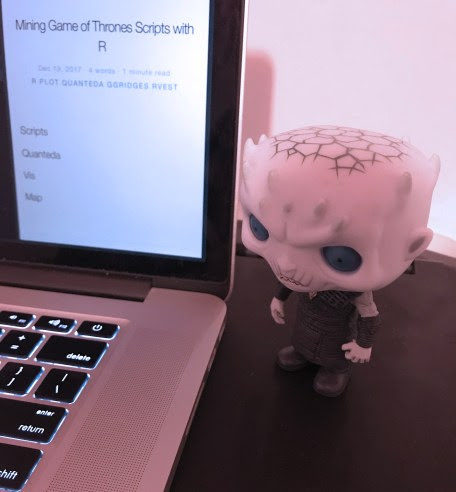
**Quantitative Text Analysis Part II**

I meant to showcase the quanteda package in my previous post on [the Weinstein Effect](https://www.gokhanciflikli.com/post/weinstein-effect/) but had to switch to tidytext at the last minute. Today I will make good on that promise. quanteda is developed by [Ken Benoit](http://kenbenoit.net/) and maintained by [Kohei Watanabe](https://koheiw.net/) – go LSE! On that note, the first [2018 LondonR meeting](http://www.londonr.org/) will be taking place at the LSE on January 16, so do drop by if you happen to be around. quanteda v1.0 will be unveiled there as well.

Given that I have already used the data I had in mind, I have been trying to identify another interesting (and hopefully less depressing) dataset for this particular calling. Then it snowed in London, and the dire consequences of this supernatural phenomenon were covered extensively by the [r/CasualUK/](https://www.reddit.com/r/CasualUK/). One thing led to another, and before you know it I was analysing Game of Thrones scripts:



He looks like he also hates the lack of native parallelism in R.

**Getting the Scripts**

Mandatory spoilers tag, the rest of the post contains (surprise) spoilers (although only up until the end of the sixth season).

I intend to keep to the organic three-step structure I have developed lately in my posts: obtaining data, showcasing a package, and visualising the end result. With GoT, there are two obvious avenues: full-text books or the show scripts. I decided to go with the show because I’m a filthy casual fan. A wise man once quipped: *‘Never forget what you are. The rest of the world will not. Wear it like armor, and it can never be used to hurt you.’* It’s probably a Chinese proverb.

Nowadays it’s really easy to scrape interesting stuff online. rvest package is especially convenient to use. How it works is that you feed it a URL, it reads the html, you locate which html tag/class contains the information you want to extract, and finally it lets you clean up the text by removing the html bits. Let’s do an example.

Mighty Google told me that [this website](https://www.springfieldspringfield.co.uk/) has GoT scripts online. Cool, let’s fire up the very first episode. With any modern browser, you should be able to inspect the page to see the underlying code. If you hover where the text is located in inspection mode, you’ll find that it’s wrapped in ‘scrolling-script-container’ tags. This is not a general rule, so you’ll probably have to do this every time you try to scrape a new website.

library(rvest)

library(tidyverse)

url <- "https://www.springfieldspringfield.co.uk/view\_episode\_scripts.php?tv-show=game-of-thrones&episode=s01e01"

webpage <- read\_html(url)

#note the dot before the node

script <- webpage %>% html\_node(".scrolling-script-container")

full.text <- html\_text(script, trim = TRUE)

glimpse(full.text)

## chr "Easy, boy. What do you expect? They're savages. One lot steals a goat from another lot, before you know it they"| \_\_truncated\_\_

Alright, that got us the first episode. Sixty-something more to go! Let’s set up and execute a for-loop in R because we like to live dangerously:

#Loop for getting all GoT scripts

baseurl <- "https://www.springfieldspringfield.co.uk/view\_episode\_scripts.php?tv-show=game-of-thrones&episode="

s <- c(rep(1:6, each = 10), rep(7, 7))

season <- paste0("s0", s)

ep <- c(rep(1:10, 6), 1:7)

episode <- ifelse(ep < 10, paste0("e0", ep), paste0("e", ep))

all.scripts <- NULL

#Only the first 6 seasons

for (i in 1:60) {

url <- paste0(baseurl, season[i], episode[i])

webpage <- read\_html(url)

script <- webpage %>% html\_node(".scrolling-script-container")

all.scripts[i] <- html\_text(script, trim = TRUE)

}

Now, the setup was done in a way to get all aired episodes, but the website does not currently have S07E01 (apparently they had an incident and still recovering data). We can find it somewhere else of course, however the point is not to analyse GoT in a complete way but to practice data science with R. So I’ll just cut the loop short by only running it until the end of the sixth season. Let’s see what we got:

got <- as.data.frame(all.scripts, stringsAsFactors = FALSE)

counter <- paste0(season, episode)

row.names(got) <- counter[1:60]

colnames(got) <- "text"

as.tibble(got)

## # A tibble: 60 x 1

## text

## \*

## 1 "Easy, boy. What do you expect? They're savages. One lot steals a goat from

## 2 "You need to drink, child. And eat. lsn't there anything else? The Dothraki

## 3 "Welcome, Lord Stark. Grand Maester Pycelle has called a meeting of the Sma

## 4 "The little lord's been dreaming again. - We have visitors. - I don't want

## 5 "Does Ser Hugh have any family in the capital? No. I stood vigil for him my

## 6 "Your pardon, Your Grace. I would rise, but Do you know what your wife has

## 7 "\"Summoned to court to answer for the crimes \"of your bannerman Gregor Cl

## 8 "Yah! Left high, left low. Right low, lunge right. You break anything, the

## 9 You've seen better days, my lord. Another visit? lt seems you're my last fr

## 10 "Look at me. Look at me! Do you remember me now, boy, eh? Remember me? Ther

## # ... with 50 more rows

Those are the first sentences of the first ten GoT episodes – looks good! We won’t worry about the backslash on line 7 for now. One quirk of this website is that they seem to have used small case L for capital I (e.g. “l’snt” in line 2 above) in some places. You can easily fix those with a string replacement solution; I’ll let them be. Right, let’s generate some numbers to go along with all this text.

**Text Analysis with Quanteda**

As I covered n-grams in my previous post, I will try to diversify a bit. That should be doable – quanteda offers a smooth ride and it has a nicely documented [website](http://docs.quanteda.io/reference/). Which is great, otherwise I don’t think I’d have gotten into it! Let’s transform our scripts dataset into a corpus. The showmeta argument should cut off the additional information you get at the end of a summary, however it doesn’t work on my computer. Yet, we can manipulate the meta data manually as well:

library(quanteda)

## quanteda version 0.99.22

## Using 7 of 8 threads for parallel computing

##

## Attaching package: 'quanteda'

## The following object is masked from 'package:utils':

##

## View

got.corpus <- corpus(got)

metacorpus(got.corpus, "source") <- "No peaking!"

summary(got.corpus, 10, showmeta = FALSE)

## Corpus consisting of 60 documents, showing 10 documents:

##

## Text Types Tokens Sentences

## s01e01 844 3388 392

## s01e02 1071 4177 398

## s01e03 1163 4436 468

## s01e04 1398 6074 512

## s01e05 1388 6434 597

## s01e06 1064 4333 481

## s01e07 1157 4704 526

## s01e08 1001 3993 400

## s01e09 1213 5335 510

## s01e10 1094 4804 418

##

## Source: No peaking!

## Created: Wed Dec 20 14:18:43 2017

## Notes:

I intentionally turned on the message option in the above chunk so that you can see quanteda is thoughtful enough to leave you with a single core for your *other* computational purposes. The Night King certainly approves. You could also pass a compress = TRUE argument while creating a corpus, which is basically a trade-off between memory space and computation speed. We don’t have that much text so it’s not a necessity for us, but it’s good to know that the option exists.

When you do things for the first couple of times, it’s good practice to conduct a couple of sanity checks. The kwic function, standing for ‘keywords-in-context’, returns a list of such words in their immediate context. This context is formally defined by the window argument, which is bi-directional and includes punctuation. If only there were sets of words in the GoT universe that are highly correlated with certain houses…

#Money money money

kwic(got.corpus, phrase("always pays"), window = 2)

##

## [s01e05, 931:932] a Lannister | always pays | his debts

## [s01e05, 1275:1276] A Lannister | always pays | his debts

## [s01e06, 1755:1756] a Lannister | always pays | his debts

## [s01e06, 3479:3480] A Lannister | always pays | his debts

## [s02e08, 3654:3655] a Lannister | always pays | her debts

## [s04e07, 1535:1536] A Lannister | always pays | his debts

#What's coming

kwic(got.corpus, "winter", window = 3)

##

## [s01e01, 383] forever. And | winter | is coming.

## [s01e01, 3180] the King. | Winter | is coming.

## [s01e03, 577] king?- | Winter | may be coming

## [s01e03, 1276] And when the | winter | comes, the

## [s01e03, 1764] our words. | Winter | is coming.

## [s01e03, 1784] . But now | winter | is truly coming

## [s01e03, 1792] And in the | winter | , we must

## [s01e03, 1968] is for the | winter | , when the

## [s01e04, 5223] remember the last | winter | ? How long

## [s01e04, 5289] during the last | winter | . It was

## [s01e04, 5559] And come the | winter | you will die

## [s01e10, 4256] Wall! And | winter | is coming!

## [s02e01, 566] an even longer | winter | . A common

## [s02e01, 579] for a five-year | winter | . If it

## [s02e01, 614] . And with | winter | coming, it'll

## [s02e01, 1140] not stand the | winter | . The stones

## [s02e01, 2719] cold breath of | winter | will freeze the

## [s02e02, 5504] will starve when | winter | comes. The

## [s02e03, 1095] of summer and | winter | is coming.

## [s02e05, 2343] The Starks understand | winter | better than we

## [s02e05, 3044] half of last | winter | beyond the Wall

## [s02e05, 3051] . The whole | winter | . He was

## [s02e10, 4609] them, through | winter | , summer,

## [s02e10, 4613] , summer, | winter | again. Across

## [s03e01, 3276] Wait out the | winter | where it's beautiful

## [s03e03, 5029] from home and | winter | is coming.

## [s03e04, 196] from home and | winter | is coming.

## [s03e04, 3336] house." | Winter | is coming!

## [s03e04, 4092] for a short | winter | . Boring and

## [s03e04, 4860] make it through | winter | ? Enough!

## [s03e05, 1844] might survive the | winter | . A million

## [s03e05, 5436] Wait out the | winter | .- Winter

## [s03e05, 5439] winter.- | Winter | could last five

## [s03e07, 5545] be dead by | winter | . She'll be

## [s04e01, 3493] your balls till | winter | ? We wait

## [s04e03, 2282] be dead come | winter | .- You

## [s04e03, 2309] be dead come | winter | . Dead men

## [s04e10, 474] both know that | winter | is coming.

## [s05e01, 4301] will survive the | winter | , not if

## [s05e01, 4507] hero. Until | winter | comes and the

## [s05e03, 2723] prisoners indefinitely. | Winter | is coming.

## [s05e04, 651] afford? With | winter | coming, half

## [s05e04, 2664] a crown of | winter | roses in Lyanna's

## [s05e04, 2796] Landing before the | winter | snows block his

## [s05e05, 656] Jon Snow. | Winter | is almost upon

## [s05e05, 1496] you. But | winter | is coming.

## [s05e05, 3382] could turn to | winter | at any moment

## [s05e07, 1305] --- | Winter | is coming.

## [s05e07, 1329] Black, we | winter | at Castle Black

## [s05e07, 1342] many years this | winter | will last?

## [s06e05, 912] the winds of | winter | as they lick

## [s06e06, 2533] . Don't fear | winter | . Fear me

## [s06e10, 2675] white raven. | Winter | is here.

## [s06e10, 4427] is over. | Winter | has come.

We find that these Lannister folks sound like they are the paying-back sort and this winter business had a wild ride before it finally arrived. However, our findings indicate many saw this coming. Moving on, let’s look at tokens. We’ll get words, including n-grams up to three, and remove punctuation:

got.tokens <- tokens(got.corpus, what = "word", ngrams = 1:3, remove\_punct = TRUE)

head(got.tokens[[7]], 15)

## [1] "Summoned" "to" "court" "to" "answer"

## [6] "for" "the" "crimes" "of" "your"

## [11] "bannerman" "Gregor" "Clegane" "the" "Mountain"

See, we didn’t have to worry about the backslash after all.

Tokens are good, however for the nitty-gritty, we want to convert our corpus into a document-feature matrix using the dfm function. After that, we can populate the top *n* features by episode:

got.dfm <- dfm(got.corpus, remove = stopwords("SMART"), remove\_punct = TRUE)

top.words <- topfeatures(got.dfm, n = 5, groups = docnames(got.dfm))

#S06E05

top.words[55]

## $s06e05

## hodor door hold men bran

## 42 33 31 21 20

Sad times. One quick note – we removed stopwords using the SMART dictionary that comes with quanteda. We could also use stopwords("english") and several other languages. SMART differs from English somewhat, however both are arbitrary by design. You can call stopwords("dictionary\_name") to see what they contain; these words will be ignored. Sometimes, you might want to tweak the dictionary if they happen to include words that you rather keep.

Let’s repeat the previous chunk, but this time we group by season. Recycle the season variable and re-do the corpus:

#Include the season variable we constructed earlier

got$season <- s[1:60]

got.group.corpus <- corpus(got)

got.group.dfm <- dfm(got.group.corpus, ngrams = 1:3, groups = "season",

remove = stopwords("SMART"), remove\_punct = TRUE)

One convenient feature of having a grouped corpus is that we can analyse temporal trends. Say, you are known by many names and/or happen to be fond of titles:

dany <- c("daenerys", "stormborn", "khaleesi","the\_unburnt",

"mhysa", "mother\_of\_dragons", "breaker\_of\_chains")

titles <- got.group.dfm[, colnames(got.group.dfm) %in% dany]

titles <- as.data.frame(titles)

#Divide all cells with their row sums and round them up

round(titles / rowSums(titles), 2)

## daenerys stormborn khaleesi mhysa mother\_of\_dragons the\_unburnt

## 1 0.20 0.02 0.78 0.00 0.00 0.00

## 2 0.13 0.08 0.54 0.00 0.25 0.00

## 3 0.11 0.11 0.35 0.35 0.05 0.00

## 4 0.19 0.04 0.30 0.44 0.04 0.00

## 5 0.46 0.04 0.04 0.31 0.15 0.00

## 6 0.41 0.16 0.09 0.06 0.16 0.03

## breaker\_of\_chains

## 1 0.00

## 2 0.00

## 3 0.03

## 4 0.00

## 5 0.00

## 6 0.09

Khaleesi dominates the first season (~80%), and it is her most one-sided title usage of any season. In S2, she gets the moniker of ‘mother of dragons’ in addition to khaleesi (25% and 55%, respectively). Seasons 3 and 4 are the most balanced, when she was known as khaleesi and mhysa somewhat equally (~35% both). In the last two seasons (in our dataset, at least), she is most commonly (>40%) called/mentioned by her actual name. This particular exercise would have definitely benefited from S7 scripts. You can refer to the titles object to see the raw counts rather than column percentages by row.

Yet another thing we can calculate is term similarity and distance. Using textstat\_simil, we can get the top *n* words that are associated with it:

sim <- textstat\_simil(got.dfm, diag = TRUE, c("throne", "realm", "walkers"),

method = "cosine", margin = "features")

lapply(as.list(sim), head)

## $throne

## iron lord men father kingdoms fire

## 0.7957565 0.7707637 0.7573764 0.7493148 0.7343086 0.7336560

##

## $realm

## protector kingdoms robert honor hundred shadowcats

## 0.7237571 0.6604497 0.6558736 0.6417062 0.6396021 0.6192188

##

## $walkers

## white deserter detail corners guardsman pups

## 0.8206750 0.7774816 0.7774816 0.7774816 0.7774816 0.7774816

Shadowcats? White Walker pups?

Finally, one last thing before we move on to the visualisations. We will model topic similarities and call it a package. We’ll need topicmodels, and might as well write another for-loop (double-trouble). The below code is not evaluated here, but if you do, you’ll find that GoT consistently revolves around lords, kings, the realm, men, and fathers with the occasional khaleesi thrown in.

library(topicmodels)

for (i in 1:6) {

x <- 1

got.LDA <- LDA(convert(got.dfm[x:(x + 9), ], to = "topicmodels"),

k = 3, method = "Gibbs")

topics <- get\_terms(got.LDA, 4)

print(paste0("Season", i))

print(topics)

x <- x + 10

}

**Joy Plots**

Numbers and Greek letters are cool, however you’ll find that a well-made graph can convey a lot at a glance. quanteda readily offers several statistics that lend themselves very well to [Joy plots](https://i.ytimg.com/vi/V3Ioohi9aqE/maxresdefault.jpg). When you call summary on a corpus, it reports descriptives on type, tokens, and sentences. These are all counts, and the difference between a type and a token is that the former provides a count of distinct tokens: (a, b, c, c) is four tokens but three types.

Let’s recycle our corpus as a dataframe and clean it up. After that, we’ll get rid of the redundant first column, followed by renaming the contents of the season variable and make sure it’s a factor. Then, we’ll calculate the average length of a sentence by dividing token count by the sentence count. Finally, we shall gather the spread-out variables of type, tokens, and sentences into a single ‘term’ and store their counts under ‘frequency’. Usually one (i.e. who works with uncurated data) does the transformation the other way around; you spread a single variable into many to tidy it up – it’s good to utilise this lesser-used form from time to time. Also, we are doing all of this just to be able to use the facet\_grid argument: you can manually plot four separate graphs and display them together but that’s not how we roll around here.

#Setup; first two lines are redundant if you populated them before

got$season <- s[1:60]

got.group.corpus <- corpus(got)

got.stats <- as.data.frame(summary(got.group.corpus), row.names = 1:60)

got.stats <- got.stats[, 2:5]

got.stats$season <- paste0("Season ", got.stats$season)

got.stats$season <- as.factor(got.stats$season)

got.stats$`Average Sentence Length` <- got.stats$Token / got.stats$Sentences

got.stats <- gather(got.stats, term, frequency, -season)

means <- got.stats %>%

group\_by(season, term) %>%

summarise(mean = floor(mean(frequency)))

#Plot

library(ggplot2)

library(ggridges)

library(viridis)

#Refer to previous post for installing the below two

library(silgelib)

theme\_set(theme\_roboto())

#counts by season data

ggplot(got.stats, aes(x = frequency, y = season)) +

#add facets for type, tokens, sentences, and average

facet\_grid(~term, scales = "free") +

#add densities

geom\_density\_ridges(aes(fill = season)) +

#assign colour palette; reversed legend if you decide to include one

scale\_fill\_viridis(discrete = TRUE, option = "D", direction = -1,

guide = guide\_legend(reverse = TRUE)) +

#add season means at the bottom

geom\_rug(data = means, aes(x = mean, group = season), alpha = .5, sides = "b") +

labs(title = "Game of Thrones (Show) Corpus Summary",

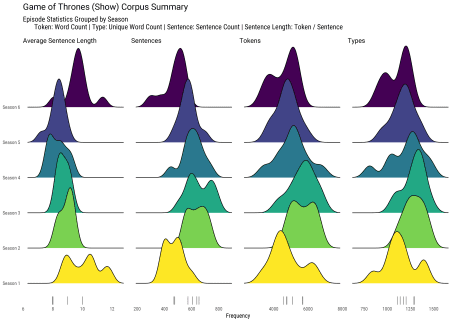
subtitle = "Episode Statistics Grouped by Season

Token: Word Count | Type: Unique Word Count | Sentence : Sentence Count | Sentence Length: Token / Sentence",

x = "Frequency", y = NULL) +

#hide the colour palette legend and the grid lines

theme(legend.position = "none", panel.grid.major = element\_blank(), panel.grid.minor = element\_blank())



[Larger PDF version here.](https://www.gokhanciflikli.com/img/joyplot.pdf) Some remarks. Each ridge represents a season and contains counts from ten episodes. These are distributions, so sharp peaks indicate clustering and multiple peaks/gradual changes signal diffusion. For example, in the first column (sentence length), we see that S1 has three peaks: some episodes cluster around 9, some at 10.5 and others at slightly less than 12. In contrast, S5 average sentence length is very specific: nearly all episodes have a mean of 9 tokens/sentence.

Moving on to the second column, we find that the number of sentences in episodes rise from S1 to S3, and then gradually go down all the way to S1 levels by the end of S6. Token and type counts follow similar trends. In other words, if we flip the coordinates, we would see a single peak between S3 and S4: increasing counts of individual terms as you get closer to the peak from both directions (i.e. from S1 to S3 and from S6 to S4), but also shorter average sentence lengths. We should be cautious about making strong inferences, however – we don’t really have the means to account for the quality of writing. Longer sentences do not necessarily imply an increase in complexity, even coupled with higher numbers of type (unique words).

**WesteRos**

In case you have seen cool ggridges plots before or generally are a not-so-easily-impressed (that counts as *one* token, by the way) type, let’s map Westeros in R. If you are also wondering why there is a [shapefile for Westeros](https://www.cartographersguild.com/showthread.php?t=30472) in the first place, that makes two of us. But don’t let these kinds of things stop you from doing data science.

The zip file contains several shapefiles; I will only read in ‘political’ and ‘locations’. You will need these files (all of them sharing the same name, not just the .shp file) in your working directory so that you can call it with ".". The spatial data come as factors, and I made some arbitrary modifications to them (mostly for aesthetics). First, in the original file the Night’s Watch controls two regions: New Gift and Bran’s Gift. I removed one an renamed the other “The Wall”. Spatial data frames are S4 objects so you need to call @data$ instead of the regular $.

Second, let’s identify the capitals of the regions and set a custom .png icon so that we can differentiate them on the map. At this point, I realised the shapefile does not have an entry for Casterly Rock – maybe they haven’t paid back the creator yet? We’ll have to do without it for now. Third, let’s manually add in some of the cool places by placing them in a vector called ‘interesting’. Conversely, we shall get rid of some so that they do not overlap with region names (‘intheway’). I’m using a %nin operator (not in) that comes with Hmisc, but there are other ways of doing it. Finally, using RColorBrewer I assigned a bunch of reds and blues – viridis looked a bit odd next to the colour of the sea.

library(Hmisc)

library(rgdal)

library(tmap)

library(RColorBrewer)

#Read in two shapefiles

westeros <- readOGR(".", "political")

locations <- readOGR(".", "locations")

#Cleaning factor levels

westeros@data$name <- `levels<-`(addNA(westeros@data$name),

c(levels(westeros@data$name),

"The Lands of Always Winter"))

levels(westeros@data$name)[1] <- "The Wall"

levels(westeros@data$name)[4] <- ""

levels(westeros@data$ClaimedBy)[11] <- "White Walkers"

#Identify capitals

places <- as.character(locations@data$name)

places <- gsub(" ", "\_", places)

capitals <- c("Winterfell", "The Eyrie", "Harrenhal", "Sunspear",

"King's Landing", "Castle Black", "Pyke",

"Casterly Rock", "Storm's End", "Highgarden")

holds <- locations[locations@data$name %in% capitals, ]

#Castle icon

castle <- tmap\_icons(file = "https://image.ibb.co/kykHfR/castle.png", keep.asp = TRUE)

#Locations we rather keep

interesting <- c("Fist of the First Men", "King's Landing",

"Craster's Keep", "Tower of Joy")

#Locations we rather get rid of

intheway <- c("Sarsfield", "Hornvale", "Cider Hall",

"Hayford Castle", "Griffin's Roost")

#Subsetting

locations <- locations[locations@data$type == "Castle" |

locations@data$name %in% interesting, ]

locations <- locations[locations@data$name %nin% intheway, ]

#Color palettes - the hard way

blues <- brewer.pal(6, "Blues")

reds <- brewer.pal(7, "Reds")

sorted <- c(blues[3], reds[4], blues[4], reds[2], reds[6],

#vale, stormlands, iron islands, westerlands, dorne

blues[6], blues[5], reds[3], reds[1], reds[5], blues[1])

#wall, winterfell, crownsland, riverlands, reach, beyond the wall

#Map

tm\_shape(westeros) +

#Colour regions using the sorted palette and plot their names

tm\_fill("ClaimedBy", palette = sorted) +

tm\_text("name", fontfamily = "Game of Thrones", size = .4, alpha = .6) +

#Plot location names and put a dot above them

tm\_shape(locations) +

tm\_text("name", size = .2, fontfamily = "Roboto Condensed", just = "top") +

tm\_dots("name", size = .01, shape = 20, col = "black", ymod = .1) +

#Plot capitals and add custom shape

tm\_shape(holds) +

tm\_dots("name", size = .05, alpha = .5, shape = castle, border.lwd = NA, ymod = .2) +

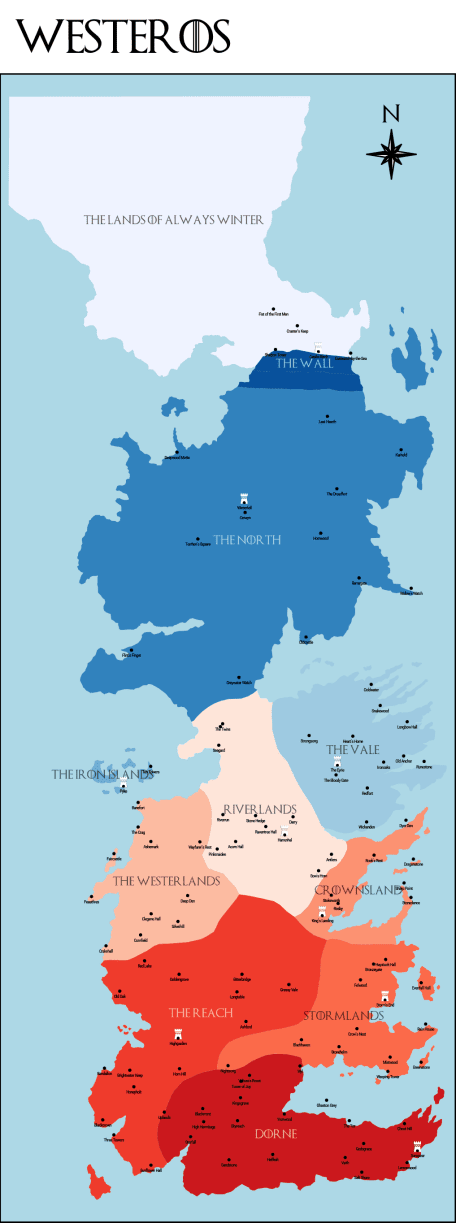
#Fluff

tm\_compass(type = "8star", position = c("right", "top"), size = 1.5) +

tm\_layout(bg.color = "lightblue", main.title = "Westeros", frame.lwd = 2,

fontfamily = "Game of Thrones") +

tm\_legend(show = FALSE)



Download the map in [PDF.](https://www.gokhanciflikli.com/img/westeros.pdf)

Woo! Okay, let’s go over what happened before wrapping this up. tmap operates similarly to ggplot grammar, so it should be understandable (relatively speaking). We are calling three shapefiles here: ‘westeros’ for the regions, ‘locations’ for the castles and manually added/subtracted places, and ‘holds’ for the capitals (which is just a subset of locations really). The tm parameters (fill, text, dots) under these shapes handle the actual plotting. For example, under westeros, we fill the regions by ‘ClaimedBy’, which would normally return the names of the Houses. However, that’s only the fill argument, and the text parameter in the next line calls ‘name’, which is the name of the regions (and what gets plotted). You can download [GoT fonts](https://fontmeme.com/fonts/game-of-thrones-font/) for added ambiance. We pass our custom castle shape by calling shape = castle and remove the square borders around the .png with the border.lwd = NA. Finally, the ymod argument helps us overcome overlapping labels by slightly moving them up in the y-axis.