Recitation 7 - DAGs and Twitter data

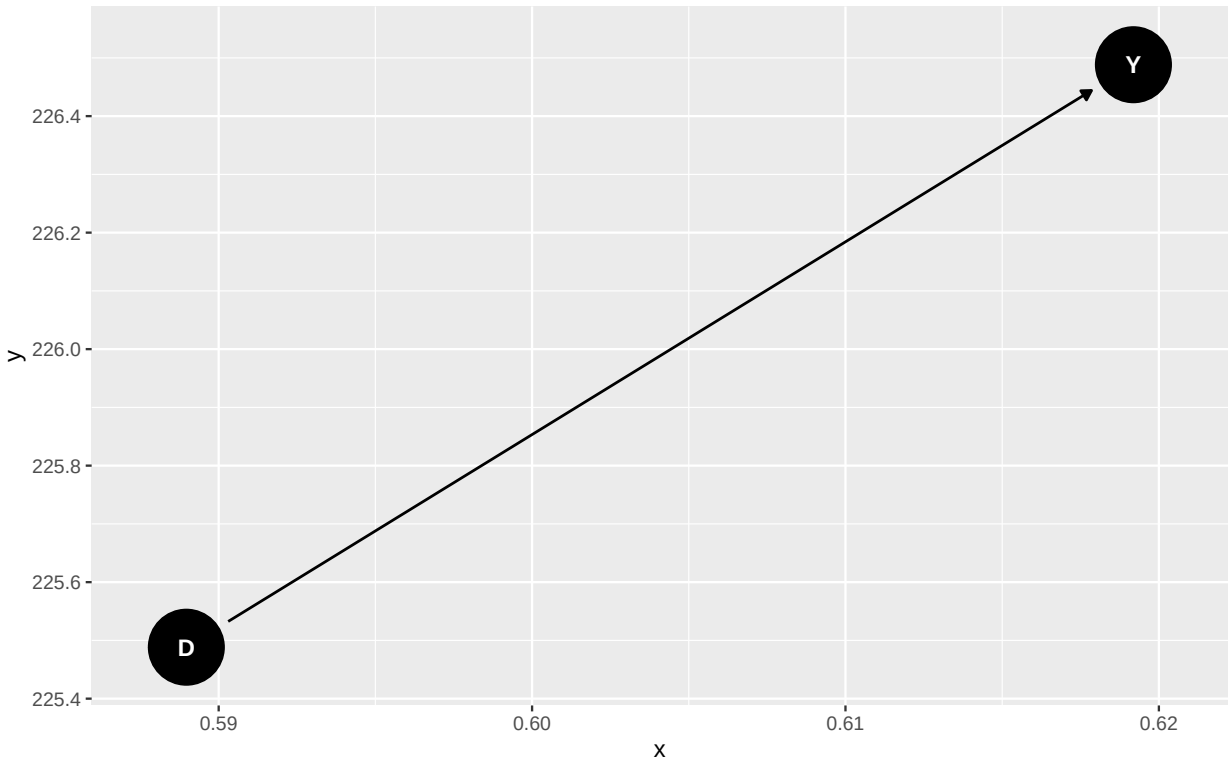
Seamus Wagner

October 8, 2021

DAGs

Directed Acyclic Graphs are a useful visualization tool for thinking about your theory and how your variables relate. I don't tend to use them in R often, but writing them out is a common practice for me. We will be using the package ggdag, which takes objects already in a DAG structure. ggdag uses the syntax of dagitty, which is built off of graphviz syntax. You do not need to load dagitty to use ggdag, but you can and make calls from that package. There are also ways to use igraph and other types of packages. ggdag is relatively new and much much easier to graph with. I recommend checking out the dagitty and ggdap help contents for some quick examples of what you should know about base syntax. I had to reinstall Rcpp before these worked. My computer would run R until crash before I figured out this was the issue.

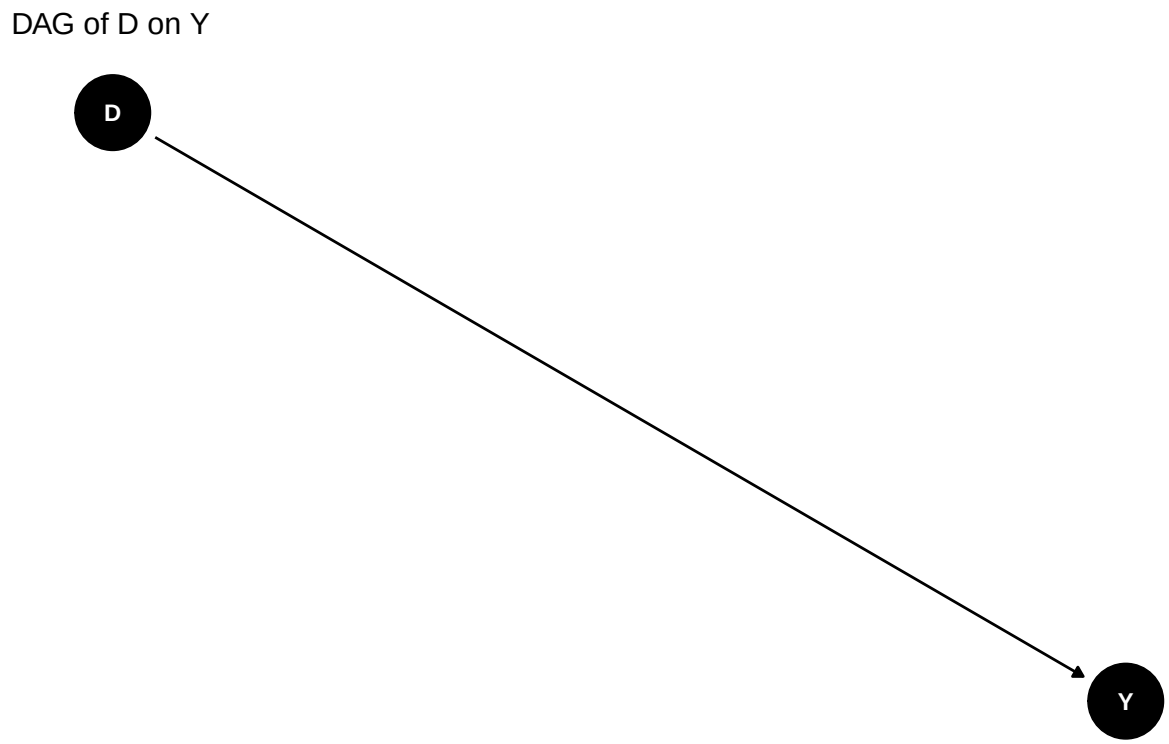
```
dag1 <- dagify(Y ~ D)
ggdag(dag1)
```



```
#?dagitty
```

We can create a simple DAG quite easily here, but this is quite ugly and has a lot of additional info on the image that is unecessary. Let's clean that up.

```
ggdag(dag1) +
  theme_dag_blank() +
  theme(axis.title = element_blank(),
        axis.text = element_blank()) +
  ggtitle("DAG of D on Y")
```



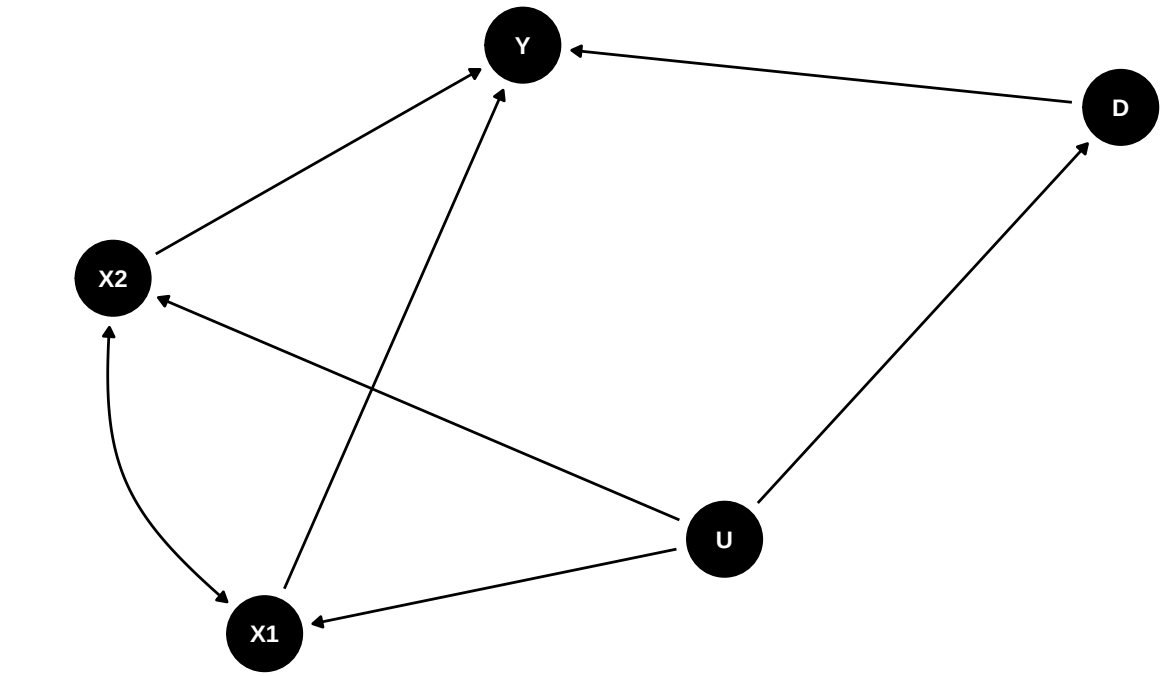
As we see here, it is much cleaner and we can add all sorts of labels, legends, and titles using ggplot syntax. Let's make a slightly more complex DAG and introduce some common features. For the dagify syntax, at least one way to write it, you can write each line, comma separated, for each node. FOr instance,

$$Y \ X1 + D$$

will create nodes Y, X1, and D and both X1 and D will be ancestors of Y.

```
dag2 <- dagify(
Y ~ D + X1 + X2,
D ~ U,
X1 ~ U,
X2 ~ U,
X1 ~~ X2 #Not D-separated here
)
ggdag(dag2) +
```

```
theme_dag_blank() +
theme(axis.title = element_blank(),
      axis.text = element_blank()) +
ggtitle("DAG of D on Y controlling for X1, which mediates U")
```



Twitter scraping

Using Twitter as a data source is useful for people interested in networks, perceptions, discourse analysis, sentiment analysis, trends, government engagement, policy strategy, and many other topics. Twitter is not as useful for making claims about behavior of individuals or anything to do with demographics.

Before you start scraping, you need API keys to use for accessing Twitters API. You do not really need to learn the language of their API for the R packages. Python is an equally (more) popular software for scraping, but I do not know Python. That being said, looking into it won't hurt.

There are two main R packages for Twitter scraping, rtweet and academictwitteR. rwteet is the more tenured package but does not support scraping academic Twitter since the implementation of the 2.0 API. I do not know if there are plans to update it. Until there is, if you plan on using the academic credentials, learn academictwitteR or another package that can make pulls properly in the new API. This also means support is more tenuous for the newer packages (July 2021 for this case).

To begin, think about what your goals for Twitter data are. I was using historical data, which you need academic developer permissions for. Regardless, I think applying for the academic permissions is a good idea. It ups your available scrape limits and allows for much more functionality. Check out the Developer Portal for your options. TO do this, you need a decent grasp of what your project is (not level of detail for IRB, but still a decent amount).

Below are some examples of using the historical (full archive) search. Fill the bearer token with yours and do not share those with anyone. The get_all_tweets function allows for a number of queries for how to

narrow your search. The query term allows for hashtags, mentions, or general terms and will return Tweets containing those query terms. The start_tweets and end_tweets functions will narrow the time frame for your queries. In this case, I collected all the tweets before and after the elections for Tanzania in 2020. The data path function is important as it will save your data as a json file. If you do not set a path and also to bind_tweets = F, your data will be stored as a data.frame, which is much less efficient for large objects (millions of tweets). It will return two files, one for the contents of the tweets and one for the user data. I bind them separately after the return. I then clean the text using gsub.

```
library(academictwitteR)
library(rtweet)
library(dplyr)
library(data.table)
library(lubridate)
rm(list=ls())
setwd("E:/Quant 3 TA/Recitations/recitation 7")
```

```
recitation_tweets <- get_all_tweets(
  query = c("accountability", "#KeepItOn", "#TanzaniaDecides2020", "#KeepItOnTz", "#OpenInternet", "#Ta
  start_tweets = "2020-10-21T00:00:00Z",
  end_tweets = "2020-10-28T00:00:00Z",
  data_path = "E:/Quant 3 TA/Recitations/recitation 7/pre_election",
  bind_tweets = FALSE,
  n = 1000,
  is_retweet = F
)
```

```
## query:  (accountability OR #KeepItOn OR #TanzaniaDecides2020 OR #KeepItOnTz OR #OpenInternet OR #Tan
## Total pages queried: 1 (tweets captured this page: 492).
## Total pages queried: 2 (tweets captured this page: 492).
## Total pages queried: 3 (tweets captured this page: 494).
## Total tweets captured now reach 1000 : finishing collection.
## Data stored as JSONs: use bind_tweets function to bundle into data.frame
```

```
tweets <- bind_tweets(data_path = "E:/Quant 3 TA/Recitations/recitation 7/pre_election")
```

```
## ================================================================================
```

```
users <- bind_tweets(data_path = "E:/Quant 3 TA/Recitations/recitation 7/pre_election", user = TRUE)
```

```
## ================================================================================
```

```
tweets_pre <- tweets
users_pre <- users

tweets_gsub_pre <- tweets_pre %>%
  mutate_at(vars(text), function(x){gsub('[^ -~]', '', x)})
```

If we want to do more of a social network or follower approach, we could start this way. For time purposes, I am not going to do a network analytical approach and all of the data work that goes into it. But from here, you could implement one and I am happy to go over it in more detail with any of you during office hours.

```
sc <- lookup_users("SkylerCranmer")
follower <- get_followers("SkylerCranmer")
follow_details <- lookup_users(follower$user_id)
follow_details <- data.frame(lapply(follow_details,as.character),
stringsAsFactors = F)
trimmed_data <- follow_details %>%
  select(c("user_id" , "screen_name", "created_at", "followers_count", "friends_count", "favourites_cou
  mutate(created_at = ymd_hms(created_at), #this converts to a date from year, month, day, hour, minute
         followers_count = as.numeric(followers_count),
         friends_count = as.numeric(friends_count),
         favourites_count = as.numeric(favourites_count))
```

I don't have the time to run this in recitation but the below code will then expand the follower search to the second degree. Once we collect these followers or followers, we could create a vector of shared ids and start to have a more dense and descriptive network. Interactions such as liking, mentions, or retweets could turn it inferential.

```
#get_followers_follower <- function(x){
#  if(x > 500){ #This number is arbitrary,
#    x*0.5
#  }else{x*0.75}
#}
# This is a blank vector for the screen names to be attached to
#follows_follow <- vector(mode = 'list', length = #length(trimmed_data$screen_name))
#names(follows_follow) <- trimmed_data$screen_name
##
#for (i in seq_along(trimmed_data$screen_name)) {
#  message("Getting followers for user #", i, "/130")
#  follows_follow[[i]] <- get_followers(trimmed_data$screen_name[i],
#                                       n = #round(get_followers_follower(trimmed_data$followers_count[i])),
#                                     retryonratelimit = TRUE)
#
#  if(i %% 5 == 0){
#    message("sleep for 5 minutes")
#    Sys.sleep(5*60) #This is vital for the looping to not fail. It still might hit rate limit, but thi
#    }
#}
#
```