LoL eSports: Data Wrangling Report

by Ryan Transfiguracion

Before we start explaining the wrangling process, here are the "final" desired data frames for the NALCS 2018 Spring Split that we'll be using for later statistical analyses and models. Actually, the first three datasets will be the "templates" from which all future analyses and datasets will derive, such as the next three datasets on the list. Click on the each link to see its corresponding CSV file:

Entire Split Match-by-Match Team Totals

Entire Split Match-by-Match Player Totals

Entire Split Match-by-Match Champ Bans

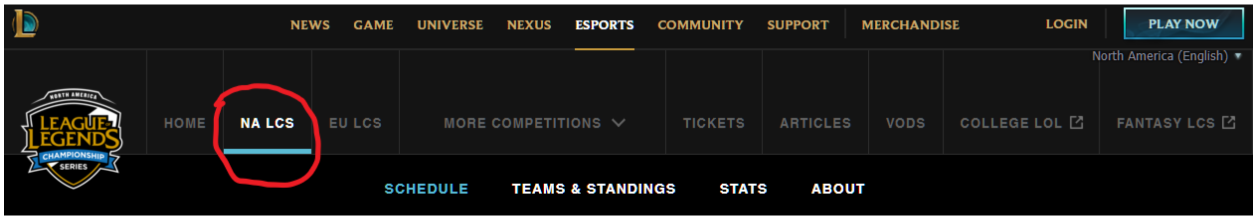
Regular Season Team Cumulative Totals

Regular Season Team Averages

Regular Season Opposing Team Cumulative Totals

Obtaining the Data for Making API Requests

All the match identifier data was obtained from lolesports.com. For eample, in order to access a NALCS 2018 Spring Split match, click on the NA LCS link as shown below:



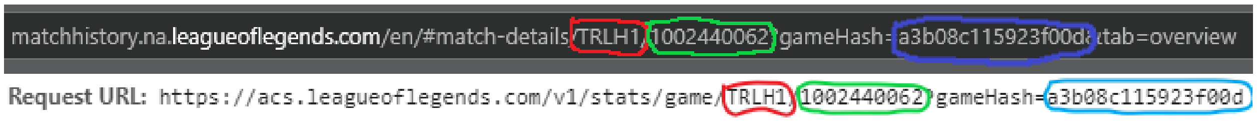
Then, slightly change the URL to this address: https://www.lolesports.com/en_US/na-lcs/na_2018_spring/schedule/regular_season/1 (presently, change the URL section saying "na_2018_summer" to "na_2018_spring"). Now, on the page, click on a single match. This will lead to that match's page which will look like this:

From here, click on the button near the bottom of the page that says "VIEW PLAYER BUILDS & GAME ANALYSIS". This will open a new Tab/Window on your browser for a page showing an overview of the match's statistics:

The URL of this page contains some important information that will be needed in order to call a Web API for this match. In fact, the Web API call can be found by opening the Web Inspector for this page (typically by pressing Ctrl+Shift+I or Command+Shift+I).

First, click on the Network tab of the inspector. Then, we must refresh the page in order to see the list of web requests. Eventually, we will find the request needed:

The encircled Request URL is, in fact, the Web API call we use in our R script in order to obtain the JSON data for this particular match. Every single match's Web API URI uses the same prefix: https://acs.leagueoflegends.com/v1/stats/game/. Each match also has its own unique identifiers, as shown the browser URL and web inspector screenshots below:



matchhistory.na.leagueoflegends.com/en/#match-details/TRLH1/1002440062?gameHash=a3b08c115923f00d&tab=overview

Request URL: https://acs.leagueoflegends.com/v1/stats/game/TRLH1/1002440062?gameHash=a3b08c115923f00d

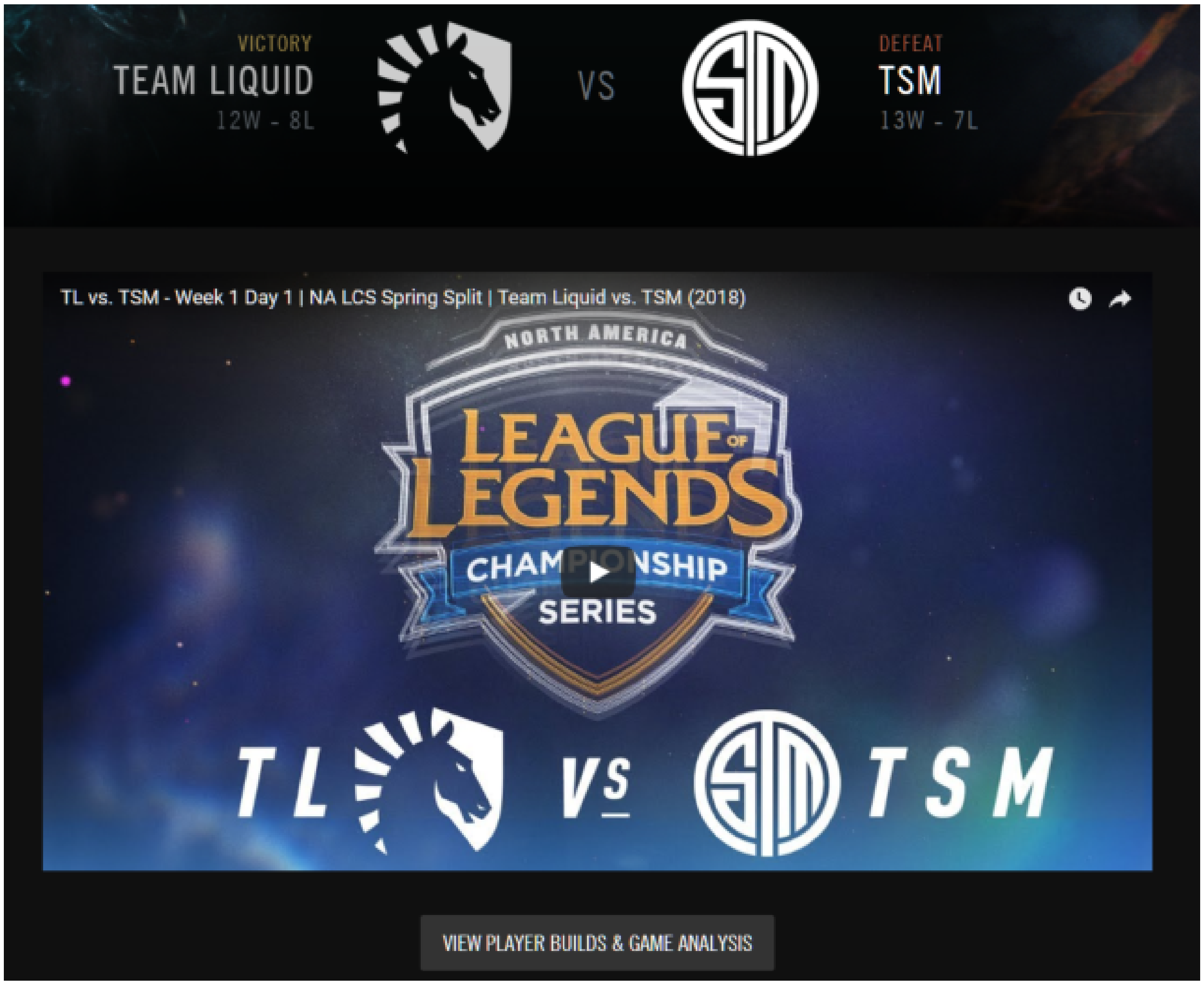
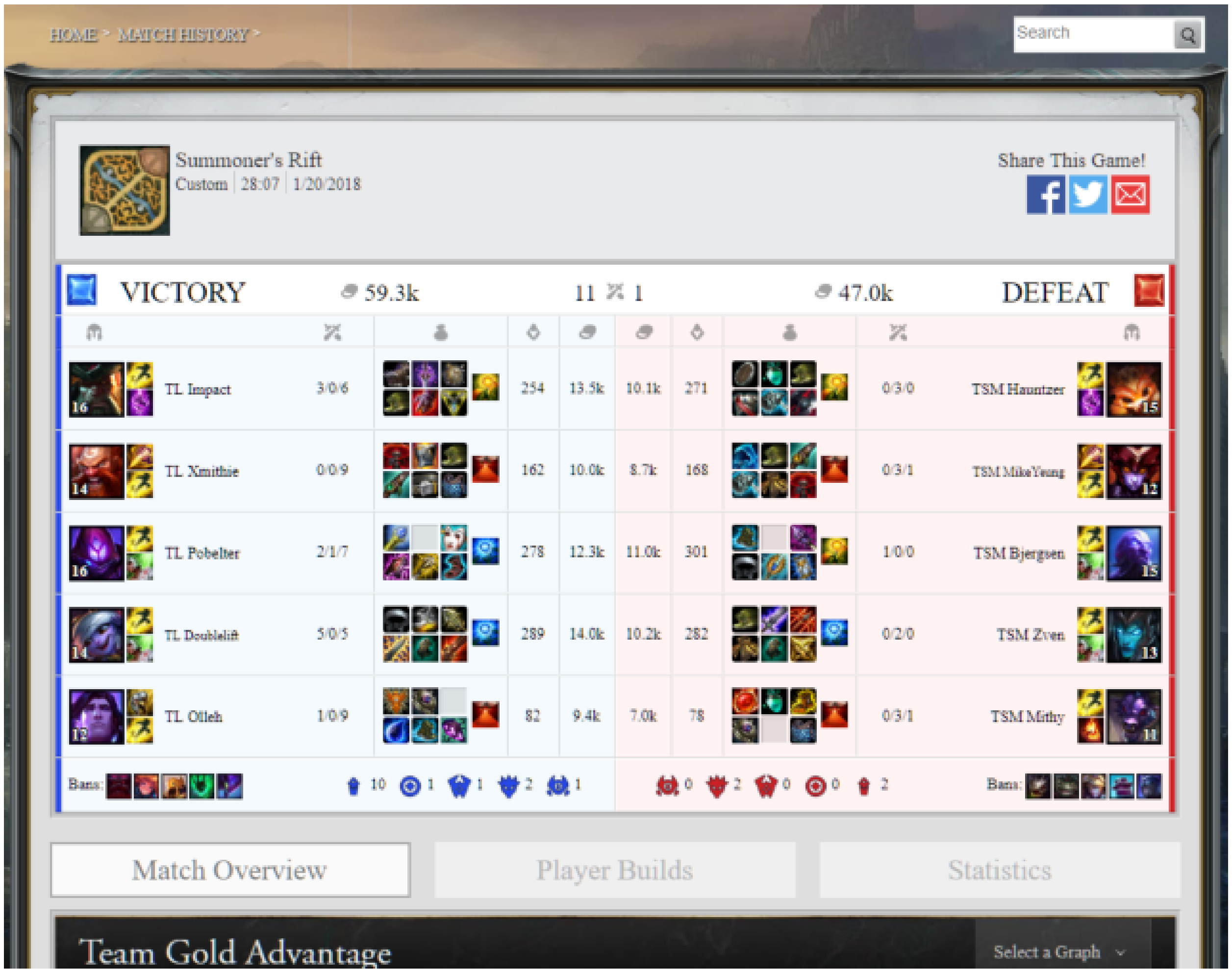

VICTORY
TEAM LIQUID
12W - 8L
VS
DEFEAT
TSM
13W - 7L
TL vs. TSM - Week 1 Day 1 | NA LCS Spring Split | Team Liquid vs. TSM (2018)
NORTH AMERICA
LEAGUE OF LEGENDS
CHAMPIONSHIP
SERIES
TL VS TSM
VIEW PLAYER BUILDS & GAME ANALYSIS

Figure 1: Match Page



HOME > MATCH HISTORY >

Search

Summoner's Rift
Custom | 28:07 | 1/20/2018

Share This Game!

VICTORY		59.3k		11	1		47.0k		DEFEAT
TL Impact	3/0/6		254	13.5k	10.1k	271		0/3/0	TSM Hauntzer
TL Xmithie	0/0/9		162	10.0k	8.7k	168		0/3/1	TSM MikeYeung
TL Pobelter	2/1/7		278	12.3k	11.0k	301		1/0/0	TSM Bjergsen
TL Doublelift	5/0/5		289	14.0k	10.2k	282		0/2/0	TSM Zven
TL Olleh	1/0/9		82	9.4k	7.0k	78		0/3/1	TSM Mithy

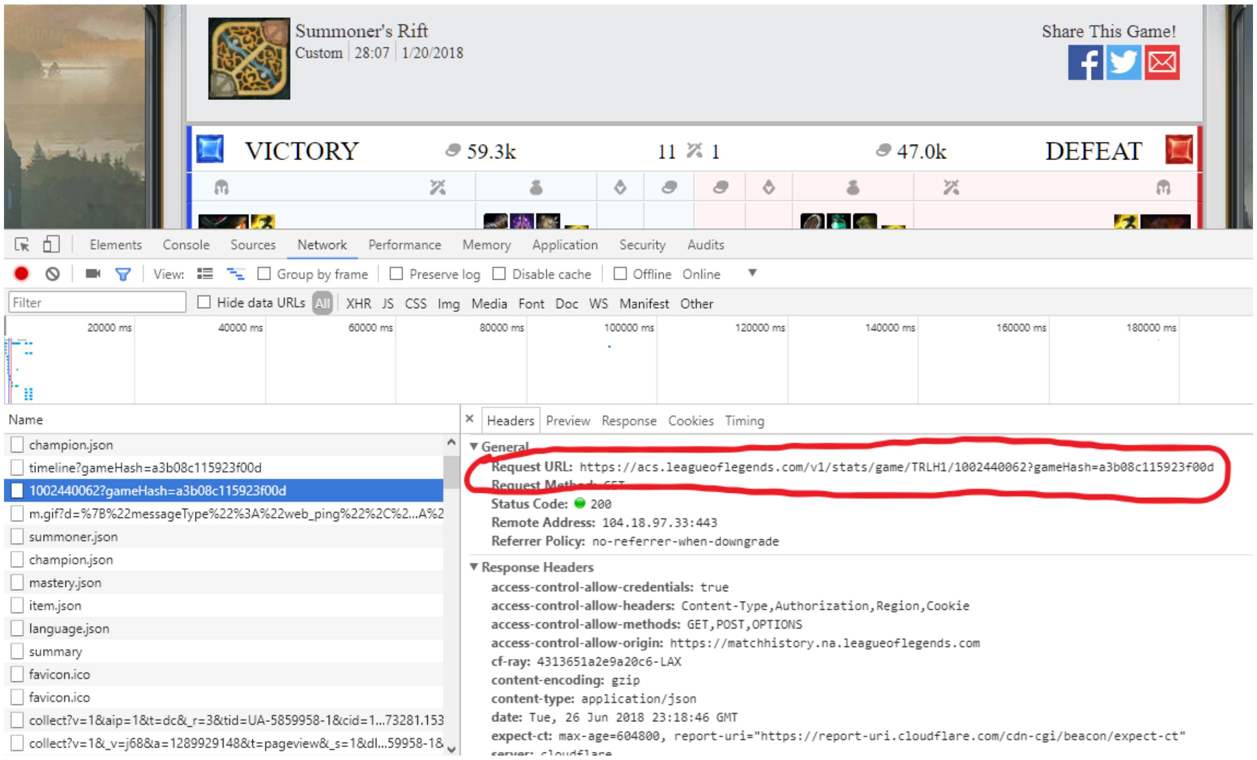
Bans: 10 1 1 2 1 | 0 2 0 0 2 Bans:

Match Overview | Player Builds | Statistics

Team Gold Advantage

Select a Graph

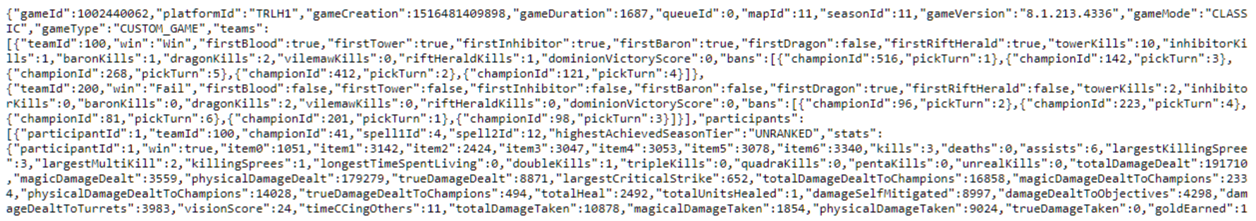
Figure 2: Match Stats Page



```
Name
champion.json
timeline?gameHash=a3b08c115923f00d
1002440062?gameHash=a3b08c115923f00d
m.gif?d=%7B%22messageType%22%3A%22web_ping%22%2C%2...A%2
summoner.json
champion.json
mastery.json
item.json
language.json
summary
favicon.ico
favicon.ico
collect?v=1&aip=1&t=dc&_r=3&tid=UA-5859958-1&cid=1...73281.153
collect?v=1&_v=j68&a=1289929148&t=pageview&_s=1&dl...59958-1&
```

```
▼ General
  Request URL: https://acs.leagueoflegends.com/v1/stats/game/TRLH1/1002440062?gameHash=a3b08c115923f00d
  Request Method: GET
  Status Code: ● 200
  Remote Address: 104.18.97.33:443
  Referrer Policy: no-referrer-when-downgrade
▼ Response Headers
  access-control-allow-credentials: true
  access-control-allow-headers: Content-Type,Authorization,Region,Cookie
  access-control-allow-methods: GET,POST,OPTIONS
  access-control-allow-origin: https://matchhistory.na.leagueoflegends.com
  cf-ray: 4313651a2e9a20c6-LAX
  content-encoding: gzip
  content-type: application/json
  date: Tue, 26 Jun 2018 23:18:46 GMT
  expect-ct: max-age=604800, report-uri="https://report-uri.cloudflare.com/cdn-cgi/beacon/expect-ct"
  server: cloudflare
```

Figure 3: Web Inspector



```
{"gameId":1002440062,"platformId":"TRLH1","gameCreation":1516481409898,"gameDuration":1687,"queueId":0,"mapId":11,"seasonId":11,"gameVersion":"8.1.213.4336","gameMode":"CLASS
IC","gameType":"CUSTOM_GAME","teams":
[{"teamId":100,"win":"Win","firstBlood":true,"firstTower":true,"firstInhibitor":true,"firstBaron":true,"firstDragon":false,"firstRiftHerald":true,"towerKills":10,"inhibitorKi
lls":1,"baronKills":1,"dragonKills":2,"vilemawKills":0,"riftHeraldKills":1,"dominionVictoryScore":0,"bans":[{"championId":516,"pickTurn":1},{"championId":142,"pickTurn":3},
{"championId":268,"pickTurn":5},{"championId":412,"pickTurn":2},{"championId":121,"pickTurn":4}]},
{"teamId":200,"win":"Fail","firstBlood":false,"firstTower":false,"firstInhibitor":false,"firstBaron":false,"firstDragon":true,"firstRiftHerald":false,"towerKills":2,"inhibito
rKills":0,"baronKills":0,"dragonKills":2,"vilemawKills":0,"riftHeraldKills":0,"dominionVictoryScore":0,"bans":[{"championId":96,"pickTurn":2},{"championId":223,"pickTurn":4},
{"championId":81,"pickTurn":6},{"championId":201,"pickTurn":1},{"championId":98,"pickTurn":3}]}],"participants":
[{"participantId":1,"teamId":100,"championId":41,"spell1Id":4,"spell2Id":12,"highestAchievedSeasonTier":"UNRANKED","stats":
{"participantId":1,"win":true,"item0":1051,"item1":3142,"item2":2424,"item3":3047,"item4":3053,"item5":3078,"item6":3340,"kills":3,"deaths":0,"assists":6,"largestKillingSpree
":3,"largestMultiKill":2,"killingSprees":1,"longestTimeSpentLiving":0,"doubleKills":1,"tripleKills":0,"quadraKills":0,"pentaKills":0,"unrealKills":0,"totalDamageDealt":191710
,"magicDamageDealt":3559,"physicalDamageDealt":179279,"trueDamageDealt":8871,"largestCriticalStrike":652,"totalDamageDealtToChampions":16858,"magicDamageDealtToChampions":233
4,"physicalDamageDealtToChampions":14028,"trueDamageDealtToChampions":494,"totalHeal":2492,"totalUnitsHealed":1,"damageSelfMitigated":8997,"damageDealtToObjectives":4298,"dam
ageDealtToTurrets":3983,"visionScore":24,"timeCCingOthers":11,"totalDamageTaken":10878,"magicalDamageTaken":1854,"physicalDamageTaken":9024,"trueDamageTaken":0,"goldEarned":1
```

Figure 4: Example JSON response

Now that we know how to obtain the identifiers for each match, we can open a spreadsheet program and enter these identifiers, as well as the team names and flags indicating whether the match was a tiebreaker or playoff, into individual rows. We can then save this spreadsheet as a CSV file, which will look like this when we import the file in R:

```r
library(knitr)
kable(head(read.csv("../gameid_data/NALCS_Spring2018.csv")))
```

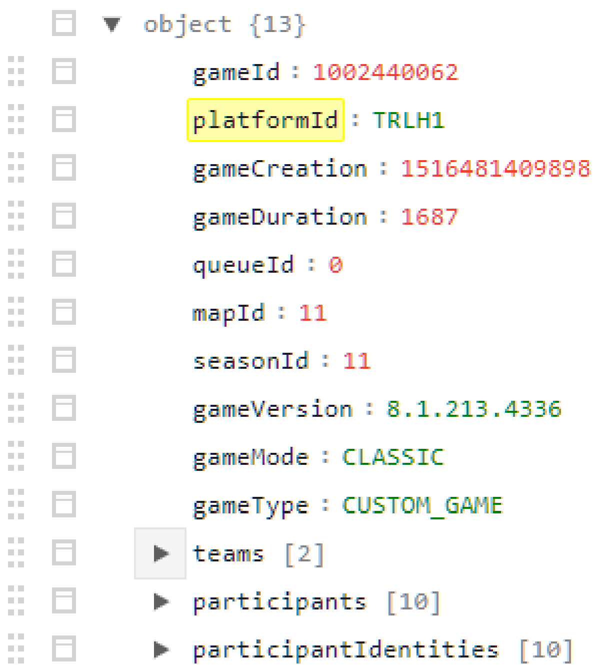
Region.ID	Game.ID	Hash.ID	Blue.Team	Red.Team	Tiebreaker	Playoff
TRLH1	1002440062	a3b08c115923f00d	Team Liquid	Team Solo Mid	FALSE	FALSE
TRLH1	1002440076	c426d3d50426edb3	100 Thieves	OpTic Gaming	FALSE	FALSE
TRLH1	1002440084	f0f86e52b6e472e9	Clutch Gaming	Golden Guardians	FALSE	FALSE
TRLH1	1002440095	fd3b9331ff5312e3	Echo Fox	FlyQuest	FALSE	FALSE
TRLH1	1002440106	d6441d4ec8f87534	Counter Logic Gaming	Cloud9	FALSE	FALSE
TRLH1	1002440127	361bcc2e848641d2	OpTic Gaming	Team Liquid	FALSE	FALSE

As stated previously, the beginning string for the URL of the Web API for obtaining the match data JSON response is:

https://acs.leagueoflegends.com/v1/stats/game/

If we were to use the example match, the full string for this Web API is:

https://acs.leagueoflegends.com/v1/stats/game/TRLH1/1002440062?gameHash=a3b08c115923f00d



```
▼ object {13}
    gameId : 1002440062
    platformId : TRLH1
    gameCreation : 1516481409898
    gameDuration : 1687
    queueId : 0
    mapId : 11
    seasonId : 11
    gameVersion : 8.1.213.4336
    gameMode : CLASSIC
    gameType : CUSTOM_GAME
  ▶ teams [2]
  ▶ participants [10]
  ▶ participantIdentities [10]
```

Figure 5: Pretty JSON response

In the block above, if we use the example match again, `acs_prefix_domain` is https://acs.leagueoflegends.com, `chr_platform_id` is TRLH1, `num_match_id` is 1002440062, and `chr_game_hash` is a3b08c115923f00d.

With the `uri` constructed, we use this string to make the API request within a custom function, `process_uri()`:

```r
process_uri <- function(str_uri) {
  response <- httr::GET(str_uri)
  while (response$status_code == 429) {
    Sys.sleep(2)
    response <- httr::GET(str_uri)
  }
  json <- jsonlite::fromJSON(content(response, as = "text"))
  return(json)
}
```

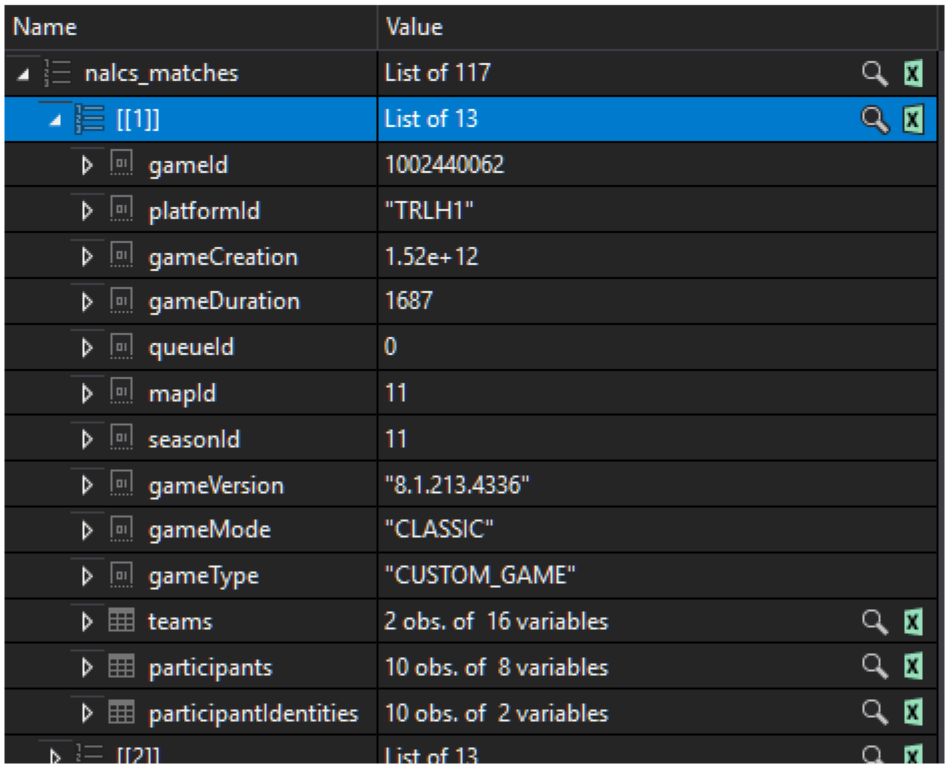
Note the statement `while (response$status_code == 429)`. The server to which we are making this request restricts the user to making a certain limited number of requests within a certain timeframe (unknown to this author). If we go beyond this restriction during a request, the response will return a status code of 429, and we wouldn't get the data from that match. Therefore, we are watching for this response status code, and if we get a 429, then we sleep the R runtime for two seconds before making the same request again. Theoretically, we should get the desired 200 response code and the match data the second time around.

Obtaining Match Data for Multiple Matches

First, we import the CSV file that we created, which contains all the match identifiers:

```r
# NA LCS 2018 Spring Split -- Regular Season, Tiebreakers, and Playoffs
nalcs_matchid_df <- read.csv("gameid_data/NALCS_Spring2018.csv")
```

Next, we, call the custom function that makes a request for each of the 117 NALCS 2018 Spring Split matches



Name	Value
nalcs_matches	List of 117
[[1]]	List of 13
gameId	1002440062
platformId	"TRLH1"
gameCreation	1.52e+12
gameDuration	1687
queueId	0
mapId	11
seasonId	11
gameVersion	"8.1.213.4336"
gameMode	"CLASSIC"
gameType	"CUSTOM_GAME"
teams	2 obs. of 16 variables
participants	10 obs. of 8 variables
participantIdentities	10 obs. of 2 variables
[[2]]	List of 13

Figure 6: Single Match Data Object

and then returns a List object containing the 117 match datas:

```r
nalcs_matches <- get_league_match_data_list(nalcs_matchid_df)
```

Here is the custom function, `get_league_match_data_list()`:

```r
get_league_match_data_list <- function(league_matchid_df) {
  matchlist <- list()
  for (i in 1:nrow(league_matchid_df)) {
    matchlist[[i]] <- get_acs_match_by_matchid(league_matchid_df$Region.ID[[i]],
     league_matchid_df$Game.ID[[i]],
     chr_game_hash = league_matchid_df$Hash.ID[[i]])
  }
  return(matchlist)
}
```

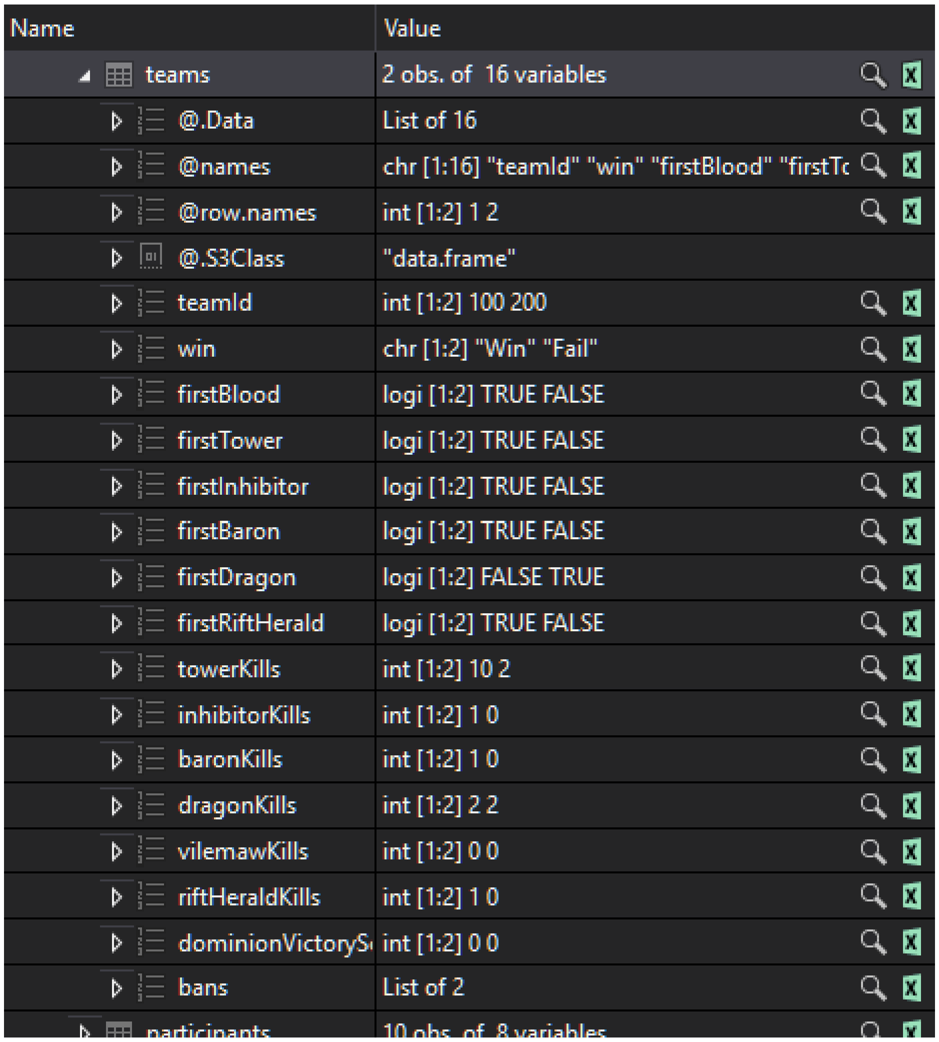
Looking Deeper into Match Data

Now that we have match data for all 117 matches in the NALCS 2018 Spring Split, let's look at Figure 6, which represents a single match.

Here, we can see the `gameId` and `platformId` strings that we used in the Web API call. We also see the `gameDuration` numeric, which we will need for our "final" datasets. However, the meat of our datasets are coming from the three data frames at the bottom: `teams`, `participants`, and `participantIdentities`. Let's look at `teams` first in Figure 7.

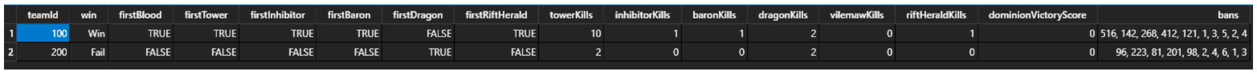
Also, let's look at the it in preview mode in Figure 8.

Before concatenating this into a cumulative data frame, this will get cleaned up in order to be useful. The



Name	Value
teams	2 obs. of 16 variables
@.Data	List of 16
@names	chr [1:16] "teamId" "win" "firstBlood" "firstTc
@row.names	int [1:2] 1 2
@.S3Class	"data.frame"
teamId	int [1:2] 100 200
win	chr [1:2] "Win" "Fail"
firstBlood	logi [1:2] TRUE FALSE
firstTower	logi [1:2] TRUE FALSE
firstInhibitor	logi [1:2] TRUE FALSE
firstBaron	logi [1:2] TRUE FALSE
firstDragon	logi [1:2] FALSE TRUE
firstRiftHerald	logi [1:2] TRUE FALSE
towerKills	int [1:2] 10 2
inhibitorKills	int [1:2] 1 0
baronKills	int [1:2] 1 0
dragonKills	int [1:2] 2 2
vilemawKills	int [1:2] 0 0
riftHeraldKills	int [1:2] 1 0
dominionVictorySc	int [1:2] 0 0
bans	List of 2
participants	10 obs. of 8 variables

Figure 7: Single Match Teams Data Object



	teamId	win	firstBlood	firstTower	firstInhibitor	firstBaron	firstDragon	firstRiftHerald	towerKills	inhibitorKills	baronKills	dragonKills	vilemawKills	riftHeraldKills	dominionVictoryScore	bans
1	100	Win	TRUE	TRUE	TRUE	TRUE	FALSE	TRUE	10	1	1	2	0	1	0	516, 142, 268, 412, 121, 1, 3, 5, 2, 4
2	200	Fail	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	2	0	0	2	0	0	0	96, 223, 81, 201, 98, 2, 4, 6, 1, 3

Figure 8: Single Match Teams Data Frame

explanations are written into the code for the custom function `get_accum_matches_teams`:

```
library(dplyr)
nalcs_matches_teams_accum <- get_accum_matches_teams(nalcs_matches, nalcs_matchid_df)
```

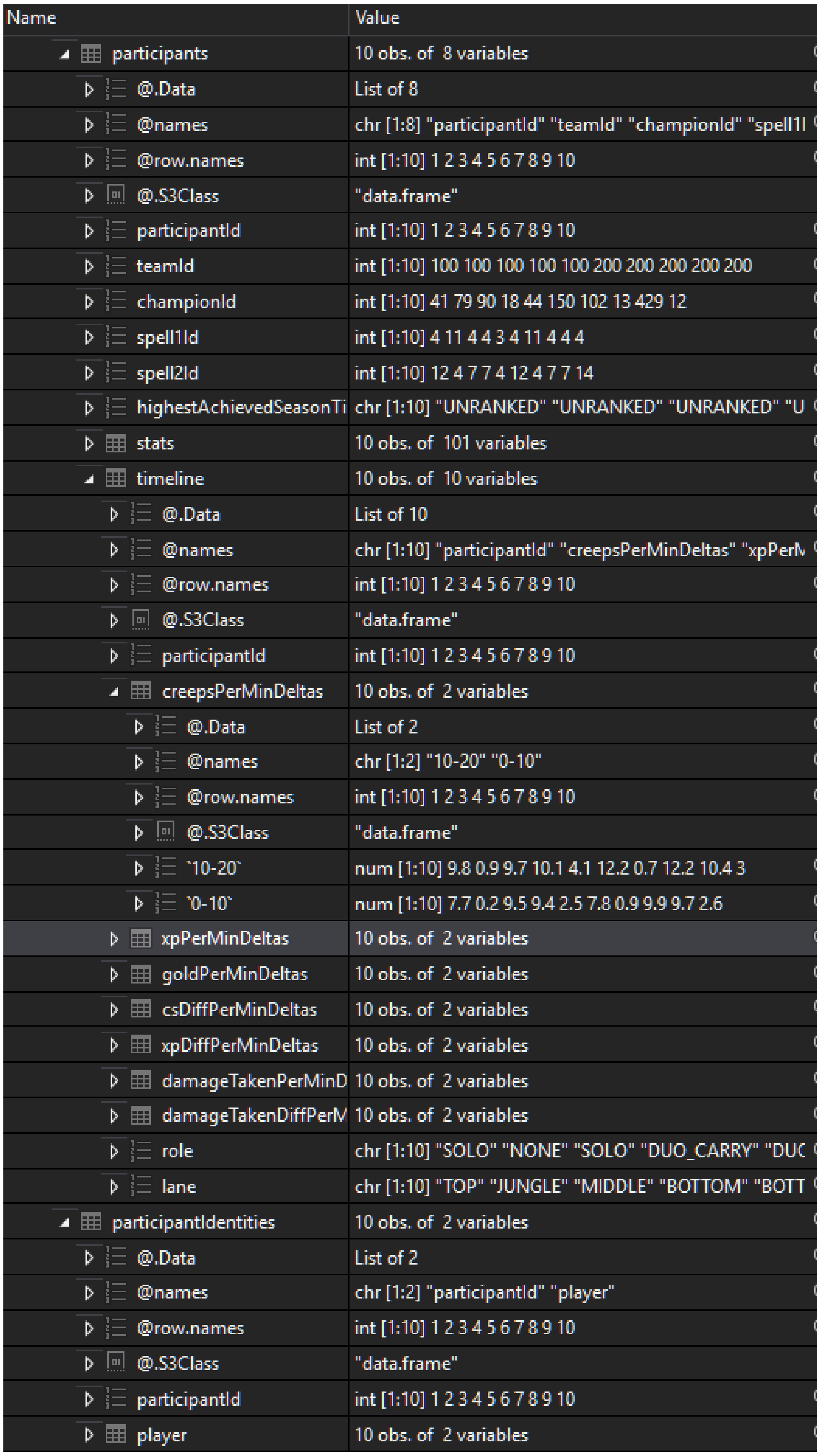
```
get_accum_matches_teams <- function(league_matchlist, league_matchid_df) {
  league_matches_teams_accum <- data.frame(NULL)
  for (i in 1:length(league_matchlist)) {
    # Convert win column (Win/Fail) to logical (TRUE/FALSE)
    league_matchlist[[i]]$teams["win"] <- as.logical(
      eval(league_matchlist[[i]]$teams["win"]) == "Win")
    # Add team names column
    league_matchlist[[i]]$teams["teamName"] <- unname(unlist(c(
      league_matchid_df[i, c("Blue.Team", "Red.Team")])))
    # Add game number column
    league_matchlist[[i]]$teams["gameNumber"] <- c(i, i)
    # Add tiebreaker and playoff column
    league_matchlist[[i]]$teams["isTiebreaker"] <- unname(unlist(c(
      league_matchid_df[i, c("Tiebreaker", "Tiebreaker")])))
    league_matchlist[[i]]$teams["isPlayoff"] <- unname(unlist(c(
      league_matchid_df[i, c("Playoff", "Playoff")])))
    # Add game duration
    league_matchlist[[i]]$teams["duration"] <- rep(
      league_matchlist[[i]]$gameDuration, 2)

    # Concatenate rows from current match onto the accumulation DF
    league_matches_teams_accum <- league_matches_teams_accum %>%
      bind_rows(league_matchlist[[i]]$teams %>% select(-bans))
  }
  # Change all teamId = 100/200 to Blue/Red
  # and remove some irrelevant columns
  league_matches_teams_accum <- league_matches_teams_accum %>%
    mutate(teamId = replace(teamId, grepl('100', teamId), 'Blue')) %>%
    mutate(teamId = replace(teamId, grepl('200', teamId), 'Red')) %>%
    select(-vilemawKills, -dominionVictoryScore)
  return(league_matches_teams_accum)
}
```

Now, let's look at the `participants` and `participantIdentities` data frames:

There are a lot of nested data frames within the data frame, but fortunately, each observation in each data frame corresponds with each player in the match, as shown by that each nested data frame (`stats`, `timeline`, and `xxxPerMinDeltas`) plus the `participantIdentities` DF has a `participantId` column. Thus, we can join them together. However, the `timeline` data frame itself has a lot of nested DFs, and each those DFs have the same column names (`` `10-20` ``, `` `0-10` ``, etc.). We will use the `jsonlite::flatten()` function on the `timeline` DF and then join all the previously mentioned nested DFs together:

```
ret_df <- match_participants_df %>%
    # Remove the nested stats and timeline DFs
    select(-stats, - timeline) %>%
    # Bind with the participantIdentites$player DF
    bind_cols(match_participantids_df$player) %>%
    # Join with the separate champions DF (by championId column)
    inner_join(champions_df_simple) %>%
    # Join with the stats DF (by participantId column)
    inner_join(match_participants_df$stats) %>%
```



Name	Value
participants	10 obs. of 8 variables
@.Data	List of 8
@names	chr [1:8] "participantId" "teamId" "championId" "spell1I
@row.names	int [1:10] 1 2 3 4 5 6 7 8 9 10
@.S3Class	"data.frame"
participantId	int [1:10] 1 2 3 4 5 6 7 8 9 10
teamId	int [1:10] 100 100 100 100 100 200 200 200 200 200
championId	int [1:10] 41 79 90 18 44 150 102 13 429 12
spell1Id	int [1:10] 4 11 4 4 3 4 11 4 4 4
spell2Id	int [1:10] 12 4 7 7 4 12 4 7 7 14
highestAchievedSeasonTi	chr [1:10] "UNRANKED" "UNRANKED" "UNRANKED" "U
stats	10 obs. of 101 variables
timeline	10 obs. of 10 variables
@.Data	List of 10
@names	chr [1:10] "participantId" "creepsPerMinDeltas" "xpPerM
@row.names	int [1:10] 1 2 3 4 5 6 7 8 9 10
@.S3Class	"data.frame"
participantId	int [1:10] 1 2 3 4 5 6 7 8 9 10
creepsPerMinDeltas	10 obs. of 2 variables
@.Data	List of 2
@names	chr [1:2] "10-20" "0-10"
@row.names	int [1:10] 1 2 3 4 5 6 7 8 9 10
@.S3Class	"data.frame"
`10-20`	num [1:10] 9.8 0.9 9.7 10.1 4.1 12.2 0.7 12.2 10.4 3
`0-10`	num [1:10] 7.7 0.2 9.5 9.4 2.5 7.8 0.9 9.9 9.7 2.6
xpPerMinDeltas	10 obs. of 2 variables
goldPerMinDeltas	10 obs. of 2 variables
csDiffPerMinDeltas	10 obs. of 2 variables
xpDiffPerMinDeltas	10 obs. of 2 variables
damageTakenPerMinD	10 obs. of 2 variables
damageTakenDiffPerM	10 obs. of 2 variables
role	chr [1:10] "SOLO" "NONE" "SOLO" "DUO_CARRY" "DUC
lane	chr [1:10] "TOP" "JUNGLE" "MIDDLE" "BOTTOM" "BOTT
participantIdentities	10 obs. of 2 variables
@.Data	List of 2
@names	chr [1:2] "participantId" "player"
@row.names	int [1:10] 1 2 3 4 5 6 7 8 9 10
@.S3Class	"data.frame"
participantId	int [1:10] 1 2 3 4 5 6 7 8 9 10
player	10 obs. of 2 variables

Figure 9: Single Match Teams Data Frame

```
    # Join with timeline DF (by participantId column)
    inner_join(match_participants_df$timeline %>% flatten())
```

Next, like with the `teams` DF, we add `gameNumber`, `isTiebreaker`, `isPlayoff`, `duration`, and `Blue` and `Red` team columns to the newly-joined `participants` DF.

Addionally, we want to add the a `teamRole` column to the `participants` DF. Luckily, through personal observation and knowledge of the game and the eSport, we know that the players on each team are placed, in the DF, in the same order by their role: Top, Jungle, Mid, Bottom Carry, and Support. Therefore, we can write such a script accordingly:

```
flattened_df['teamRole'] <- NULL
# Get team roles
for (j in 1:nrow(flattened_df)) {
  if        (flattened_df[j, 'participantId'] == 1 ||
             flattened_df[j, 'participantId'] == 6) {
    flattened_df[j, 'teamRole'] = "TOP"
  } else if (flattened_df[j, 'participantId'] == 2 ||
             flattened_df[j, 'participantId'] == 7) {
    flattened_df[j, 'teamRole'] = "JUNGLE"
  } else if (flattened_df[j, 'participantId'] == 3 ||
             flattened_df[j, 'participantId'] == 8) {
    flattened_df[j, 'teamRole'] = "MID"
  } else if (flattened_df[j, 'participantId'] == 4 ||
             flattened_df[j, 'participantId'] == 9) {
    flattened_df[j, 'teamRole'] = "BOTCARRY"
  } else {
    flattened_df[j, 'teamRole'] = "SUPPORT"
  }
}
```

Concatenating the multiple `participants` DFs together is very similar to when we put together the `teams` DFs, so we won't include a code sample here; we basically ran a for loop through all the matches data, wrangled each `participants` sub-DF, then used `bind_rows()` to add it to an accumulative DF of all the other `participants` DFs. With that, we have our match-by-match player totals data set, as shown here and at the top of the report:

Entire Split Match-by-Match Player Totals

Creating Match-by-Match Team Totals Dataset

Now that we have the player totals data set, we can use that set to add up the totals of each player of each team of each match to create team totals for each match. With the custom function `get_match_combined_participant_stats_df()`, we add up the stats of the five players of one team in one match, as shown below:

```
get_match_combined_participant_stats_df <- function(match_team_df) {

  # Replace all the NAs in the "Deltas" columns with zeroes
  match_team_df <- match_team_df %>%
    mutate_at(vars(contains("Deltas")), funs(replace(., is.na(.), 0)))

  # Groups observations by game number and team name
  # (the other group-by variables are used just so they can be included in the output DF)
  # Sums up a bunch of columns together
  match_team_df <- match_team_df %>%
    group_by(teamName, teamId, win, gameNumber, duration, isTiebreaker, isPlayoff) %>%
    summarize_at(vars(kills:assists, totalDamageDealt:trueDamageDealt,
      totalDamageDealtToChampions:goldSpent, totalMinionsKilled:wardsKilled,
      'creepsPerMinDeltas.10-20', 'creepsPerMinDeltas.0-10', 'xpPerMinDeltas.10-20',
      'xpPerMinDeltas.0-10', 'goldPerMinDeltas.10-20', 'goldPerMinDeltas.0-10',
      'damageTakenPerMinDeltas.10-20', 'damageTakenPerMinDeltas.0-10'), sum)
  return(match_team_df)
}
```

As indicated in the code sample above, we encountered some `NA`s in the `participants` data set. While there are a few outliers, most of the `NA`s occur, because some matches do not last longer than 30 minutes, so any of the `XxxPerMinDelta.30-end` columns do not apply to those matches.

With this new teammates-summed-up-per-match DF created, we can now join this together with the accumulated `teams` DF to create the ***match-by-match team totals*** dataset, as shown in this code snippet:

```
# Joins the "teams" DF and the "participants combined" DF together
nalcs_matches_tpc_accum <- nalcs_matches_participants_combined_accum %>%
  inner_join(nalcs_matches_teams_accum)
```

And here is the link to the dataset (same as at the top of the report): Entire Split Match-by-Match Team Totals

Creating Regular Season Team Totals Dataset

With the new ***match-by-match team totals*** dataset, we can add up the stats of each team across an entire regular season.

First, we filter out the regular season matches:

```
# Filter for just regular season games
league_regseason_tpc_df <- league_matches_tpc_accum %>%
  filter(isTiebreaker == FALSE & isPlayoff == FALSE)
```

Then, we group the rows of the DF by team and add up the numerical stats:

```
league_regseason_team_totals_df <-
# First parentheses group: sums of stats by team
(league_regseason_tpc_df %>%
  group_by(teamName) %>%
  summarise_at(vars(duration, kills:wardsKilled, towerKills:riftHeraldKills,
    'creepsPerMinDeltas.10-20', 'creepsPerMinDeltas.0-10', 'xpPerMinDeltas.10-20',
    'xpPerMinDeltas.0-10', 'goldPerMinDeltas.10-20', 'goldPerMinDeltas.0-10',
    'damageTakenPerMinDeltas.10-20', 'damageTakenPerMinDeltas.0-10'), sum)) %>%
    ...
```

Then, we continue by tallying most of the TRUE/FALSE columns:

```
# Include tidyr package at top of script file so we can use spread(
library(tidyr)
  ...
  # More parenthesis groups: tallying first-objective columns
  inner_join(league_regseason_tpc_df %>%
    group_by(teamName, firstBlood) %>%
    tally() %>% spread(firstBlood, n) %>% select('TRUE') %>%
    rename('firstBloods' = 'TRUE')) %>%
  inner_join(league_regseason_tpc_df %>%
    group_by(teamName, firstTower) %>%
    tally() %>% spread(firstTower, n) %>% select('TRUE') %>%
    rename('firstTowers' = 'TRUE')) %>%
  ...
```

Then, we tally the wins and losses:

```
  ...
  # Last parentheses group: tallying wins and losses by team
  inner_join(league_regseason_tpc_df %>%
    group_by(teamName, win) %>%
    tally() %>%
    spread(win, n) %>% # "transposes" the DF so that TRUE (win) and FALSE (loss) are the column names
    rename('losses' = 'FALSE', 'wins' = 'TRUE')) # renames the T/F columns to W/L
```

Finally, we re-order the columns in this big joined DF:

```
# Reordering columns - teamName, wins, losses, <everything else>
league_regseason_team_totals_df <- league_regseason_team_totals_df[, c(1, 56, 55, 2:54)]
```

We now have the ***regular season team totals*** dataset: Regular Season Team Cumulative Totals

Creating Regular Season Team Averages Per Match Dataset

With the ***match-by-match team totals*** dataset, we can compute the mean stats of each team across an entire regular season. This is requires much less code than computing the ***regular season team totals*** dataset. First, we filter out the regular season matches. Then, we group the DF by team and then summarize the columns we need using the mean function:

```
league_regseason_team_avgs_fg <- league_regseason_tpc_df %>%
  group_by(teamName) %>%
  summarise_at(vars(win, duration, kills:riftHeraldKills), mean)
```

Here is the resulting dataset: Regular Season Team Averages

Creating Regular Season Opponent Totals Dataset

Here we go back to using the ***match-by-match team totals*** dataset, and then, for each match, we just swap the team names by using ***dplyr***'s `lead()` and `lag()` functions:

```
# Get opponent's data (just swapping the team names of each game in the previous DF)
nalcs_matches_tpc_opps_accum <- nalcs_matches_tpc_accum %>%
  group_by(gameNumber) %>%
  mutate(teamName = ifelse(teamId == "Blue",
    as.character(lead(teamName)),
    as.character(lag(teamName))
  ))
```

Then, we just use the same procudure we used to create the ***regular season team totals*** data set.

Finally, we rename most of the columns of this data frame so they have an `O_` prefix:

```
# Rename the columns of regular season team opponents totals DF
nalcs_regseason_teamopps_totals_df <- nalcs_regseason_teamopps_totals_df %>%
  rename_at(vars(wins:firstDragons), function(x) {
    return (paste("O", x, sep="_"))
  })
```

Here's the link to the ***regular season opponent totals*** dataset: Regular Season Opposing Team Cumulative Totals