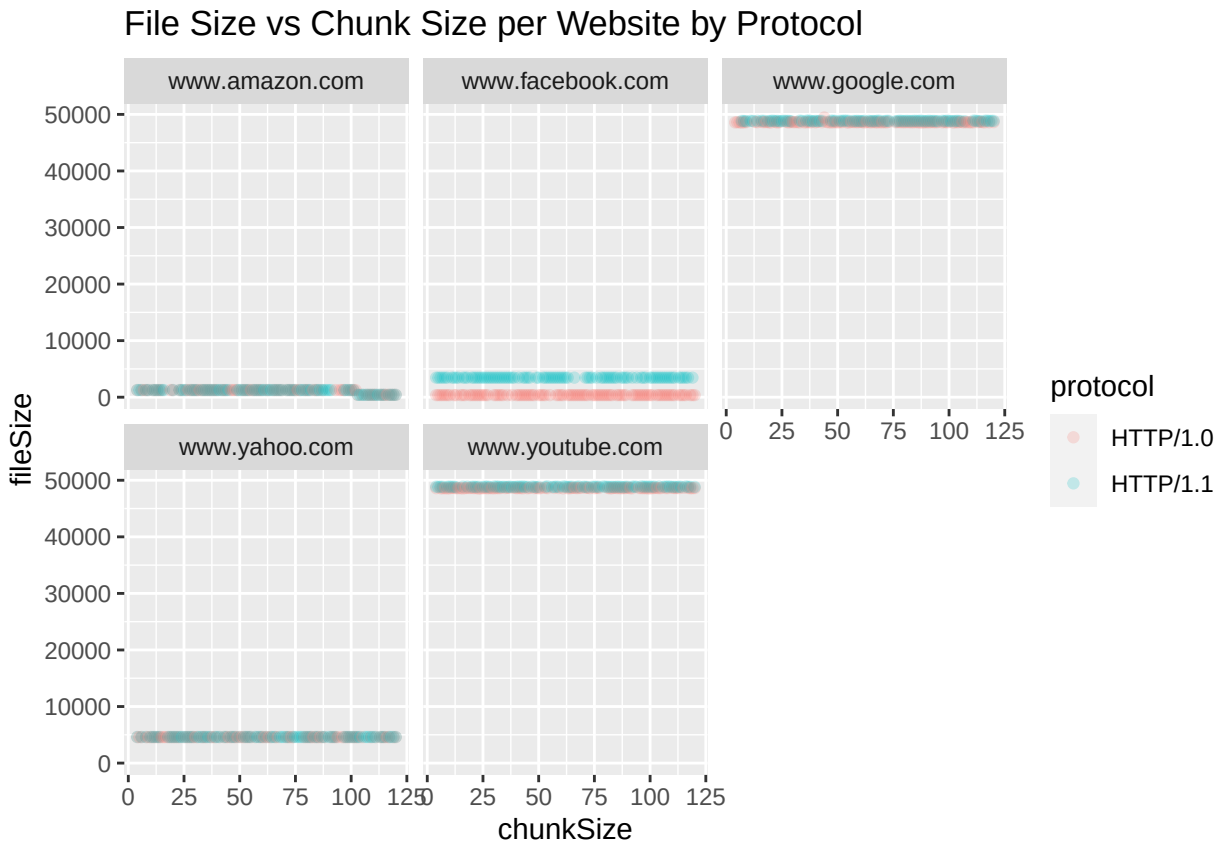
We can see from the graph above for file size frequency that we are repeatedly receiving the same size files but there is some variability and a huge gap where there are no files between size 5000 and 45000 bytes. Most observations show files near evenly split as either being under 5000 or between 45000-50000 bytes. Is it perfectly even? Let's change the width of the histogram bins and observe the results.

```
ggplot(data = rest) +
  geom_histogram(mapping = aes(x = fileSize), binwidth = 25000)+
  ggtitle("Frequency of File Sizes")
```



From comparing both charts above we can deduce that it is not exactly even, but as an approximation we could say of the 936 observations a little less than 400 are in the 45000-50000 range and somewhere around 550 are under 5000 bytes. To try to understand why so many file sizes are clustered around each other, let's apply the graphs we used in deliverable one.

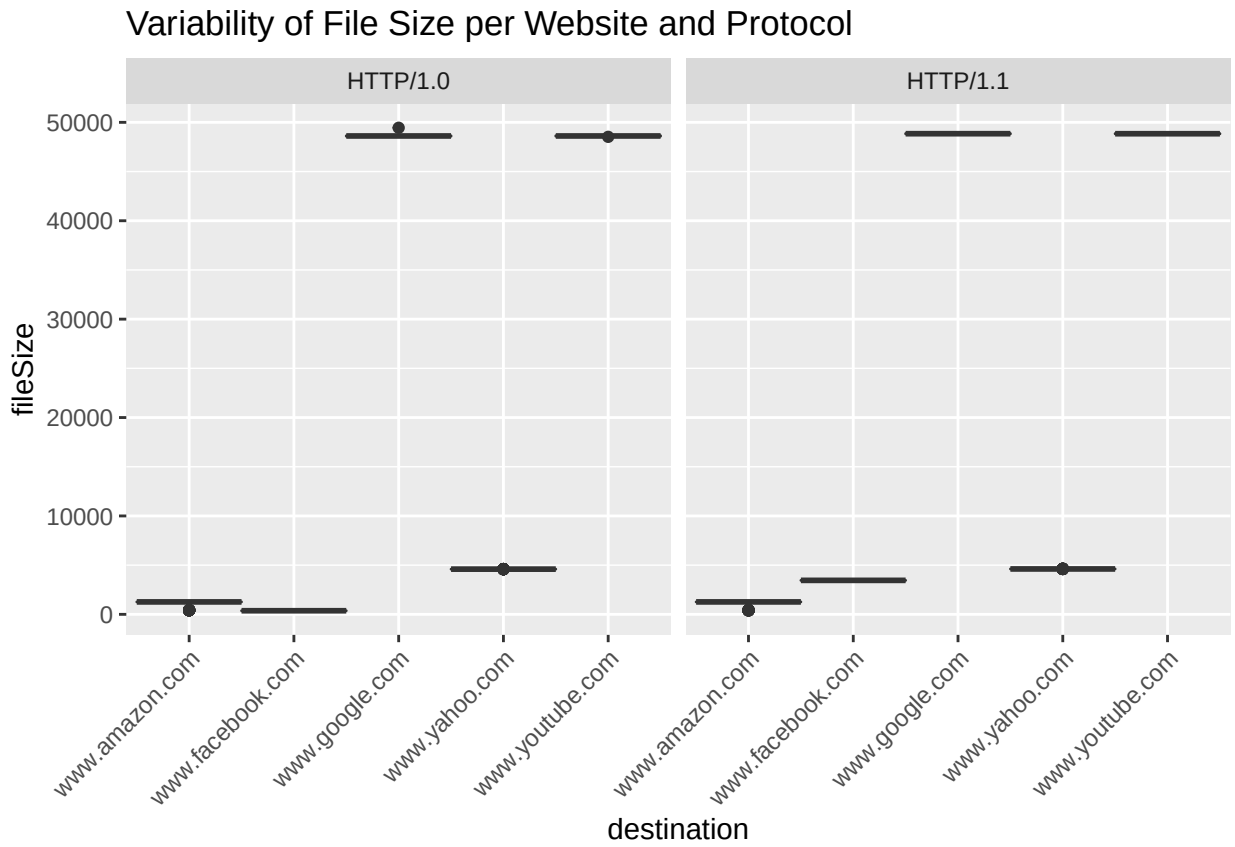
```
ggplot(data = rest) +
  geom_point(mapping = aes(x = chunkSize, y = fileSize, color = protocol), alpha = 0.2) +
  facet_wrap(~ destination, nrow = 2) +
  ggtitle("File Size vs Chunk Size per Website by Protocol")
```



We can confirm from this table that the results are nearly identical to deliverable 1. Google and Youtube root calls deliver substantially larger file sizes, explaining the gap in the first histogram above.

What we did not note in the first deliverable is that it seems as though the file size is almost always the same for each web site (destination). We can confirm this with a boxplot below:

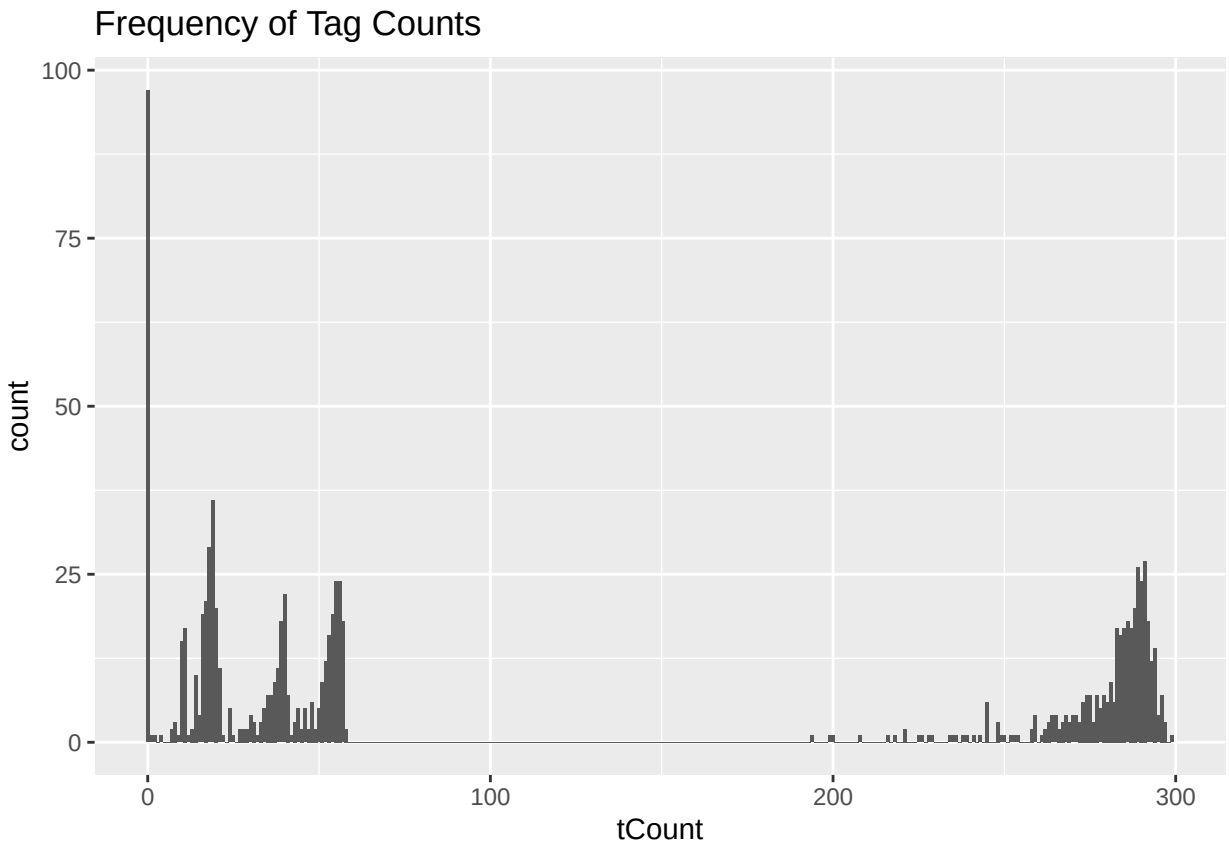
```
ggplot(data = rest) +
  geom_boxplot(mapping = aes(x = destination, y = fileSize), width = 1) +
  facet_wrap(~ protocol) +
  ggtitle("Variability of File Size per Website and Protocol") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```



I am not quite sure how this connection may be of use to me later, but I'm going to make a note here that anyplace we measure the continuous variable fileSize as a factor on another variable we may also in some way associate it with the categorical separation by website (destination). In other words, if we see an impact on another variable by fileSize, we cannot be certain it's not some other attribute of the website because we have not collected a variety of file sizes from the same website to test.

Tag Counts:

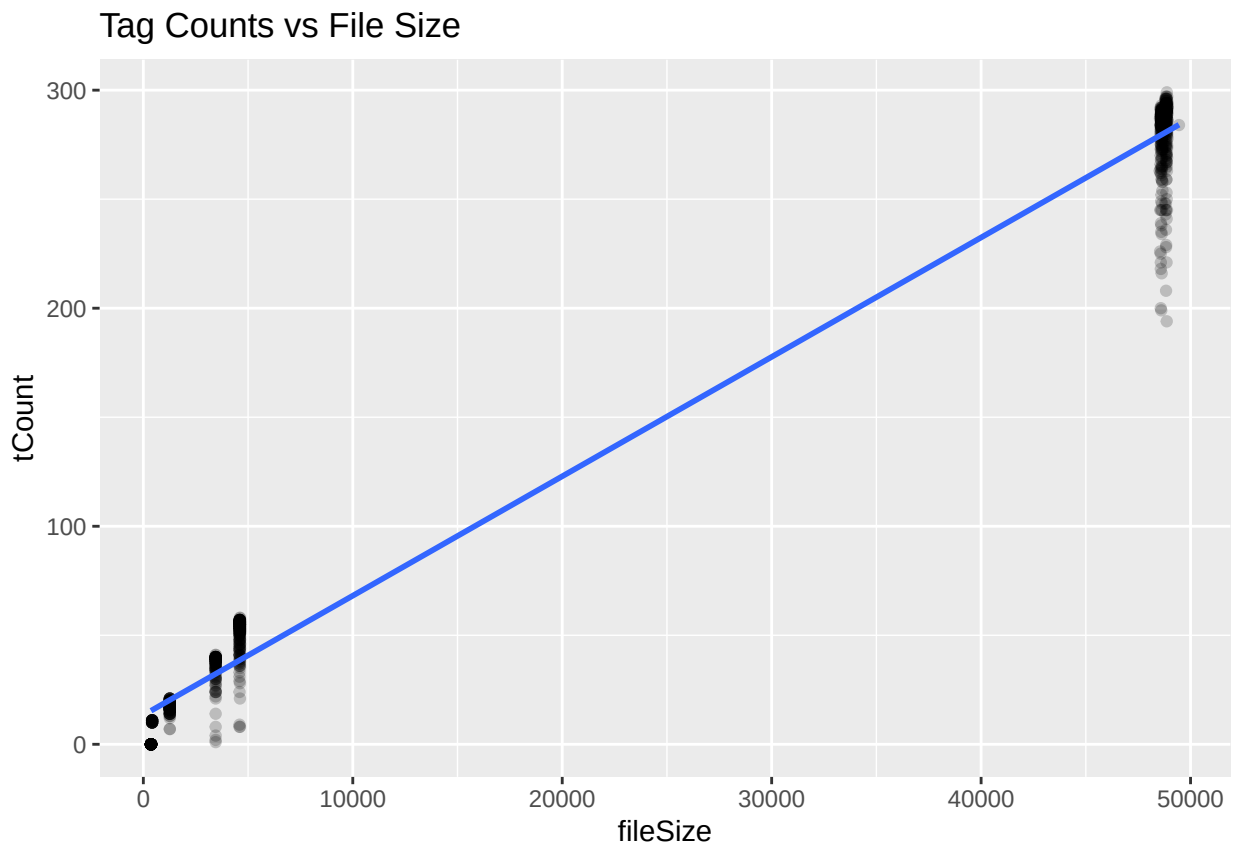
```
ggplot(data = rest) +
  geom_histogram(mapping = aes(x = tCount), binwidth = 1) +
  ggtitle("Frequency of Tag Counts")
```



We can see a significant gap in frequency for tag counts. There are almost no observation of tag counts less than 50 or greater than 250, with almost all clustering under 50 or between 250 and 300. This gap pattern is similar to the gap in file size above. Let's next see if there looks like a pattern when comparing the two.

```
ggplot(data = rest) +
  geom_point(mapping = aes(x = fileSize, y = tCount), alpha = 0.2) +
  geom_smooth(mapping = aes(x = fileSize, y = tCount), method="lm", se = FALSE) +
  ggtitle("Tag Counts vs File Size")
```

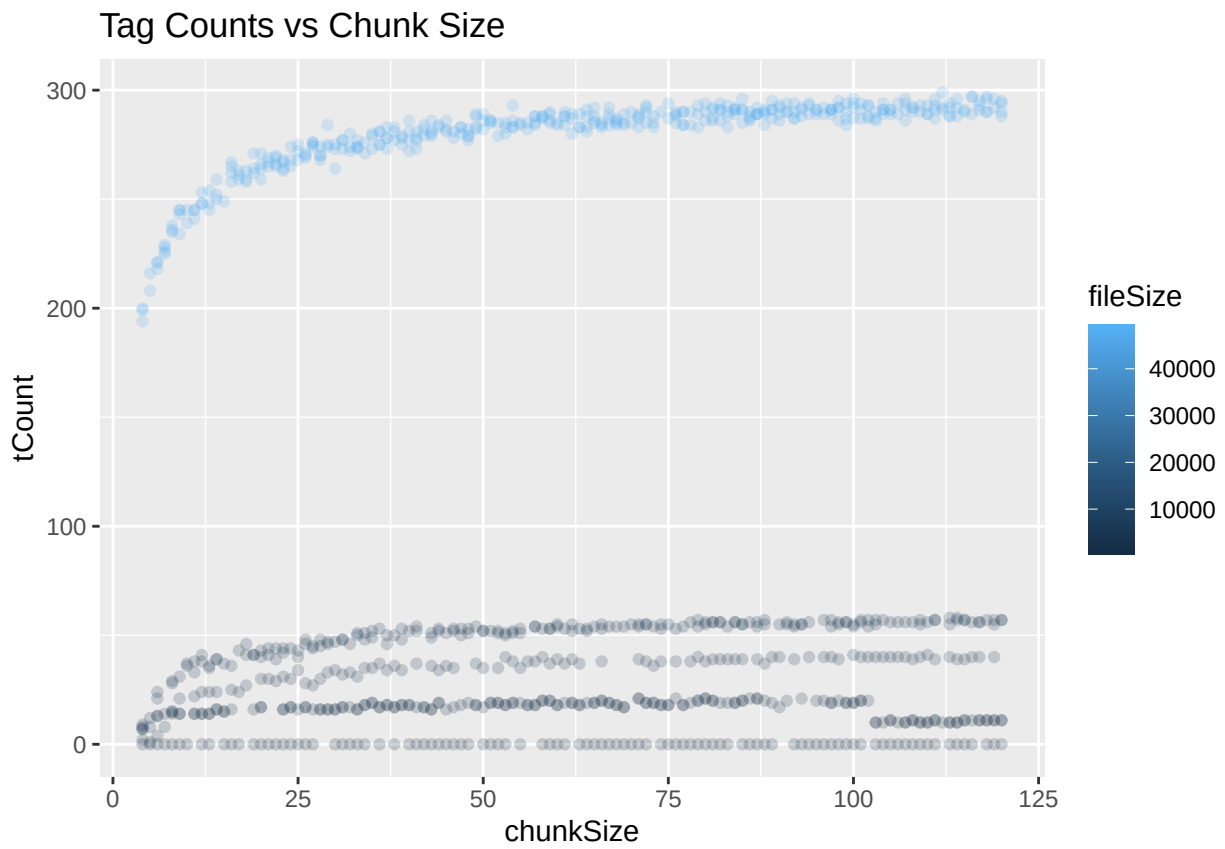
```
## `geom_smooth()` using formula 'y ~ x'
```



We can observe from increases in file size correspond with increases in tag counts and that the clusters we saw in histograms for both variables match (files $<$ 5000 bytes have $<$ 50 tags and files with 45000:50000 bytes have 250:300 tags).

Is there any influence from our manipulated variable from the experiment? Let's compare with chunk size.

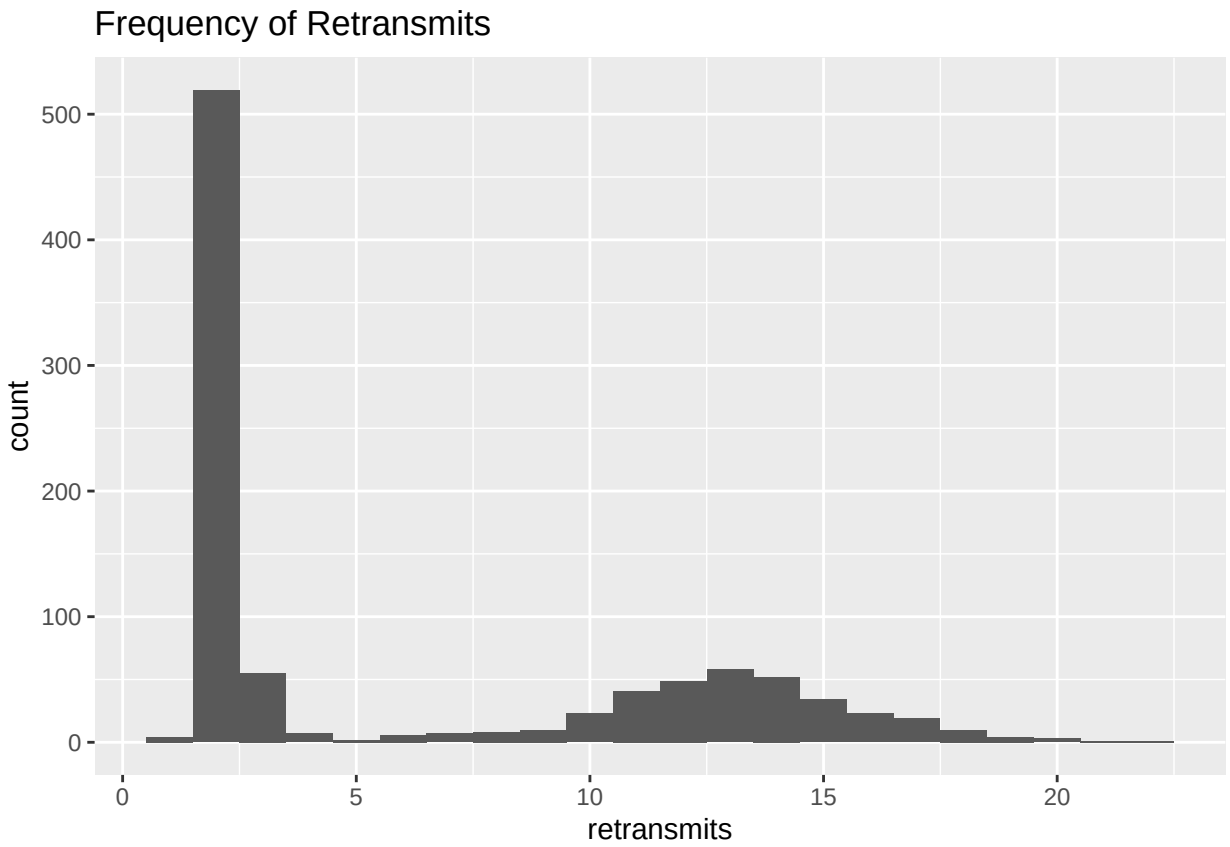
```
ggplot(data = rest) +
  geom_point(mapping = aes(x = chunkSize, y = tCount, color=fileSize), alpha = 0.2) +
  ggtitle("Tag Counts vs Chunk Size")
```



For both groups of tags we looked at earlier (<50, 250:300) there is an increase of tag counts as chunk size increases. The relationship appears to be logarithmic.

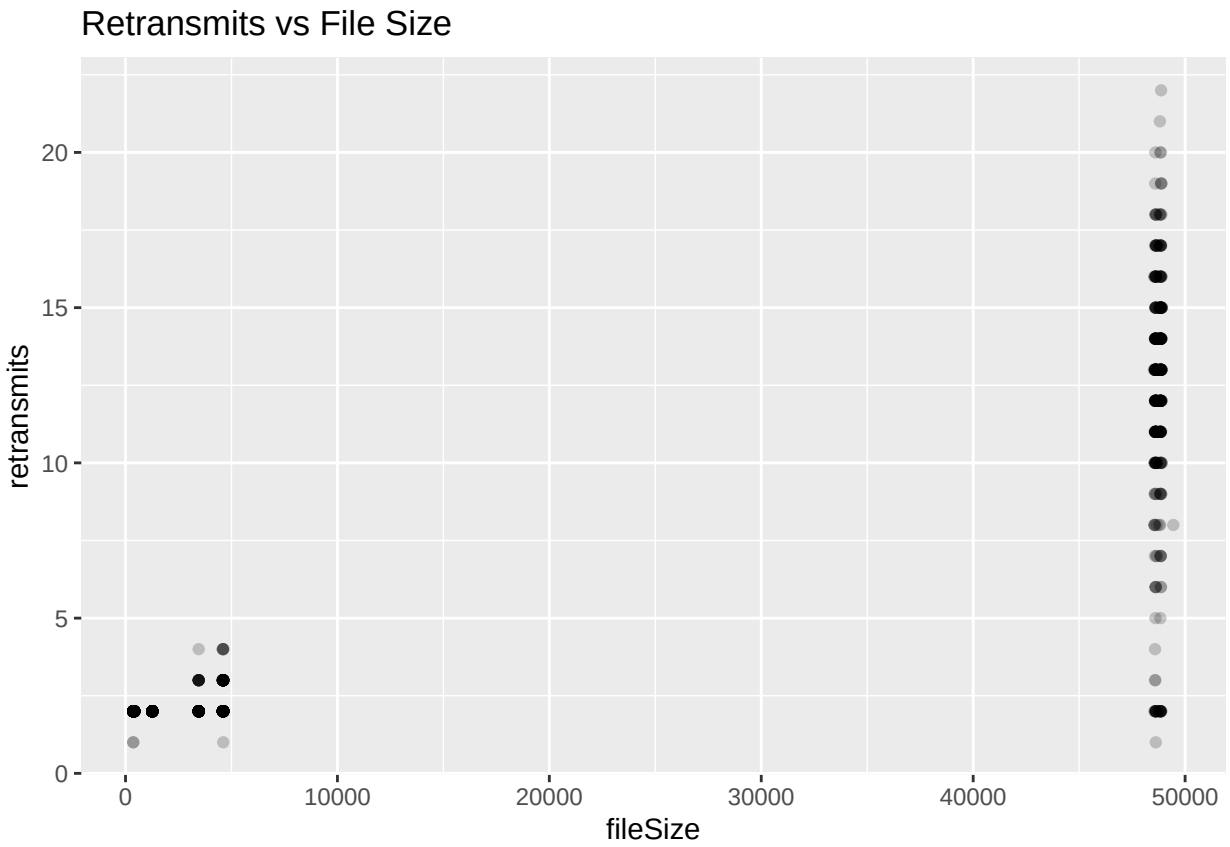
Retransmits:

```
ggplot(data = rest) +
  geom_histogram(mapping = aes(x = retransmits), binwidth = 1)+
  ggtitle("Frequency of Retransmits")
```



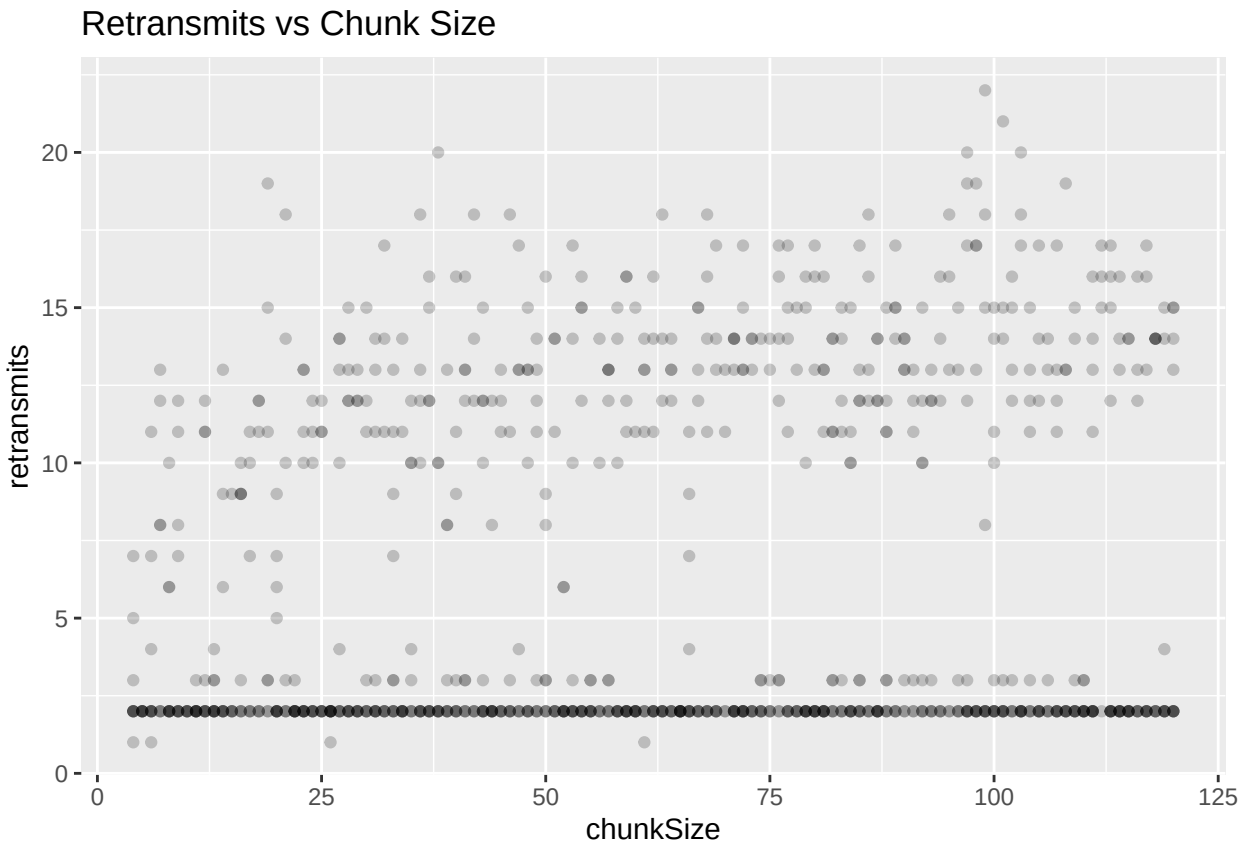
I'm seeing a large number of observations where files did not require many retransmits of packets, with some files requiring 10-15 packets to be resent. Let's first look to see if larger files might be creating the need for more retransmits.

```
ggplot(data = rest) +
  geom_point(mapping = aes(x = fileSize, y = retransmits), alpha = 0.2) +
  ggtitle("Retransmits vs File Size")
```



Each cluster of file sizes appear to have all ranges of retransmission values. There does not appear to be a pattern of influence from looking at this. Let's try chunk size next.

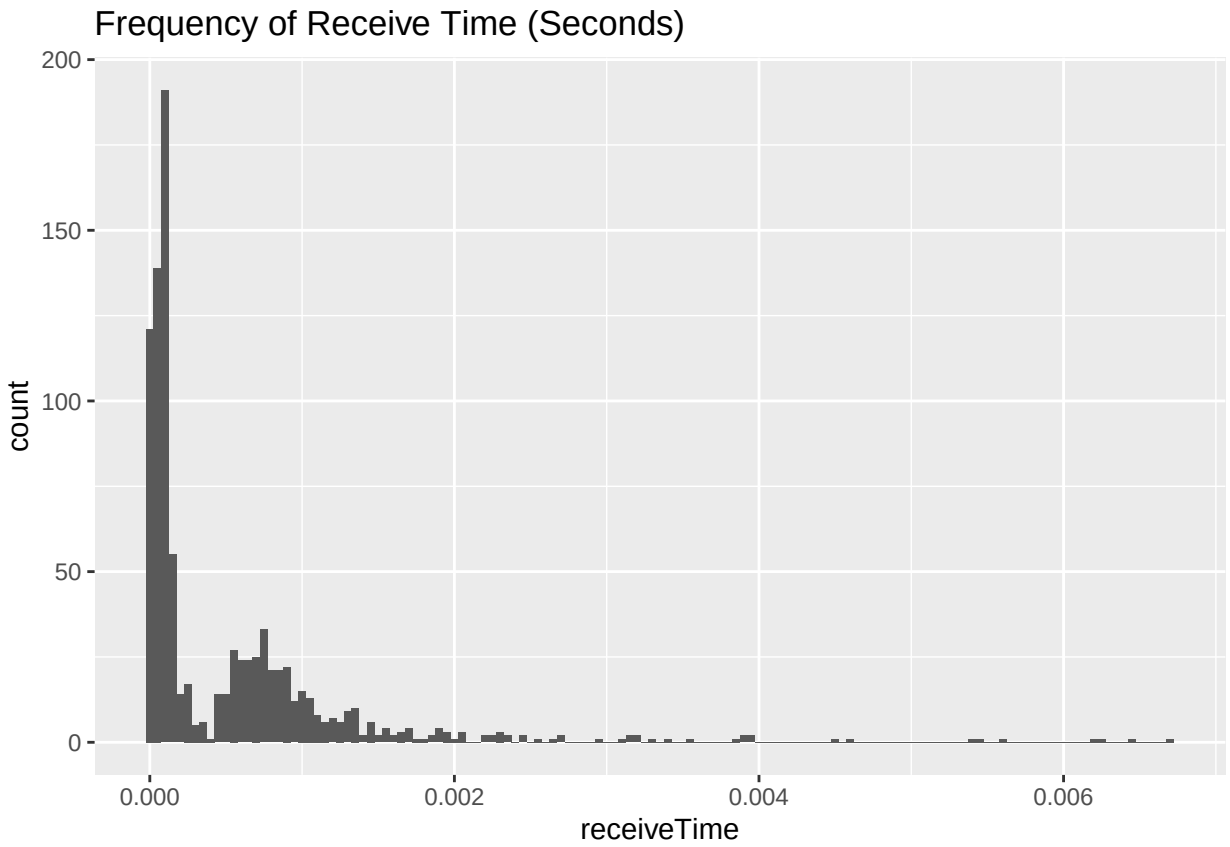
```
ggplot(data = rest) +
  geom_point(mapping = aes(x = chunkSize, y = retransmits), alpha = 0.2) +
  ggtitle("Retransmits vs Chunk Size")
```



Again the values are widely spread. While in the first deliverable we saw a few clusters of note in some of the websites at very high chunk values, we are not finding anything here in the revised set. I will discard the number of retransmission in our modeling for now.

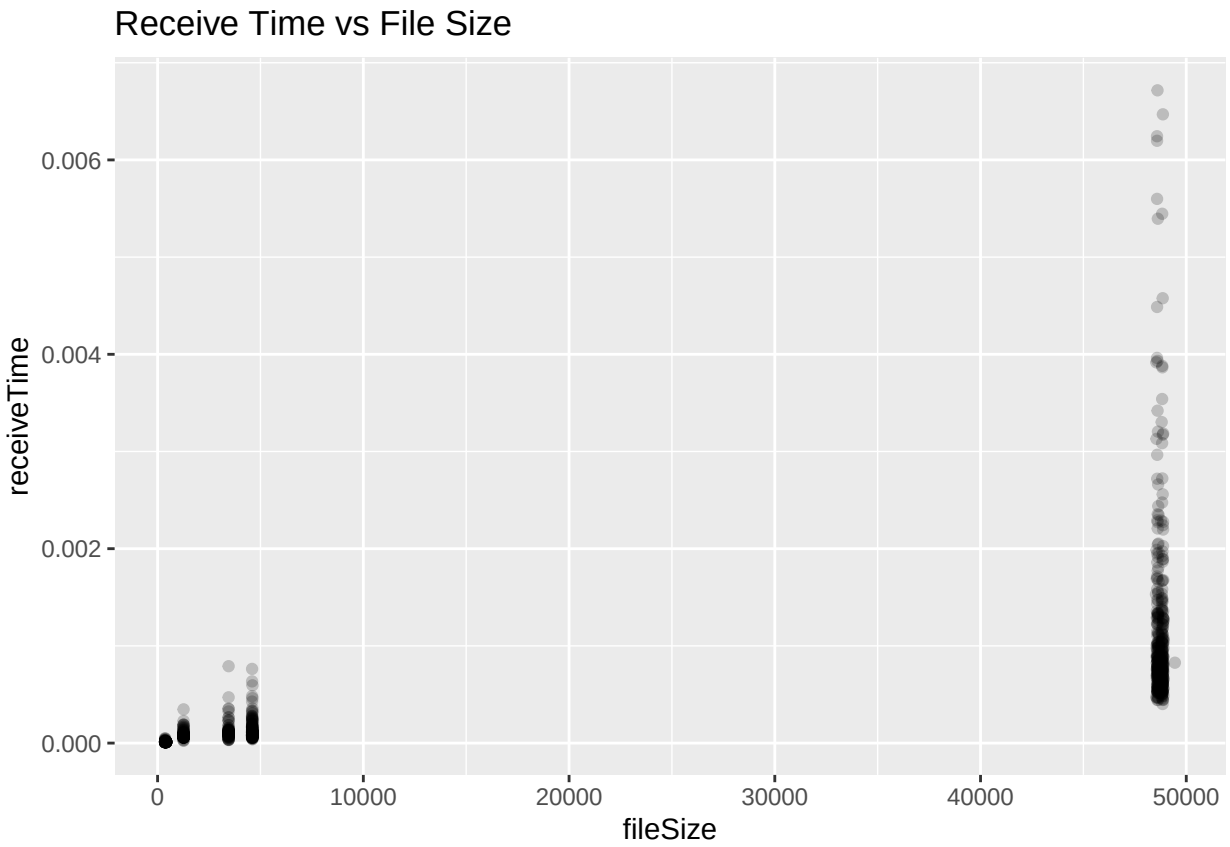
Receive Time:

```
ggplot(data = rest) +
  geom_histogram(mapping = aes(x = receiveTime), binwidth = 0.00005) +
  ggtitle("Frequency of Receive Time (Seconds)")
```



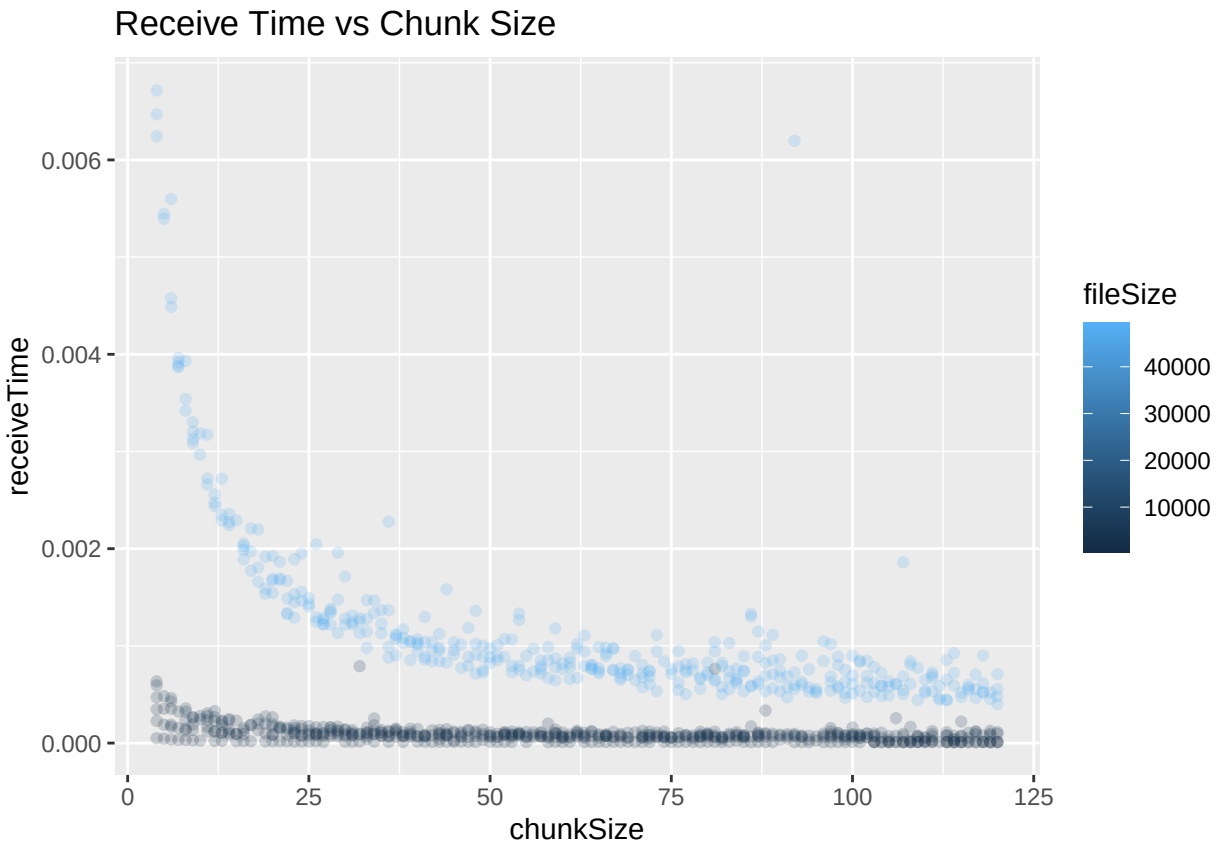
A large number of file transfers occurred under 0.005 with a sloped hill of frequency between the 0.005 and 0.015 value. Very few took more than 0.015 seconds but there are a few scattered up to 0.070. There are some minor gaps but nothing matching file size or tag counts. Let's look at how it matches against file size directly:

```
ggplot(data = rest) +
  geom_point(mapping = aes(x = fileSize, y = receiveTime), alpha = 0.2) +
  ggtitle("Receive Time vs File Size")
```



While there is no clear sign that larger files produce higher receive times, it is clear that only higher receive times are coming from larger files. Does chunk size have a more of an influence here?

```
ggplot(data = rest) +
  geom_point(mapping = aes(x = chunkSize, y = receiveTime,color = fileSize), alpha = 0.2) +
  ggtitle("Receive Time vs Chunk Size")
```



It seems very clear here that larger files with smaller chunk sizes are connected to higher receiving times.

Data Modeling with Data Set 1

So far I have a hunch that we could build a model wherein an increase in chunk size should increase tag count and decrease response time logarithmically.

However, we have also observed that there should be a linear model between file size and tag count, so for this deliverable we will be limiting the scope to building and analyzing this.

We will start with breaking up our remaining data we separated from the test group into two new groups: validation and training. Our test set removed 20% of our data, and now we will put 60% into validation and the remaining 20% into test. We want 60% of our remaining 80%, so we will split this off again into 25% and 75%.

```
set.seed(1234)
train_rows <- as.vector(createDataPartition(rest$chunkSize, p = 0.75, list = FALSE))
train_rows
```

```
##   [1]   2   4   6   8   9  10  11  12  14  16  17  18  19  20  21  22  23  25
##  [19]  26  28  29  30  35  36  37  38  40  41  42  43  45  46  47  49  50  53
##  [37]  54  56  57  58  59  60  61  62  63  65  67  68  70  72  73  74  76  77
##  [55]  78  79  81  82  83  84  85  86  89  90  91  92  94  95  96  97 100 101
##  [73] 104 105 106 107 108 109 111 112 113 114 115 116 118 119 121 122 123 124
##  [91] 125 126 127 128 130 132 133 135 136 138 139 140 141 142 143 145 146 147
```

```
## [109] 149 150 151 152 153 155 156 158 159 160 161 162 163 164 165 166 167 171
## [127] 172 173 174 175 176 177 180 181 182 187 188 189 190 192 198 199 201 202
## [145] 203 204 205 207 208 209 211 212 213 214 216 217 218 220 221 224 226 227
## [163] 228 229 231 234 235 237 238 239 240 241 244 245 246 247 249 250 251 252
## [181] 254 256 258 260 261 262 265 269 270 271 272 273 274 275 277 278 280 281
## [199] 284 285 286 287 288 289 290 292 293 295 297 298 299 300 302 303 305 306
## [217] 308 309 310 311 312 313 314 317 320 321 323 325 326 328 329 330 331 333
## [235] 334 335 337 341 343 345 346 347 351 352 353 355 359 360 362 363 365 366
## [253] 367 369 370 371 374 375 377 378 379 380 381 382 383 384 386 387 388 389
## [271] 390 392 393 394 395 396 397 398 400 401 402 404 405 406 409 410 411 412
## [289] 413 414 415 416 417 419 421 422 423 424 425 426 427 428 429 431 433 435
## [307] 436 437 438 440 441 443 444 445 446 447 448 450 451 452 453 455 456 458
## [325] 459 460 461 462 463 464 465 466 467 468 469 470 471 472 474 476 477 478
## [343] 480 481 483 484 485 486 489 490 491 493 494 495 496 497 498 499 500 501
## [361] 502 503 504 505 506 507 509 510 511 512 513 516 518 519 520 521 523 524
## [379] 525 526 527 528 529 530 531 532 533 535 537 538 540 541 542 543 544 545
## [397] 546 547 548 550 551 552 554 555 556 557 558 559 562 563 564 565 567 570
## [415] 571 572 574 575 576 577 579 580 581 582 583 584 585 586 587 588 589 590
## [433] 591 592 593 594 595 596 597 598 599 600 601 602 604 605 606 608 609 610
## [451] 611 613 614 615 616 617 619 620 621 622 623 624 626 627 628 629 630 631
## [469] 632 633 636 637 638 639 640 641 642 643 644 645 646 647 648 649 651 652
## [487] 653 654 655 656 658 659 660 661 662 664 665 666 667 669 670 672 674 675
## [505] 676 679 680 681 682 683 684 686 687 688 689 690 691 693 694 695 697 698
## [523] 699 700 702 703 705 706 707 708 710 712 715 716 717 718 719 720 721 722
## [541] 723 725 727 729 730 732 733 734 735 737 739 740 743 744 745 746 747 748
## [559] 749 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 770
## [577] 771 772 777 778 779 780 781 782 783 784 785 786 788 789 790 791 792 794
## [595] 796 798 799 802 805 808 809 810 811 812 813 814 815 816 817 818 820 821
## [613] 822 823 824 825 828 829 830 832 833 835 836 839 840 842 843 844 846 847
## [631] 849 850 851 852 854 856 860 861 863 865 866 867 868 869 870 871 872 873
## [649] 874 877 878 879 880 881 882 883 884 885 886 888 889 890 891 892 894 895
## [667] 896 899 900 901 902 903 904 905 906 908 909 910 911 912 913 914 915 916
## [685] 917 918 919 920 921 922 923 925 926 927 928 929 930 931 932 933 934 936
```

```
validate <- rest[-train_rows, ]
train <- rest[train_rows, ]
```

Next I will create the linear model to predict tag counts based on knowing the file size.

```
model <- lm(tCount ~ fileSize, data = train)
```

This model can be used to make predictions based on the validation portion of the data we created earlier.

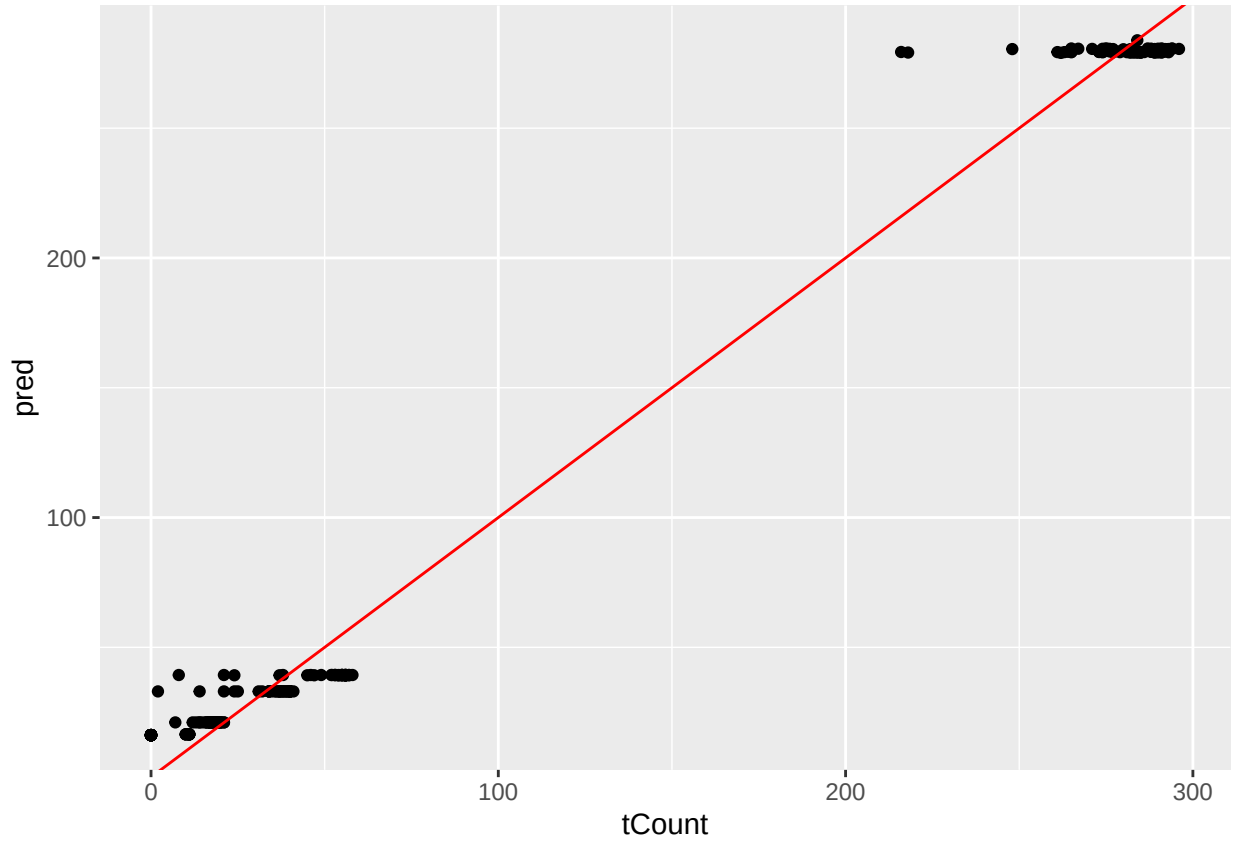
```
predictions <- add_predictions(validate, model)
predictions
```

```
## # A tibble: 234 x 11
##    source destination protocol chunkSize tCount fileSize retransmits connectTime
##    <chr>  <chr>       <chr>        <dbl>  <dbl>    <dbl>       <dbl>       <dbl>
##  1 192.1~ www.amazon~ HTTP/1.0         4      7     1263           2    0.00084
##  2 192.1~ www.amazon~ HTTP/1.1         5     12     1263           2    0.000463
##  3 192.1~ www.amazon~ HTTP/1.1         6     13     1263           2    0.000520
```

```
##  4 192.1~ www.amazon~ HTTP/1.0         8     15     1263           2    0.000452
##  5 192.1~ www.amazon~ HTTP/1.0        12     14     1263           2    0.000401
##  6 192.1~ www.amazon~ HTTP/1.0        13     14     1263           2    0.000465
##  7 192.1~ www.amazon~ HTTP/1.1        20     17     1263           2    0.000458
##  8 192.1~ www.amazon~ HTTP/1.0        24     17     1263           2    0.000477
##  9 192.1~ www.amazon~ HTTP/1.1        26     17     1263           2    0.000447
## 10 192.1~ www.amazon~ HTTP/1.0        27     16     1263           2    0.000422
## # ... with 224 more rows, and 3 more variables: requestTime <dbl>,
## #   receiveTime <dbl>, pred <dbl>
```

Let's graph and compare our tCount values with the predictions from our new model.

```
ggplot(data = predictions, mapping = aes(x = tCount, y = pred)) +
  geom_point() +
  geom_abline(intercept = 0, slope = 1, color = "red")
```



I also need to calculate goodness of fit measures for the model on the validation set.

```
R2(predictions$pred, predictions$tCount)
```

```
## [1] 0.9897537
```

```
MAE(predictions$pred, predictions$tCount)
```

```
## [1] 9.940797
```

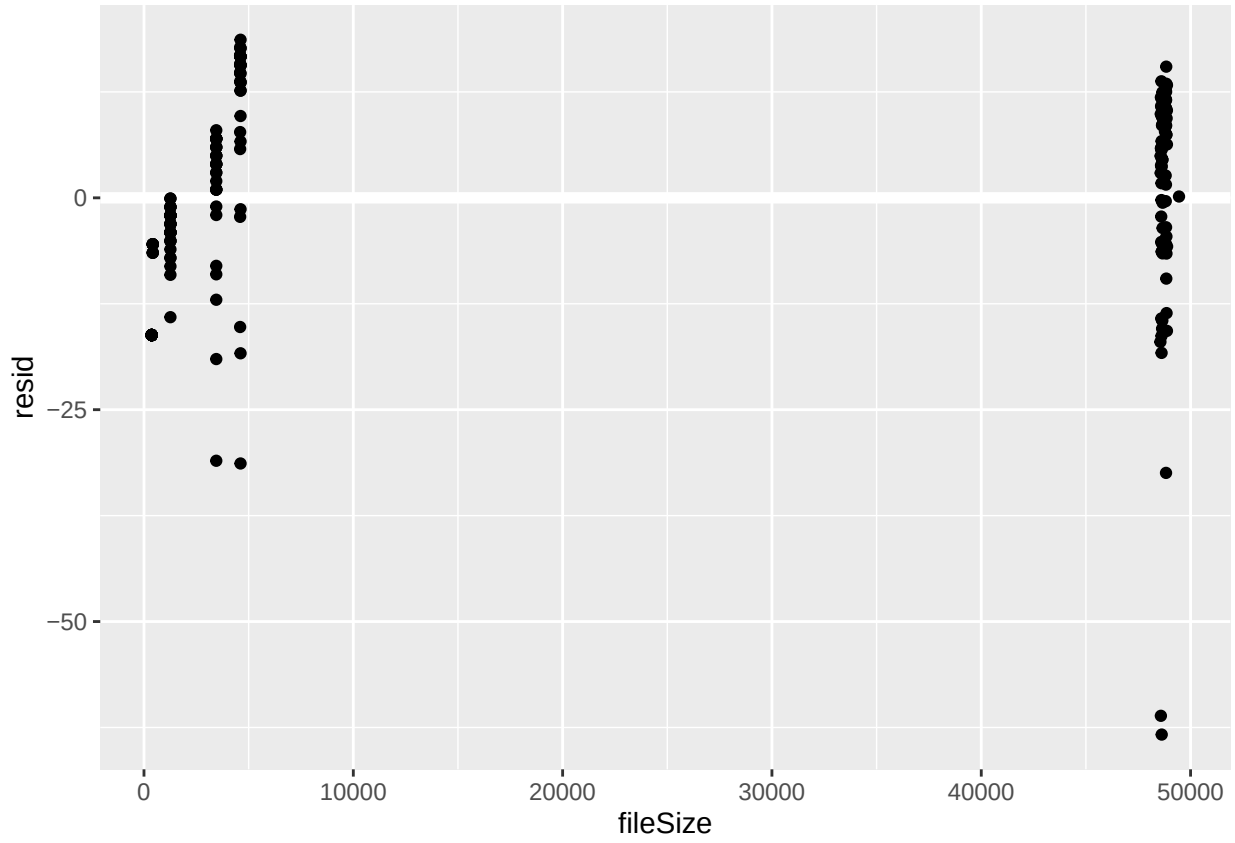
```
RMSE(predictions$pred, predictions$tCount)
```

```
## [1] 12.66555
```

Next I will calculate and plot the residual values.

```
resids <- add_residuals(validate, model)

ggplot(data = resids, mapping = aes(x = fileSize, y = resid)) +
  geom_ref_line(h = 0) +
  geom_point()
```



I would also like to compare this with a cross validation training approach.

```
train.control <- trainControl(method = "cv", number = 5)
model <- train(tCount ~ fileSize, data = rest, method = "lm",
               trControl = train.control)
model
```

```
## Linear Regression
##
## 936 samples
##   1 predictor
##
```

```
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 750, 749, 748, 748, 749
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   13.86025  0.9877894  10.20081
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

```
summary(model)
```

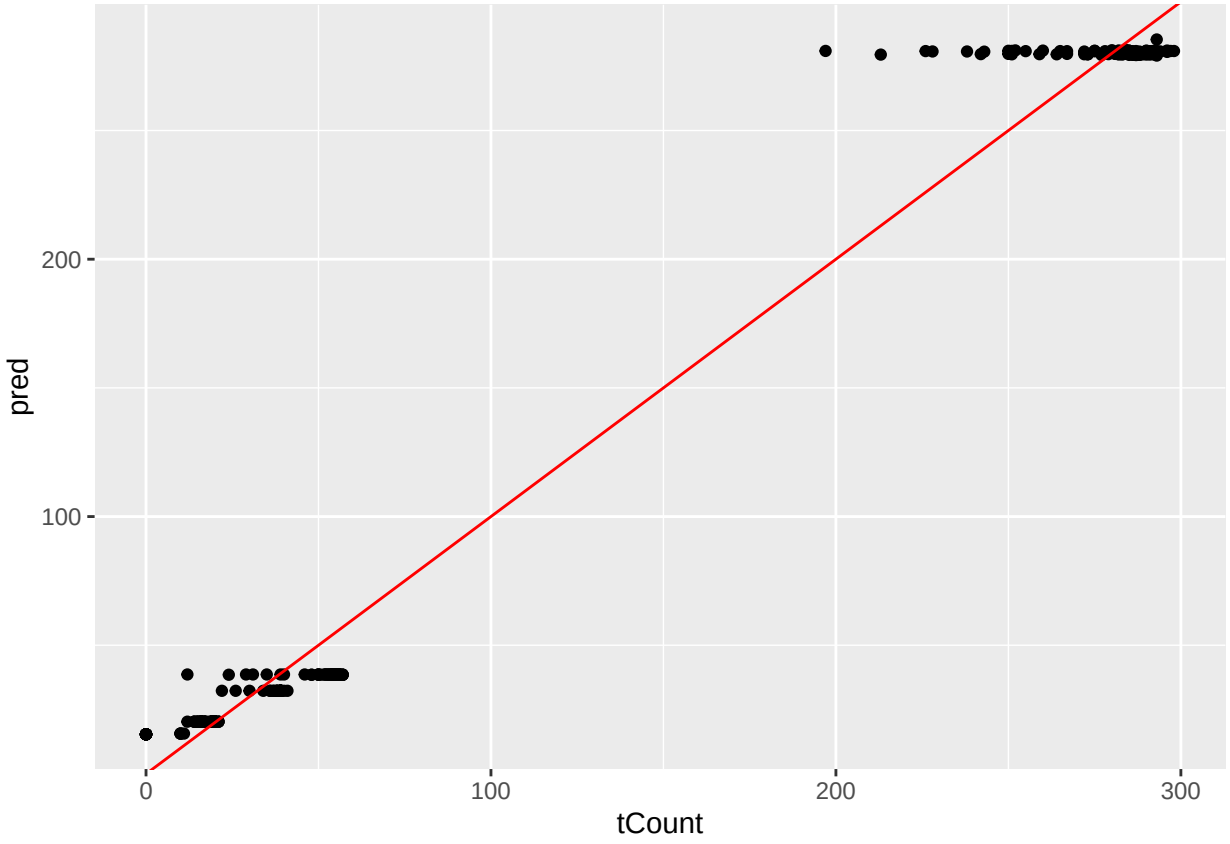
```
##
## Call:
## lm(formula = .outcome ~ ., data = dat)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -86.958  -5.681   1.700   9.458  19.354
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 1.339e+01  6.225e-01   21.52   <2e-16 ***
## fileSize    5.476e-03  1.999e-05  273.92   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 13.9 on 934 degrees of freedom
## Multiple R-squared:  0.9877, Adjusted R-squared:  0.9877
## F-statistic: 7.503e+04 on 1 and 934 DF,  p-value: < 2.2e-16
```

Finally I will examine with the test set.

```
predictions <- add_predictions(test, model)
predictions
```

```
## # A tibble: 234 x 11
##    source destination protocol chunkSize tCount fileSize retransmits connectTime
##    <chr>  <chr>       <chr>        <dbl>  <dbl>    <dbl>       <dbl>       <dbl>
##  1 192.1~ www.amazon~ HTTP/1.0         5     12     1263           2    0.000437
##  2 192.1~ www.amazon~ HTTP/1.1         7     14     1263           2    0.000457
##  3 192.1~ www.amazon~ HTTP/1.0        10     14     1263           2    0.000412
##  4 192.1~ www.amazon~ HTTP/1.1        10     14     1263           2    0.000443
##  5 192.1~ www.amazon~ HTTP/1.0        16     16     1263           2    0.000435
##  6 192.1~ www.amazon~ HTTP/1.0        17     15     1263           2    0.000466
##  7 192.1~ www.amazon~ HTTP/1.1        17     15     1263           2    0.000485
##  8 192.1~ www.amazon~ HTTP/1.0        18     15     1263           2    0.000454
##  9 192.1~ www.amazon~ HTTP/1.1        18     15     1263           2    0.000521
## 10 192.1~ www.amazon~ HTTP/1.1        19     16     1263           2    0.000462
## # ... with 224 more rows, and 3 more variables: requestTime <dbl>,
## #   receiveTime <dbl>, pred <dbl>
```

```
ggplot(data = predictions, mapping = aes(x = tCount, y = pred)) +
  geom_point() +
  geom_abline(intercept = 0, slope = 1, color = "red")
```




```
R2(predictions$pred, predictions$tCount)
```

```
## [1] 0.9852329
```

```
MAE(predictions$pred, predictions$tCount)
```

```
## [1] 10.75246
```

```
RMSE(predictions$pred, predictions$tCount)
```

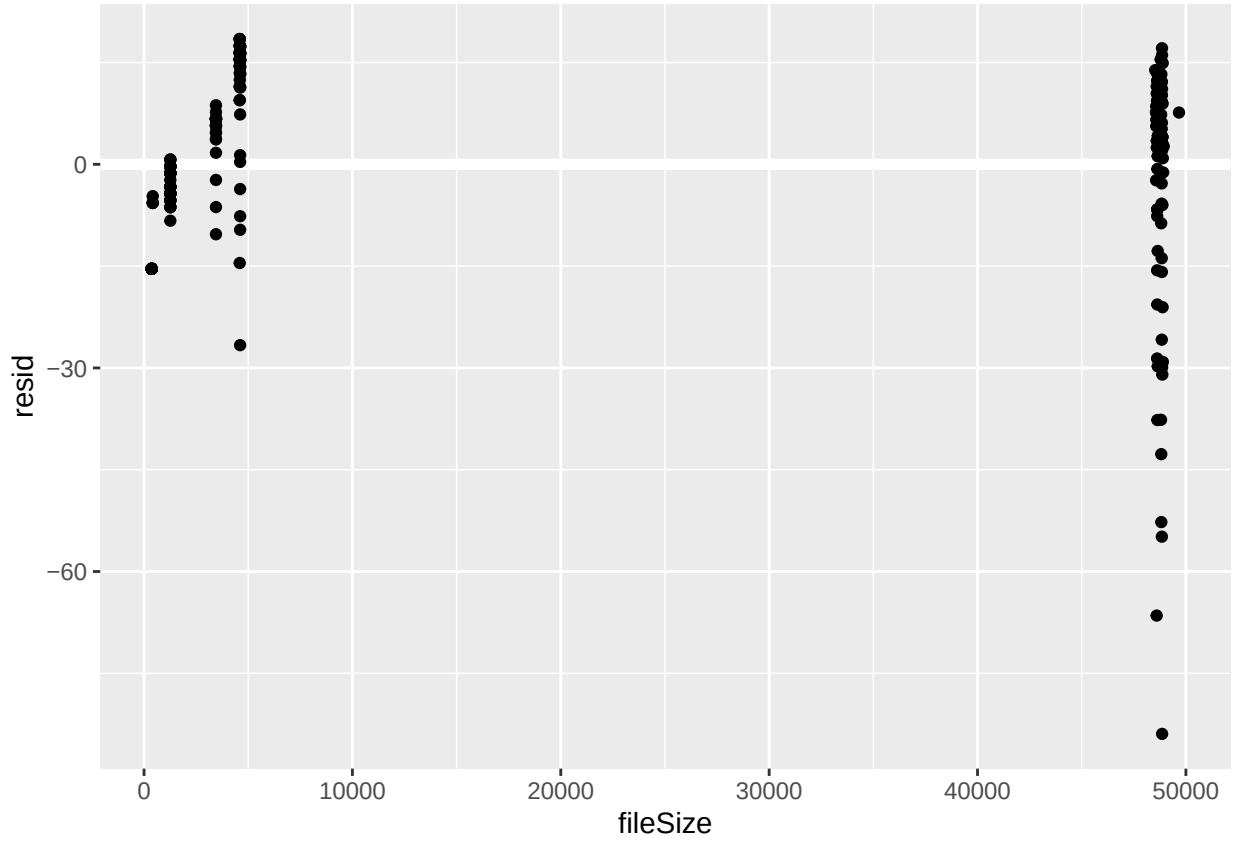
```
## [1] 14.88198
```

```
resids <- add_residuals(test, model)
resids
```

```
## # A tibble: 234 x 11
##    source destination protocol chunkSize tCount fileSize retransmits connectTime
##    <chr>  <chr>       <chr>        <dbl>  <dbl>    <dbl>       <dbl>       <dbl>
```

```
##  1 192.1~ www.amazon~ HTTP/1.0         5     12     1263           2    0.000437
##  2 192.1~ www.amazon~ HTTP/1.1         7     14     1263           2    0.000457
##  3 192.1~ www.amazon~ HTTP/1.0        10     14     1263           2    0.000412
##  4 192.1~ www.amazon~ HTTP/1.1        10     14     1263           2    0.000443
##  5 192.1~ www.amazon~ HTTP/1.0        16     16     1263           2    0.000435
##  6 192.1~ www.amazon~ HTTP/1.0        17     15     1263           2    0.000466
##  7 192.1~ www.amazon~ HTTP/1.1        17     15     1263           2    0.000485
##  8 192.1~ www.amazon~ HTTP/1.0        18     15     1263           2    0.000454
##  9 192.1~ www.amazon~ HTTP/1.1        18     15     1263           2    0.000521
## 10 192.1~ www.amazon~ HTTP/1.1        19     16     1263           2    0.000462
## # ... with 224 more rows, and 3 more variables: requestTime <dbl>,
## #   receiveTime <dbl>, resid <dbl>
```

```
ggplot(data = resids, mapping = aes(x = fileSize, y = resid)) +
  geom_ref_line(h = 0) +
  geom_point()
```



Initial Modeling and Testing Observations:

All goodness of fit measurements scored very well. However, the range of residuals and general gap in fileSize is still a concern.

Initial Data Set 2 Draft

During the experiment, while the program ran I also collected packet capture data from Wireshark and exported the data to JSON. As previously, I ran a separate bash script for each destination / web site. The

end result was 5 separate JSON files which I will now attempt to import the data into R and combine.

```
netdat <- fromJSON("amazon.json",simplifyVector = TRUE,flatten = TRUE)
netdat <- as_tibble(netdat)

names(netdat)<-make.names(names(netdat),unique = TRUE)

netdat <- tibble(site="www.amazon.com",frame=netdat$X_source.layers.frame.frame.number,seq=netdat$X_sou

netdatc <- netdat

netdat <- fromJSON("facebook.json",simplifyVector = TRUE,flatten = TRUE)
netdat <- as_tibble(netdat)

names(netdat)<-make.names(names(netdat),unique = TRUE)

netdat <- tibble(site="www.facebook.com",frame=netdat$X_source.layers.frame.frame.number,seq=netdat$X_s

netdatc <- bind_rows(netdatc,netdat)

netdat <- fromJSON("google.json",simplifyVector = TRUE,flatten = TRUE)
netdat <- as_tibble(netdat)

names(netdat)<-make.names(names(netdat),unique = TRUE)

netdat <- tibble(site="www.google.com",frame=netdat$X_source.layers.frame.frame.number,seq=netdat$X_sou

netdatc <- bind_rows(netdatc,netdat)

netdat <- fromJSON("yahoo.json",simplifyVector = TRUE,flatten = TRUE)
netdat <- as_tibble(netdat)

names(netdat)<-make.names(names(netdat),unique = TRUE)

netdat <- tibble(site="www.yahoo.com",frame=netdat$X_source.layers.frame.frame.number,seq=netdat$X_sour

netdatc <- bind_rows(netdatc,netdat)

netdat <- fromJSON("youtube.json",simplifyVector = TRUE,flatten = TRUE)
netdat <- as_tibble(netdat)

names(netdat)<-make.names(names(netdat),unique = TRUE)

netdat <- tibble(site="www.youtube.com",frame=netdat$X_source.layers.frame.frame.number,seq=netdat$X_so

netdat <- bind_rows(netdatc,netdat)


transmute(netdat, frame = as.double(frame),seq = as.double(seq),nxtseq = as.double(nxtseq),time = as.dou
```

```
## # A tibble: 32,758 x 8
##    frame   seq nxtseq      time   ttl srcport dstport   len
##    <dbl> <dbl>  <dbl>     <dbl> <dbl>   <dbl>   <dbl> <dbl>
##  1     6     0      0 0.000393     64   57402      80     0
```

```
##  2     7     0      0 0.0194      128      80   57402     0
##  3     8     1      1 0.0000410    64   57402      80     0
##  4     9     1     20 0.0000777    64   57402      80    19
##  5    10     1      1 0.000374    128      80   57402     0
##  6    11     1   1265 0.0221      128      80   57402  1263
##  7    12    20     20 0.00148      64   57402      80     0
##  8    13    20     20 0.000436     64   57402      80     0
##  9    14  1265   1265 0.000323    128      80   57402     0
## 10    15     0      0 0.00129      64   57404      80     0
## # ... with 32,748 more rows
```

```
head(unique(netdat$site),n=20)
```

```
## [1] "www.amazon.com"   "www.facebook.com" "www.google.com"   "www.yahoo.com"
## [5] "www.youtube.com"
```

```
head(unique(netdat$frame),n=20)
```

```
##  [1] "6"  "7"  "8"  "9"  "10" "11" "12" "13" "14" "15" "17" "18" "19" "20" "21"
## [16] "22" "23" "24" "25" "26"
```

```
head(unique(netdat$seq),n=20)
```

```
##  [1] "0"     "1"     "20"    "1265"  "1264"  "420"   "421"   "368"   "3454"
## [10] "3455"  "1741"  "367"   "3133"  "3446"  "349"   "2054"  "2861"  "14301"
## [19] "17161" "18591"
```

```
head(unique(netdat$nxtseq),n=20)
```

```
##  [1] "0"     "1"     "20"    "1265"  "1264"  "420"   "421"   "368"   "3454"
## [10] "3455"  "1741"  "367"   "3133"  "3446"  "349"   "2054"  "2861"  "14301"
## [19] "17161" "18591"
```

```
head(unique(netdat$time),n=20)
```

```
##  [1] "0.000393454" "0.019393696" "0.000041033" "0.000077729" "0.000374425"
##  [6] "0.022139966" "0.001484397" "0.000436187" "0.000322767" "0.001289815"
## [11] "0.014240829" "0.000033560" "0.000068435" "0.000201422" "0.022059671"
## [16] "0.001691084" "0.000248701" "0.000243144" "0.001252079" "0.018761084"
```

```
head(unique(netdat$ttl),n=20)
```

```
## [1] "64"  "128"
```

```
head(unique(netdat$src),n=20)
```

```
##  [1] "192.168.198.131" "65.8.171.9"      "104.84.224.106"  "69.171.250.35"
##  [5] "172.217.164.100" "172.217.6.36"    "142.250.72.196"  "35.224.99.156"
##  [9] "74.6.231.21"     "74.6.231.20"     "74.6.231.19"     "98.137.11.164"
## [13] "172.217.5.110"   "142.250.72.206"  "172.217.6.46"    "172.217.164.110"
## [17] "216.58.194.206"  "216.58.194.174"
```

```
head(unique(netdat$dst),n=20)
```

```
##  [1] "65.8.171.9"      "192.168.198.131" "104.84.224.106"  "69.171.250.35"
##  [5] "172.217.164.100" "172.217.6.36"    "142.250.72.196"  "35.224.99.156"
##  [9] "74.6.231.21"     "74.6.231.20"     "74.6.231.19"     "98.137.11.164"
## [13] "172.217.5.110"   "142.250.72.206"  "172.217.6.46"    "172.217.164.110"
## [17] "216.58.194.206"  "216.58.194.174"
```

```
head(unique(netdat$srcport),n=20)
```

```
##  [1] "57402" "80"    "57404" "57406" "57408" "57410" "57412" "57414" "57416"
## [10] "57418" "57420" "57422" "57424" "57426" "57428" "57430" "57432" "57434"
## [19] "57436" "57438"
```

```
head(unique(netdat$dstport),n=20)
```

```
##  [1] "80"    "57402" "57404" "57406" "57408" "57410" "57412" "57414" "57416"
## [10] "57418" "57420" "57422" "57424" "57426" "57428" "57430" "57432" "57434"
## [19] "57436" "57438"
```

```
head(unique(netdat$len),n=20)
```

```
##  [1] "0"    "19"   "1263" "418"  "419"  "366"  "3453" "1740" "1713" "3132"
## [11] "321"  "3445" "8"    "348"  "2784" "3105" "1392" "2053" "1400" "2860"
```

Of the 423 Wireshark variables captured, I have found ten I would like to work with for the second set.

Continuous Variables

frame (Frame Number): Identification of frame. Note some frame values will be missing because only TCP frames are captured. Other frames such as DNS lookup and miscellaneous traffic have been filtered out.

time (Frame Time Delta): Time in seconds to start and end frame.

seq (TCP Sequence Number): Starting byte of fragment/sequence.

nxtseq (TCP Next Sequence Number): Ending byte of fragment/sequence pointing to start of next fragment/sequence.

ttl (IP Time to Live): Maximum allowed hops (router/routes) over the internet between the source and destination.

len (TCP Data Length): Size of data transferred in TCP packet.

Categorical

src (IP Source): Internet address where data is being transferred from.

dst (IP Destination): Internet address where data is being transferred to.

srcport (TCP Source Port): Network port where data is being transferred from.

dstport (TCP Destination Port): Network port where data is being transferred to.

site (Website): Name of website corresponding to .json Wireshark capture data.

Data Set 2 reflects the same file transfers that occurred to create Data Set 1. However, so far there is no clear and easy hook to corss reference the two. The most important variable in the experiment, 'chunkSize', is not

reflected in Wireshark because it is a product of the C program's method of processing socket information. Wireshark can only capture data passed to and from the computer to the websites and chunkSize is not passed outside of the C program.

What we will try to do to tie the two together in the next deliverable is to match Data Set 2's site with Data Set 1's destination(website). Frames are processed sequentially in the same order that the chunkSize was sent out, so once we identify distinct sets of frames as a single transaction of the program we will be able to map these together.

The difficult will be in filtering out all the sequence and next sequence numbers and keeping track of each new file transfer to correspond with Data Set #1's chunkSize. We should have this process completed for Deliverable 3.

With this additional information tied in we should be able to explore if other network factors can be incorporated to improve our modeling.

Data Science Questions

Can we predict the number of tags that a website's root file based on the total file size? Based on the limited collection of sites so far it seems as though this is possible with a simple linear model.

How does the size of chunk parsing impact response time and tag count? Our data observations in deliverable 2 lead me to believe I can create a logarithmic model using chunkSize to predict tCount and receiveTime. This will be a part of Deliverable 3. The ultimate question would be can we use this model to help optimise chunkSize usage? If the model is known then the C program could incorporate it into its functionality, processing at an optimal chunkSize to browse web files.

Ethical Implications

Opacity: While the data collection is relatively straightforward and my code is available for inspection to recreate my experiment, an understanding of C programming is required for scrutiny. The learning curve is rudimentary however. Wireshark is a very complicated program, and knowing about network functionality is helpful in understanding these models.

Scale: Although my experiment can be reran to include other protocols and a much larger number of websites, the R modeling here would need to be adjusted and reconsidered for such a larger scale.

For future deliveribles, I would like to tie in my C program's experiment with similar file transfer captured by Wireshark.

The process would be as follows: C Program Captures to CSV (deliverable 1 and 2) Wireshark Captures to JSON (deliverable 2) C Program Data is Cross Referenced to Wireshark (Deliverable 3) Multiple Web Browsers (Chrome, Firefox, Edge) Collect Root of same Sites (Deliverable 3) Wireshark Captures to JSON of Browsers Data (Deliverable 3) Web Browser and C Program are Cross Referenced for Analysis and Modeling

Damage: Benchmarks results could be misinterpreted if compared to newer versions of protocols for web file transfer. HTTP/2 was introduced in 2015 and is currently used by 7% of web browsers. Unlike HTTP/1.0 and 1.1, this new protocol allows synchronous communication through web sockets for two way simulanteous communication. HTTP/3 is an upcoming standard that is currently only in use by Safari 14 introduced in September 2020, though over 10 million websites have implemented support for this newer standard. Unlike HTTP/2 and HTTP 1.0 and 1.1, HTTP/3 uses a new transport protocol called QUIC (general purpose transfer protocol) instead of the previous TCP (transfer control protocol). Future experiments using modern browser test could include HTTP/2 and HTTP/3 comparisons.