*The Battle for Wesnoth* is an open-source, turn-based strategy game. The game world is rich, with several factions, maps and literally hundreds of available units. In this tutorial, you will learn how to transform a medium-sized data sets, such as game metadata, into a useful format for further analysis using R.

You will learn what key principles a *tidy* data set adheres to, why it is useful to follow them consequently, and how to clean the data you are given. Tidying is also a great way to get to know a new data set.

Finally, in this tutorial you will learn how to write a function that makes your analysis look much cleaner and allows you to execute repetitive elements in your analysis in a very reproducible way. The function will allow you to load the latest version of the data dynamically into a flexible data scheme, which means that large parts of the code will not have to change when new data is added.

[](https://i2.wp.com/datascienceplus.com/wp-content/uploads/2019/04/battle_of_wesnoth_screen.png?ssl=1)

**The Data**

The data is stored in a Google Cloud bucket, which you may find following this [link](https://storage.googleapis.com/wesv15/Wesv15.csv). The csv file that will be downloaded is called Wesv15.csv and 50MB in size. You set your working directory and copy the downloaded file in there.

Because it is a relatively small data set, you can load the data in one go. As this tutorial uses for the most part functionality from the tidyverse, it is consistent to use the read\_csv function instead of base Rs function read.csv.

One benefit in using the tidyverse function compared to the base function read.csv is that the data is immediately transformed into a tibble, a file format similar to data frames, but with the benefit of printing at any time only a manageable subset of the data onto the screen.

# Plenty of useful functions for cleaning and plotting

library(tidyverse)

# For some datetime utility

library(lubridate)

# For regular expression handling

library(rebus)

data <- read\_csv('https://storage.googleapis.com/arvid-test-bucket/TidyWesnothArticle/Wesv15.csv')

It is best to first check the dimensions of the loaded data frame.

dim(data)

*## [1] 54591 570*

Wow, 570 columns, that is certainly a lot! It is always a good idea to look at some of the actual data. You can do this with just typing data, View(data) or glimpse(data) to get a feeling.  
As you follow this code, it is a good idea to set the maximum number of printed columns to a very low number, e.g. 3, otherwise you might be flooded by the default 100 column names that are printed in tibble data formats every time you type data from now on.

options(tibble.max\_extra\_cols = 3)

data

*## # A tibble: 54,591 x 570*

*## game\_id era map color faction leader number player team stats.cost*

*##*

*## 1 1 rby … the … teal Drakes Drake… 1 Maste… north 271*

*## 2 1 rby … the … oran… Loyali… Lieut… 2 norri… south 274*

*## 3 2 rby … fall… red Northe… Orcis… 1 Bonobo west 384*

*## 4 2 rby … fall… teal Undead Death… 2 whatc… east 415*

*## 5 3 rby … den … blue Drakes Drake… 1 Xplor… south 188*

*## 6 3 rby … den … red Loyali… White… 2 Fanjo north 192*

*## 7 4 era\_… haml… red Rebels Elvis… 1 Balte… north 245*

*## 8 4 era\_… haml… green Northe… Troll… 2 Veron… south 254*

*## 9 5 rby\_… sabl… teal Rebels White… 1 Orang… north 169*

*## 10 5 rby\_… sabl… teal Northe… Troll 2 khiM south 177*

*## # … with 5.458e+04 more rows, and 560 more variables:*

*## # stats.inflicted\_actual , stats.inflicted\_expected ,*

*## # stats.taken\_actual , …*

Since the output is large, it is often easier to look at a subset of the data. You can use head(data) or tail(data), or directly subset with numeric indices as you will see below. It is also good to know that when there are many columns, View(data) is not suitable to look at all of them. Instead, It is better to use utils::View(data), which opens a separate window for viewing. While you cannot filter and search the data, you can at least see it all[3]. That was a lot of information on just viewing your data, but it is really important to have the right tool at the right time.

[3]: It seems to suggest that most tools are built for tidy data, where there are typically more rows than columns.

Let us inspect the first six row entries of the first eight column variables.

data[1:6, 1:8]

*## # A tibble: 6 x 8*

*## game\_id era map color faction leader number player*

*##*

*## 1 1 rby no mi… the freela… teal Drakes Drake Fl… 1 MasterB…*

*## 2 1 rby no mi… the freela… orange Loyalists Lieutena… 2 norring…*

*## 3 2 rby no mi… fallenstar… red Northern… Orcish S… 1 Bonobo*

*## 4 2 rby no mi… fallenstar… teal Undead Deathbla… 2 whatcha…*

*## 5 3 rby no mi… den of onis blue Drakes Drake Ar… 1 Xplorer*

*## 6 3 rby no mi… den of onis red Loyalists White Ma… 2 Fanjo*

Interestingly, the first column is a game ID and it seems that two rows correspond to the same game. This is an issue that needs to be picked up at a later point. What are the next columns? These are player and unit related statistics. The dataset is very sparse, with a lot of NAs in many columns. In these columns, the column name is indicative of the statistic that is being displayed, and it is typically related to the units that are recruited or killed in a game.

data[1:6, 50:58]

*## # A tibble: 6 x 9*

*## stats.kills.dra… stats.kills.dra… stats.kills.sau… stats.kills.sau…*

*##*

*## 1 NA NA NA NA*

*## 2 1 2 4 1*

*## 3 NA NA NA NA*

*## 4 NA NA NA NA*

*## 5 NA NA NA NA*

*## 6 NA NA NA NA*

*## # … with 5 more variables: stats.kills.saurian\_soothsayer ,*

*## # stats.recruits.drake\_burner , stats.recruits.drake\_clasher ,*

*## # …*

Each player during each game can only recruit from a small subset of all available creatures. Therefore, most of the other cells will be empty.

As is often the case in real life, you have very quickly arrived at a point where the data set will need cleaning before really insightful analysis can be done.

**What is Tidy Data?**

Tidy data is a convention.  
Even before *Tidy Data* was written, most relational database engineers would consider it best practice to follow certain rules common in RDBMS (relational database management systems), such as using normal forms, which are similar to the rules laid out in *Tidy Data*. As an aspiring data scientist, you might not yet be familiar with databases, which is perfectly fine. Please keep in mind that tidy data is there to adhere to a certain standard during data analysis and ultimately helps to make your life easier. Adhering to the tidy standard means that your data is stored efficiently and your code is less likely to need much maintenance.

As it is ultimately a convention, not all data storages adhere to it. Spreadsheet applications typically do not enforce a particular tidy standard and lack the tools of easily tidying your data.. That is one of the reasons why they slowly fade out of fashion, at least in the field of big data analysis.

Even the tidy standard itself contains an element of arbitrariness. For example, data rows are usually observations and data columns are usually variables. Their roles might be reversed and the resulting new standard would work perfectly fine, provided everyone would follow it. However, most ML algorithms implicitly follow the tidy standard, therefore such a switch would be very hard to do in practice and could not be meaningfully justified.

These are the three rules that make data *tidy*:

**1. *Each row is an \*\*independent*\* observation.\***

Independent means here that one cannot guess the contents of adjacent rows in any way. For our data, this is clearly violated. The entries of the map, winner and loser columns are repeated within one game. Dependent rows mean that the data looks artificially larger than it is, and wrong statistical conclusions might be drawn. It can also confuse many machine learning algorithms.

**2. *Each column is an \*\*independent*\* variable.\***

A variable here is anything that can be observed. A variable typically has a well-defined data type (e.g. numeric or character) and enforcing it has some benefits. For example, the date column of the data set is of data type character, even though all observations are dates. It is beneficial to turn them into proper datetimes instead. This does not only save memory (dates are internally stored as numbers), it also allows date calculations, such as finding the earliest/latest game.

What about independence? Independence of the variables means that you cannot calculate one column from another one. Again, your data violates this principle, as for example the weekday column data$wday can be completely inferred from the data$date column.

Why is the independence of the columns such a precious property? Imagine your date column would suggest the game happened on a Friday, but the weekday column were to say it was a Tuesday. Which one would you trust? Having only one definite value will build **consistency** into your design scheme, leading to **data integrity**[2].

Lastly, it is important to understand the difference between variables and values, and a common issue is having values as columns. Have a look at a non-related data example below, where a table is expanded every year in the column direction.

[2]: Data integrity does not necessarily mean that all data is correct, merely that no obvious arrear can appear.

Imagine you want to store the number of new players in each year in a table such as this:

| **2010** | **2011** | **2012** |
| --- | --- | --- |
| 100 | 120 | 140 |

During visualization, you might like to add coloring dependent on the year of the data. As the values are spread over many columns, you would have to hard-code coloring rules, which is time-consuming and prone to errors.  
Perhaps even more problematic, the number of values, and therefore the number of columns may change.

In the example above it is quite clear that at some point the values for the year 2013 will come in. This means that the overall structure of the table would not be consistent over time, which is a liability. Effectively, you and probably others will have to remember to make the change to the code in time. There is a real chance that this change in table structure will cause some other code that works on the data not to run anymore. Untidy datasets are more likely to be expensive, difficult and cumbersome to maintain.

If the table had instead the form:

| **year** | **values** |
| --- | --- |
| 2010 | 100 |
| 2011 | 120 |
| 2012 | 140 |

then a new value is appended as another independent observation, just as the tidy format suggests.

| **year** | **values** |
| --- | --- |
| 2010 | 100 |
| 2011 | 120 |
| 2012 | 140 |
| 2013 | 180 |

Have another look at the data. Imagine that newer versions of the game could introduce more units, factions, or special characters to the game. This problem is especially prevalent when columns start with stats.kills or stats.recruit and end with the unit type killed or recruited during the game. Any code that makes use of the column location in the original data will be affected by game-changing upgrades.

It makes sense to change the format of the data, so that you can find out which units were recruited/killed in this game, instead of having NAs for all available units of all factions not present in this game. In the next section, you will see how to change the data structure using tools from the tidyverse. This process will get rid of many of the columns that create NA clutter in the data frame, reduce memory consumption during analysis, and make your data easier to gather, inspect, summarize and plot.

**3. *Each table describes one thing.***

Most of the time not all data fits into one table and it is desirable to split a data set into many smaller tables. These tables are then related via the items with each other. The process of breaking up the tables is also called data normalization. In data, all information about one game is squeezed into two rows, specifically one for each player. It does make more sense for the outcome of each game to be single row entry. This single row entry might have references in other tables that can be unfolded and queried when necessary.

But let us start small and with an example.

The units a player recruits in a game could be found in a table with as many rows per game as there are different units the player recruit.

| **game\_id** | **player\_id** | **unit\_name** | **num\_recruited** |
| --- | --- | --- | --- |
| 1 | 1 | A | 2 |
| 1 | 1 | B | 2 |
| 1 | 1 | C | 1 |
| 1 | 2 | A | 2 |
| 1 | 2 | B | 4 |
| 2 | 3 | A | 1 |
| 2 | 3 | B | 4 |
| 2 | 4 | A | 1 |
| 2 | 4 | B | 2 |
| 2 | 4 | C | 1 |
| 2 | 4 | D | 1 |

On the other hand, the table that contains the information about which player has won is a table with a single row per game.

| **game\_id** | **first\_player\_wins** |
| --- | --- |
| 1 | 0 |
| 2 | 1 |

There is simply no way of combining the data into one table without redundant data entries or many NAs.

Splitting a big table into many smaller ones gives the data the freedom to fit a flexible structure. Each of those individual tables can be tidy and without unnecessary NAs. It would be straightforward to calculate summaries and derived quantities. The unique entries of these tables can be referenced with specific identifiers, also called primary keys, in other tables. Although sometimes considered to be a additional point to make data tidy, these relations that connect the content of different tables are really data in themselves and therefore must be stored.

At this point you might wonder whether it is inconvenient to have your data separated into many small tables, as you want to do an analysis on all the data at once. This ultimately depends on the kind of analysis you want to do. Keep in mind that you rarely know beforehand what analysis will be important for you or your audience. On the level of single tables, you can get a lot of insight, formulate hypotheses and re-combine/mix data from different tables with relatively easy queries to answer new questions.

Compare data analysis to the art of cooking. Sure, it would be convenient to have flour, eggs and milk mixed in a certain ratio like ready-made baking mixtures. Yet, great chefs store the best ingredients for their recipes separately and only mix them shortly before cooking. That might seem at times a bit more cumbersome, but also allows for more flexibility and greater freedom to experiment.

**Tidying the game data**

Using these three *tidy* principles as a guide you can start thinking about what tables are necessary for your data set to be *tidy*.

A good starting point is the players. Each game has two of them and each player can play several games. This suggests that you create a player lookup table, also called the bridge table, in which each player has a unique number associated with him or her. As a little bonus, more 'expensive' character type data is stored as cheaper numbers instead. You will use row\_number to create a player ID. You might rightly observe that using the factor function would also store characters as numbers, but it is here slightly more educational to use the new ID as a direct reference.

In the following sections you will make heavy use of the pipe operator %>%. It is a simple tool to make your code more readable. The pipe automatically moves the output of one line of code into the first argument of the next line.

# You create a new table called player\_lookup

player\_lookup <- data %>%

# Chose player column

select(player) %>%

# Ensure that each player has precisely one ID by

# Removing duplicates

unique() %>%

# The row\_number function simply returns the unique row number

mutate(player\_id = row\_number())

# Check the dimension of the new table

dim(player\_lookup)

*## [1] 2738 2*

# It is a good idea to check a few values at the beginning and

# End of your new table

player\_lookup %>% head()

*## # A tibble: 6 x 2*

*## player player\_id*

*##*

*## 1 MasterBao 1*

*## 2 norrington 2*

*## 3 Bonobo 3*

*## 4 whatchair 4*

*## 5 Xplorer 5*

*## 6 Fanjo 6*

player\_lookup %>% tail()

*## # A tibble: 6 x 2*

*## player player\_id*

*##*

*## 1 MAst 2733*

*## 2 Luther 2734*

*## 3 puukameli 2735*

*## 4 Xom 2736*

*## 5 Pentarctagon 2737*

*## 6 fermiore 2738*

Next, you want to replace all references to the players with their respective shorter numeric indices. This way you can capitalize on the more efficient memory usage. To do this, you will use a so-called left join. A left\_join will attempt to match every occurrence of value on the left side with all possible matches of the new data frame from the right-hand side. It does not only match the first value, which is why it is so important to have unique values in the player table. If no match is found, the entry from the left table is still kept. This way, no data from the original table is lost, which is why such a join operation is sometimes labeled as a *non-filtering join*.

data <- data %>%

# You indicate that you want to match by the 'player' column

left\_join(player\_lookup, by = c('player'='player')) %>%

# You do not need the original 'player' column anymore,

# The IDs of the player\_lookup table are a good replacement

# Use select with a minus '-' to remove columns

select(-player)

# Check that the IDs are indeed in the table now

glimpse(data$player\_id)

*## int [1:54591] 1 2 3 4 5 6 7 8 9 10 ...*

There other columns in the data that reference the player name. You will remove these names there as well. As this data set is rather large, you might also want to consider changing the default display options of your tibbles.

data <- data %>%

# Create a new variable with the ID of the winner

left\_join(player\_lookup %>%

rename(winner\_id = player\_id), by = c('winner' = 'player')) %>%

select(-winner) %>%

# Do the same thing for the loser as well

left\_join(player\_lookup %>%

rename(loser\_id = player\_id), by = c('loser' = 'player')) %>%

select(-loser)

# Modify the tibble print size to fine-tune how many rows you want to see by default

options(tibble.print\_min = 10)

options(tibble.print\_max = 20)

# Check on the new columns

data %>% select(player\_id, winner\_id, loser\_id)

*## # A tibble: 54,591 x 3*

*## player\_id winner\_id loser\_id*

*##*

*## 1 1 2 1*

*## 2 2 2 1*

*## 3 3 4 3*

*## 4 4 4 3*

*## 5 5 5 6*

*## 6 6 5 6*

*## 7 7 7 8*

*## 8 8 7 8*

*## 9 9 9 10*

*## 10 10 9 10*

*## # … with 5.458e+04 more rows*

Take note that you have overwritten your data table. You might object, as it seems that you are losing data. However, as long as you do not change the source file, which you should never do, you have not lost anything. There are real benefits to keeping the overall number of tables small. It is good to not store many large data objects in memory, as this can, in extreme cases, cause performance issues. Another reason is that you would accumulate copies of the same data, some of it stale. You might accidentally write code that works on a data copy. When you test the code it seems to run fine, but only because you have in the R environment still a correct version of the object. If you restart the session, and you will have to at some point, there is a chance that your code suddenly breaks because the correct data is missing. Debugging such code after a long work session can be frustrating, as the errors are often subtle.

If you overwrite your data you guarantee that the transformations are applied on all of your data consistently. If you found you made a mistake, you simply restart the session and run it up to the point where you made the change. Flushing your session every so often is a good way to prevent stale memory of the system to build up. This will reduce unnecessary development work and headaches. If you are worried about old portions of code that you might or might not use in the future, use a version control system like git. You can keep your code base free of dead code, while still being able to recover your brilliant ideas from the older versions. This makes working on your code easier in the long run. Cleanliness is next to godliness.

Often you want to craft more refined code, such as the one further below. It is worth noting that such code is the result of many iterations of executing code line by line, and verifying that the steps have led to the desired outcome. By keeping your code base tidy, you give yourself more freedom to experiment while not having to worry that data will be lost.

Let us remove some of the derived date columns like the weekday, month and year. These columns are of no use immediately, but may lead to consistency issues, for instance when different dates would not correspond to the same weekday. You can use the opportunity to enforce some of the proper column types. It is sensible to enforce categorical data using the factor function, as it simplifies some analyses somewhat, but also reduces the amount of memory needed.

data <- data %>%

# Enforce variable type and remove calculated features

mutate(date = ymd\_hm(date)) %>%

select(-c('year','mon','wday' )) %>%

mutate(color = factor(color),

map = factor(map),

era = factor(era),

faction = factor(faction),

leader = factor(leader),

team = factor(team),

version = factor(version))

As you have separated out some of the player data, you can focus on what tables you should build next. There is a lot of repetitive meta information that can be outsourced to smaller tables. Just as each player is an entity that has some characteristics, so is every game. It is sensible to have one table that contains all the meta information of each game, as this reduces some of the two-row redundancy exhibited by the original data set.

game\_info <- data %>%

# Grouping is a good way to reduce the information

group\_by(game\_id) %>%

# Using first() or last() is a little trick to pick

# Only one of the (ideally) redundant data bits

summarise(turns = first(turns),

date = first(date),

map = first(map),

title = first(title),

game\_file = first(game\_file),

# The number is the only quantity not redundant in each row

num\_players = max(number))

As game\_info by itself is already tidy, you can get some useful information regarding the distribution of number of players in games.

game\_info %>%

group\_by(num\_players) %>%

# The n() function counts all non-NA entries

summarise(n\_games = n())

*## # A tibble: 4 x 2*

*## num\_players n\_games*

*##*

*## 1 1 2*

*## 2 2 27286*

*## 3 4 3*

*## 4 5 1*

Clearly, the vast majority of games is two-player. The one-player games might be an issue related to missing data, which should be investigated further. Apparently, somebody managed to report a 5 player game as well, but overall numbers are very small. At this point, you might ask whether it would be sensible to create a lookup table for the maps and player colors and so on. The answer is: *it depends*.

Database purists who insist on full normalization (essentially the concept of tidy data applied to relational databases) would say yes. While this may be true, you know that each game has only one map, so keeping the map information as game metadata seems practically sufficient.

By encoding the map as a factor, you have, under the hood, already translated the map into an integer representation. No real practical gains in terms of memory would be made. By keeping things as they are, you reduce the overall number of tables necessary to describe the data, which is a good thing. You will have fewer objects stored in your environment and it is worth aiming for this simplicity.

Because two-player games are so prevalent, and you do not alter the underlying raw data, you could make the executive decision to focus on two-player games only. They have a somewhat simplified structure, as there are only one winner and one loser. First, you find all the game IDs that belong to those games and then you encode which of the two players wins the game. This way each two-player game occupies only one row in the data table and outcomes are boolean. This new variable can later serve as a target for learning algorithms.

# Extract the two player games into a vector

two\_player\_game\_ids <- game\_info %>%

filter(num\_players == 2) %>%

# The pull function take a column and returns it as a ready-to-use vector

pull(game\_id)

two\_player\_games <- data %>%

select(game\_id, player\_id, winner\_id) %>%

# The %in% operator is used to filter by vector membership

filter(game\_id %in% two\_player\_game\_ids) %>%

# The two-player structure allows you to simply use

# first() and last() to uniquely identify each player

group\_by(game\_id) %>%

summarise(first\_player\_id = first(player\_id),

second\_player\_id = last(player\_id),

winner\_id = first(winner\_id)) %>%

# The winner\_id must be either the first or last player,

# So you only need to compare it with the first player ID

mutate(first\_player\_wins = if\_else(first\_player\_id == winner\_id, TRUE, FALSE)) %>%

select(-winner\_id)

two\_player\_games %>% head()

*## # A tibble: 6 x 4*

*## game\_id first\_player\_id second\_player\_id first\_player\_wins*

*##*

*## 1 1 1 2 FALSE*

*## 2 2 3 4 FALSE*

*## 3 3 5 6 TRUE*

*## 4 4 7 8 TRUE*

*## 5 5 9 10 TRUE*

*## 6 6 11 12 FALSE*

You can also get a quick overview by running summary on the new tibble.

summary(two\_player\_games)

*## game\_id first\_player\_id second\_player\_id first\_player\_wins*

*## Min. : 1 Min. : 1.0 Min. : 1.00 Mode :logical*

*## 1st Qu.: 6823 1st Qu.: 69.0 1st Qu.: 75.25 FALSE:12810*

*## Median :13646 Median : 179.0 Median : 195.00 TRUE :14475*

*## Mean :13646 Mean : 358.9 Mean : 390.52 NA's :1*

*## 3rd Qu.:20469 3rd Qu.: 440.0 3rd Qu.: 491.00*

*## Max. :27292 Max. :2736.0 Max. :2738.00*

Hm, interesting. Overall, it seems that the first player is more likely to win than the second, an insight that could be quite useful in future modeling.

Next, you can look at individual players ranking. The rank of a player is the Elo number assigned after the game. The Elo of a player is a measure of his or her strength relative to other players and changes over time. Simply put, you do not want a change in Elo to inform us whether somebody will win or not. The Elo in each row is the value *after* a game. It is clear that the Elo will correlate with a win, perhaps only weakly. To have the data from after the game is a subtle form of data creep. Before the game, you do not have that knowledge, so if you were to train a model on that data, you might be overly confident in your ability to predict the outcome.

As a *tidy* alternative, it makes sense to store all the Elo numbers for a player and the time of the recording of that Elo in one table. The change in Elo of an individual player can then be tracked better and you can easier enforce that only the latest *known* value is used for any predictions.

The Elos appear in two positions in the data, you can hence use the bind\_rows function to create a table of the combined rankings. The Elo of a player at the end of a game is independent of whether he or she won or lost that game and the Elo table should not be divided into winners and losers.

winner\_elos <- data %>% select(player\_id = winner\_id,

elo = winner\_elo,

date)

loser\_elos <- data %>% select(player\_id = loser\_id,

elo = loser\_elo,

date)

elos <- winner\_elos%>%

bind\_rows(loser\_elos) %>%

# Enforce that Elo is a number

mutate(elo = as.numeric(elo)) %>%

# Remove potential duplicates

distinct()

elos %>% head()

*## # A tibble: 6 x 3*

*## player\_id elo date*

*##*

*## 1 2 1672 2014-07-01 14:55:00*

*## 2 4 1884 2014-03-05 08:20:00*

*## 3 5 1775 2012-02-03 16:31:00*

*## 4 7 1759 2014-02-24 07:47:00*

*## 5 9 1956 2012-01-28 15:53:00*

*## 6 12 1785 2015-11-16 21:06:00*

After processing all the players and games, you can now attempt to capture information that is particular for each player-game combination. Most of that information is stored in columns that start with stats but are not yet the unit statistics.

player\_game\_statistics <- data %>%

select(game\_id, player\_id, color, team,number, faction, leader,

cost = stats.cost,

infliced\_expected = stats.inflicted\_expected,

inflicted\_actual = stats.inflicted\_actual,

taken\_expected = stats.taken\_expected,

taken\_actual = stats.taken\_actual

)

player\_game\_statistics %>% head()

*## # A tibble: 6 x 12*

*## game\_id player\_id color team number faction leader cost*

*##*

*## 1 1 1 teal north 1 Drakes Drake… 271*

*## 2 1 2 oran… south 2 Loyali… Lieut… 274*

*## 3 2 3 red west 1 Northe… Orcis… 384*

*## 4 2 4 teal east 2 Undead Death… 415*

*## 5 3 5 blue south 1 Drakes Drake… 188*

*## 6 3 6 red north 2 Loyali… White… 192*

*## # … with 4 more variables: infliced\_expected ,*

*## # inflicted\_actual , taken\_expected , …*

What do these columns mean?

The inflicted expected/ inflicted actual distinction exists due to the random nature of the game. All attacks are stochastic. A die is rolled whether an attack misses or not, and so is the damage dealt. It could be interesting to analyze how the lucky of a player, i.e. how much damage he or she inflicted or has taken compared to what was expected during the game, and whether this has a measurable influence on winning or losing the game as a result.

The last big task is to extract the unit data, which is responsible for the majority of the columns. As the unit name is part of the column name, you can use *regular expressions* to extract the name. They are essentially little pieces of code that allow you to extract specific patterns out of larger strings.

The structure of the named columns is 'stats.name\_of\_statistic.name\_of\_unit' and you want to extract 'name\_of\_statistic' and 'name\_of\_unit' without working through 500 columns by hand. The rebus package is very convenient for constructing larger regular expressions out of smaller parts, which makes it that much easier to reason about them. Remember that regular expressions are piped with the %R% operator, rather than the standard pipe %>%.

statistic\_pattern <- START %R% # Start of column name

'stats.' %R% # Uninformative identifier

capture(one\_or\_more(WRD)) %R% #Captures the statistic

'.' %R% # The dot separates the components

capture(one\_or\_more(WRD)) %R% # Captures the name of the unit

END

# R sees this

statistic\_pattern

*## ^stats.([\w]+).([\w]+)$*

# we test our pattern on a dummy string

test\_string <- 'stats.sample\_statistic.sample\_unit'

# looks good

str\_match(test\_string, statistic\_pattern)

*## [,1] [,2] [,3]*

*## [1,] "stats.sample\_statistic.sample\_unit" "sample\_statistic" "sample\_unit"*

The different columns of the output of str\_match correspond to the various captured parts.

You have to transform the data set from the wide format, where columns correspond to values, to the long format, where the columns represent variables. You do this with the gather function. With key, you specify the name of the new variable that is made up of the column names, and with value, you specify the name of the new variable that holds the original values.

Once that is done, you can match the statistic\_pattern to the new key variable, which is the original column name, to extract the unit names and the statistics being described.

# The statistics columns that are already treated above

player\_statistics\_cols <- c('stats.cost',

'stats.inflicted\_actual',

'stats.inflicted\_expected',

'stats.taken\_actual',

'stats.taken\_expected')

player\_unit\_statistics <- data %>%

select(game\_id, player\_id, starts\_with('stats')) %>%

select(-player\_statistics\_cols) %>%

# game\_id and player\_id are the effective indices of the new table

gather(key = description, value = number, -game\_id,-player\_id) %>%

# Keep only non-empty values to reduce the overall size

filter(!is.na(number)) %>%

# Parsing of the column names

mutate(statistic = str\_match(description, statistic\_pattern)[,2],

unit = str\_match(description, statistic\_pattern)[,3]) %>%

select(-description)

player\_unit\_statistics %>% head()

*## # A tibble: 6 x 5*

*## game\_id player\_id number statistic unit*

*##*

*## 1 1 1 1 advances saurian\_soothsayer*

*## 2 18 33 1 advances saurian\_soothsayer*

*## 3 40 68 1 advances saurian\_soothsayer*

*## 4 44 76 1 advances saurian\_soothsayer*

*## 5 72 119 1 advances saurian\_soothsayer*

*## 6 94 143 1 advances saurian\_soothsayer*

Now that you have a list of all units, you can create a lookup just as you did for the players and the games.

unit\_lookup <- player\_unit\_statistics %>%

select(unit) %>%

distinct() %>%

mutate(unit\_id = row\_number())

unit\_lookup %>% head()

*## # A tibble: 6 x 2*

*## unit unit\_id*

*##*

*## 1 saurian\_soothsayer 1*

*## 2 longbowman 2*

*## 3 drake\_burner 3*

*## 4 drake\_clasher 4*

*## 5 drake\_fighter 5*

*## 6 drake\_glider 6*

The unit information here is rather bare bones. Naturally, all of this data is best stored in tidy data too.

The rvest package is ideal to scrape this information.

Now that each unit has an ID assigned, it makes sense to replace the unit name with the unit ID in the statistics table.

player\_unit\_statistics <- player\_unit\_statistics %>%

left\_join(unit\_lookup, by = c('unit')) %>%

select(-unit)

You can even split the stats into the relevant categories.

advances\_stats <- player\_unit\_statistics %>%

filter(statistic == "advances") %>%

select(-statistic)

deaths\_stats <- player\_unit\_statistics %>%

filter(statistic == "deaths") %>%

select(-statistic)

kills\_stats <- player\_unit\_statistics %>%

filter(statistic == "kills") %>%

select(-statistic)

recruits\_stats <- player\_unit\_statistics %>%

filter(statistic == "recruits") %>%

select(-statistic)

**Combining everything into one function**

Up to this point, you cleaned a data set by breaking it into several smaller, tidy sets. That was quite cumbersome, but it served a purpose. What happens when new data is released? You do not want to rerun all the steps one-by-one. Or imagine, you want to move the data cleaning process into a larger pipeline in your company. For all these requirements it is best to write a function. Functions are great, as they separate the data from the calculations on that data. They allow the data to flow through your applications and are maintainable.

If your cleaning process changes, but your functional interface is the same, i.e. the structure of the inputs and outputs does not change, then other processes will keep working without any additional intervention. You can simply rerun the function again and the data flow persists.

To write the function you need to wrap the calculations into functional arguments and think about the inputs and outputs. It is reasonable to write default parameters that can later be overwritten. This way, your function is convenient most of the time, and flexible when needed. Here, your input is the path to the data that you imported and an additional parameter that will control the output. The main part are the pieces of code that you already wrote above.

load\_wesnoth <- function(path = 'Wesv15.csv', return\_object = FALSE){

# If the location of the data changes, a new path can be passed

# Without changing the architecture of the function itself

data <- read\_csv(path)

# All the code from above that processed the data is copied into this section:

# [...]

# A list object that contains all the items you want to export

# and return when you load the function in a next step

to\_export <- list('player\_lookup'= player\_lookup,

'units\_lookup' = units\_lookup,

'game\_info' = game\_info,

'two\_player\_games' = two\_player\_games,

'player\_game\_statistics' = player\_game\_statistics,

'elos' = elos,

'advances\_stats' = advances\_stats,

'deaths\_stats' = deaths\_stats,

'kills\_stats' = kills\_stats,

'recruits\_stats' = recruits\_stat)

# You have the choice of whether the object is returned as a list object

# containing the relevant data, or is directly loaded into the R environment

if (return\_object){

return(to\_export)

} else{

list2env(to\_export, envir = .GlobalEnv)

}

}

The return\_object keyword gives you control over whether the tables are returned as a list object in which each element corresponds to a table, or whether the tables are individually loaded into the R environment. The latter is certainly convenient but also clutters. Besides, you could potentially overwrite other objects in the environment, which is an unwanted side effect. If you save the function as a script, for example, called tidy\_function.R, then you can simply source the file and load the data in a single line.

# Import the function

source('tidy\_function.R')

# Clean the data

cleaned\_data <- load\_wesnoth(return\_object = T)

# Check that the returned object is what you expect it to be

cleaned\_data$two\_player\_games %>% head()

*## # A tibble: 6 x 4*

*## game\_id first\_player\_id second\_player\_id first\_player\_wins*

*##*

*## 1 1 1 2 0*

*## 2 2 3 4 0*

*## 3 3 5 6 1*

*## 4 4 7 8 1*

*## 5 5 9 10 1*

*## 6 6 11 12 0*

You can also load the data directly into the R environment.

load\_wesnoth()

*##*

two\_player\_games %>% head()

*## # A tibble: 6 x 4*

*## game\_id first\_player\_id second\_player\_id first\_player\_wins*

*##*

*## 1 1 1 2 0*

*## 2 2 3 4 0*

*## 3 3 5 6 1*

*## 4 4 7 8 1*

*## 5 5 9 10 1*

*## 6 6 11 12 0*

Instead of loading the function from a script, you could also package it into your own R package that can be easily shared with others.

Congratulations, you now have the data in an easy-to-load and ready-to-analyse state.

**A sample analysis**

What is even more fun than cleaning your data? Analyzing it! You can start answering some simple questions about the data. Which player has played the most games?

player\_game\_statistics %>%

group\_by(player\_id) %>%

summarise(num\_games = n()) %>%

arrange(desc(num\_games)) %>%

head()

*## # A tibble: 6 x 2*

*## player\_id num\_games*

*##*

*## 1 3 1227*

*## 2 101 949*

*## 3 6 868*

*## 4 33 809*

*## 5 130 808*

*## 6 84 712*

The player with player\_id == 3 has the most games played. What is his win ratio?

win\_first\_position <- two\_player\_games %>%

filter(first\_player\_id == 3) %>%

summarise(wins = sum(first\_player\_wins),

games\_played = n())

(win\_first\_position)

*## # A tibble: 1 x 2*

*## wins games\_played*

*##*

*## 1 437 618*

win\_second\_position <- two\_player\_games %>%

filter(second\_player\_id == 3) %>%

summarise(wins = sum(!first\_player\_wins),

games\_played = n())

(win\_second\_position)

*## # A tibble: 1 x 2*

*## wins games\_played*

*##*

*## 1 391 609*

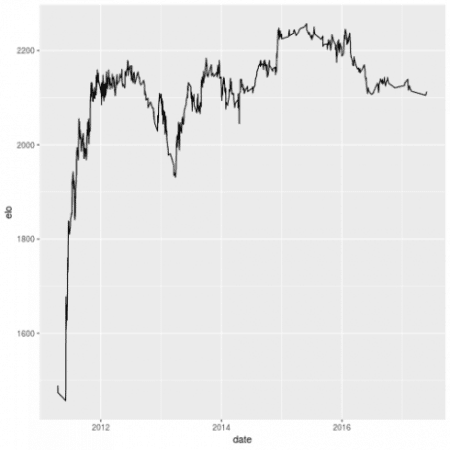
The total win rate (summing up all games won as player1 and as player2 and divided by the total games played: (437+391)/(618+609) = 0.6748166, a very solid player indeed. How did the Elo of that player change over time?

elos %>%

filter(player\_id == 3) %>%

ggplot(aes(x = date, y = elo)) +

geom\_line()

[](https://i2.wp.com/datascienceplus.com/wp-content/uploads/2019/04/unnamed-chunk-33-1-1.png?ssl=1)

His Elo rose quickly within the span of half a year and then more or less plateaued with occasional peaks and troughs. From 2015 on it seems to have been slightly on the decline.

Another quite interesting thing to look at are the average army compositions. What units do competitive players recruit?

player\_game\_statistics %>%

select(game\_id, player\_id, faction) %>%

inner\_join(recruits\_stats, on = c('player\_id', 'game\_id')) %>%

# Some units are shared between factions, hence the inclusion

# Of faction in the group\_by

group\_by(faction, unit\_id) %>%

# You summarise over all units ever recruited

summarise(total\_recruited = sum(as.numeric(number))) %>%

left\_join(units\_lookup, on = 'unit\_id') %>%

# Normalise

mutate(recruited\_fraction = total\_recruited/sum(total\_recruited)) %>%

# Sometimes units are recruited irregularly, this filter takes care of that

filter(total\_recruited > 500) %>%

# Visualisation parameters

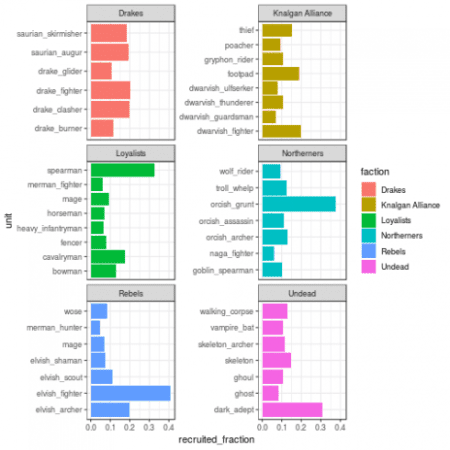
ggplot(aes(x = unit, y = recruited\_fraction, fill = faction)) +

geom\_bar(stat = 'identity') +

facet\_wrap(faction ~ ., nrow = 3, ncol = 2, scales = 'free\_y')+

coord\_flip() +

theme\_bw()

[](https://i1.wp.com/datascienceplus.com/wp-content/uploads/2019/04/unnamed-chunk-34-1-1.png?ssl=1)

You see some clear preferences. For instance, there are units which are recruited with a high occurrence in certain factions, such as spearmen, orcish grunts, elvish fighters or dark adepts. Those are all good staple units and should not be missing in any army. The Knalgan Alliance and Drakes faction, however, seem to require all units from their recruiting pool to play most efficiently. It is likely that there are sub-strategies for particular faction matchups or maps, which you could investigate further!

**Conclusions**

You learned to structure your data for analysis. If your data is not tidy, which most of the time will be the case, you know what needs to be done to clean it. During the cleaning process, you will look at tables and run simple summaries to get a feeling for the data. In this way, you will get to know your data more intimately, and you will add value faster in your role as a data analyst or data scientist.

When the data is tidy, you can visualize the data by inspection. Here is where you often get your first ideas for new hypotheses to test or features to engineer into your dataset, which are prerequisites for good data models. By keeping your data and your code tidy you make it easier for other people to work with your data and analyze your code.

Finally, you now have a very interesting data set for further analysis. You may use it for your own advantage in the game or to get better at doing data science. Whatever your next steps may be, go and check out Wesnoth.