Terms of Use

Connor Krenzer

7/15/2021

Introduction

The next time you go on your favorite website, scroll down to the bottom of the page and look at all the links to different sections. One of these will surely be the terms of service agreement that you consent to abide by while using their platform. No one bothers reading them unless he or she plans on doing something that has the potential to bring negative repercussions.

Much of the data I work with originates from web scraping. In the event you didn't know, most companies hate web scrapers for two reasons: (1) because of legitimate data privacy concerns, and (2) because companies want to sell the information that is easily acquired for free by a scraper. To raise barriers between data and people at least somewhat competent with a computer, companies add red tape to their services in the form of a terms of use agreement. If you violate the terms of use–depending on the severity of the offense–you can get anything from a suspension or ban from the company's services, sued, or even a criminal prosecution in very egregious cases. Make no mistake, there is a great deal of data out there that currently is and should continue to be off limits–public or not. It is the second case from which the bulk of my issues originate. Yes, intellectual property rights exist for good reason, but the problem for people like me, however, is that excessive red tape hinders those seeking to answer interesting questions with that data.

The motivation for this investigation stems from the anti-scraping policies so prevalent online. If the companies aren't going to allow people to scrape their content, their terms of use are still fair game! These documents will be used to test algorithms to see whether they confirm what we already know about the companies.

Links to Policies

This section is certain to become outdated in the future, so if you want to see the raw text used in this project, read the text files in the 'data' folder of this project's GitHub repository. The links used to get this information, however, are still provided below:

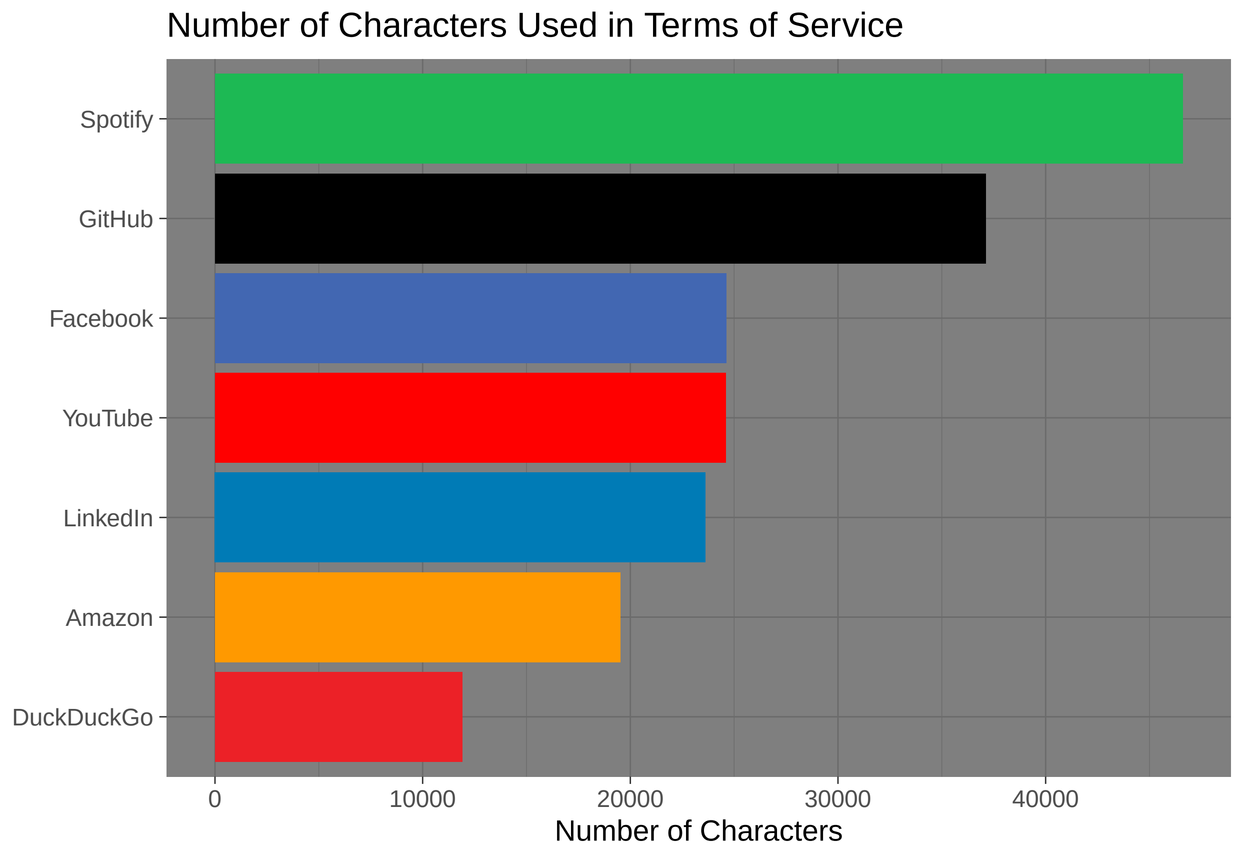
- LinkedIn's policy
- Facebook's policy
- YouTube's policy
- Amazon's policy
- Spotify's policy
- GitHub's policy
- DuckDuckGo's policy

I couldn't find a terms of use agreement for DuckDuckGo, likely because of the lax nature of their rules, so their privacy policy is used in place of a terms of use agreement. DuckDuckGo had to be included in this list as the 'outsider' that essentially lets you do whatever you want with the data collected from its search engine.

Document Lengths

To begin, why don't we calculate the length of each document? If you don't feel like navigating to the web pages hosting these documents yourself, this will save you some time! To calculate the length, the number of characters used in the document are counted. This includes all characters–newlines, punctuation, letters, digits, everything. While this may be considered imperfect, it gives us a 'good-enough' estimate for the length of each terms of use agreement.

Which company has the longest terms of use agreement?



Unsurprisingly, DuckDuckGo has the shortest policy. Does this mean that refusing to track user data equates to needing fewer lawyers? I am inclined to think so, but the above graph is incapable of confirming this suspicion because of differences intrinsic to the services provided by these companies. Further, Amazon–the largest company in the bunch by annual revenue–has a shorter terms of use page than its contemporaries. One explanation among many is that these companies spread legal documents out into different sections of their websites to organize policies into logical chunks.

Data Prep

The Porter stemming algorithm will be used to pre-process the text. While in an industrious mood, I decided to implement the algorithm myself using this article and the algorithm's official web page as guides. After many hours tweaking the algorithm's intricacies, my implementation returns the same result as the official implementation in the SnowballC package 98% of the time when using this dataset.

Differences between the Snowball version and mine include:

- Differences between the logic in the guides used (which walked through the original algorithm) and the slightly modified Snowball algorithm (the Snowball algorithm is also known as the 'Porter2' stemmer to differentiate it from the originally published algorithm).
- Random errors that slip through the cracks due to regular expressions used, **m** calculation rules, and other oddities.
- My version strips contractions before stemming ("aren't" becomes "are", "they'll" becomes "they", etc.).
- My version uses a 'tidy' format–it takes a data frame and column name as input and return the original data frame after stemming the words in the specified column.
- Speed. My version is about 300 times slower than the SnowballC package's implementation. This is primarily due to the copying of the data frame after each dplyr::mutate() call in my code. This difference is not very important here, since my version takes two seconds to run on this data set while Snowball takes .008 seconds. It is noticeably slower, but not enough for it to actually matter for this use-case.

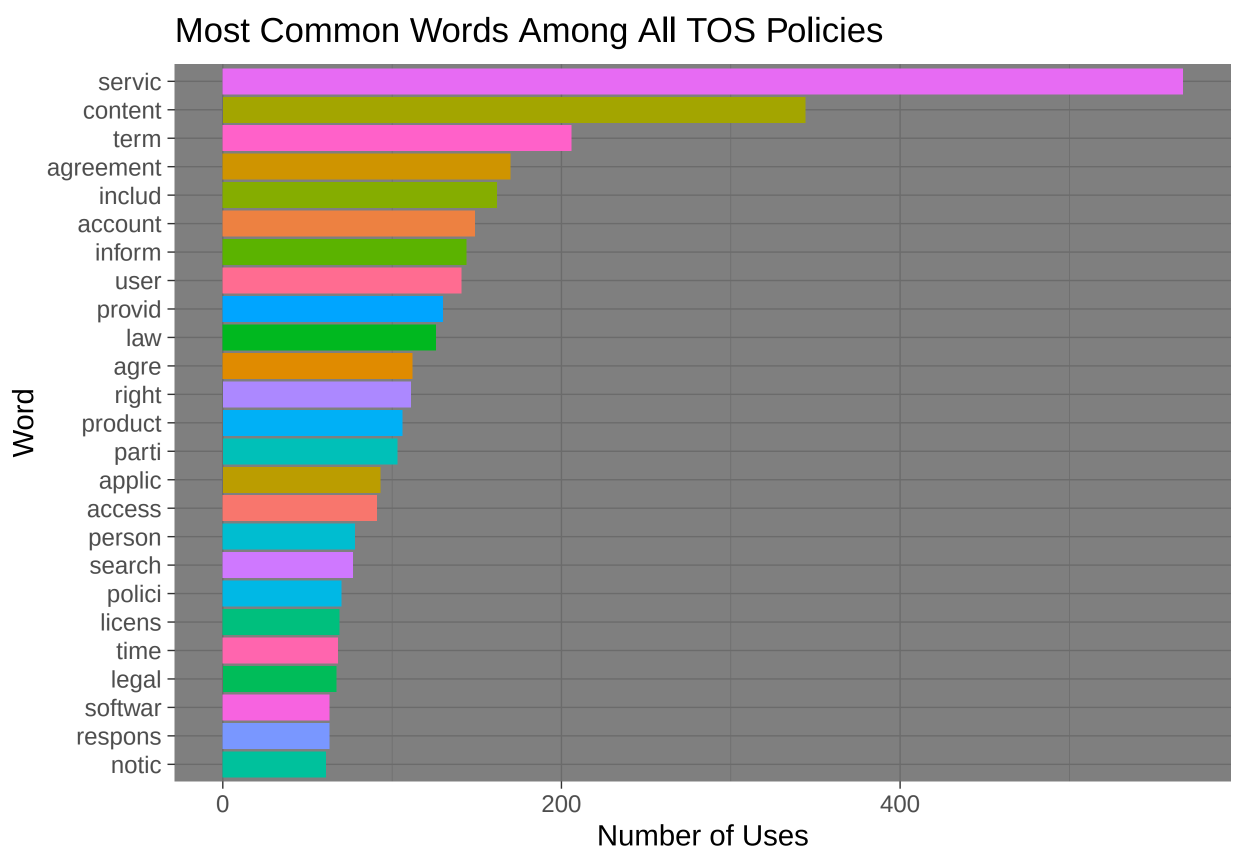
Second, I will be using my own n-gram tokenizer instead of the tidytext package's unnest_tokens(). Several iterations of this algorithm were tried, with each one providing notable differences in performance. Since the theme here is a DIY approach to algorithms, I am using the custom version that provides the most flexibility (f5() from the benchmarking done with these algorithms). Implementing this step was tricky but asking a question on StackOverflow helped to sort things out. My approach to an n-gram tokenizer bases itself on the assembly of the n-grams from unigrams. Using a tokenizer that allows greater flexibility dampened speed harshly; more restricted versions provide speed boosts in excess of an order of magnitude over unnest_tokens().

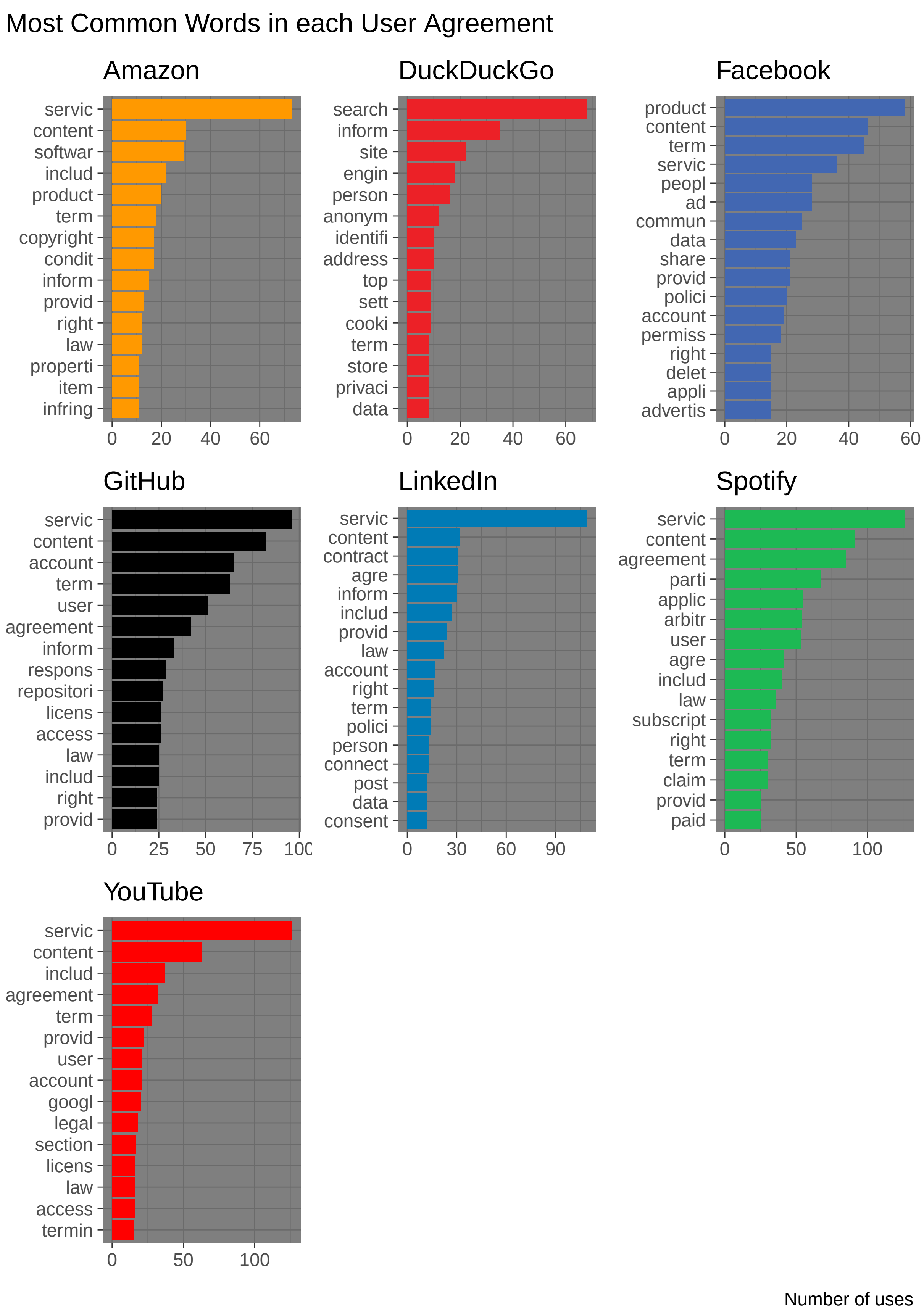
Other algorithms written 'by-hand' include a pairwise counting function (which was important for calculating skipgram probabilities), the TF-IDF statistic, and word2vec (word embeddings). These do not have significant enough differences in end results to provide a play-by-play.

Doing this taught me the value of writing an algorithm yourself; rewriting and implementing an algorithm from scratch is a very effective way to learn how it works. Having the code to run the algorithm handy **and** knowing the algorithm inside and out grants unique opportunities to make improvements that better suit your needs.

Word Counts

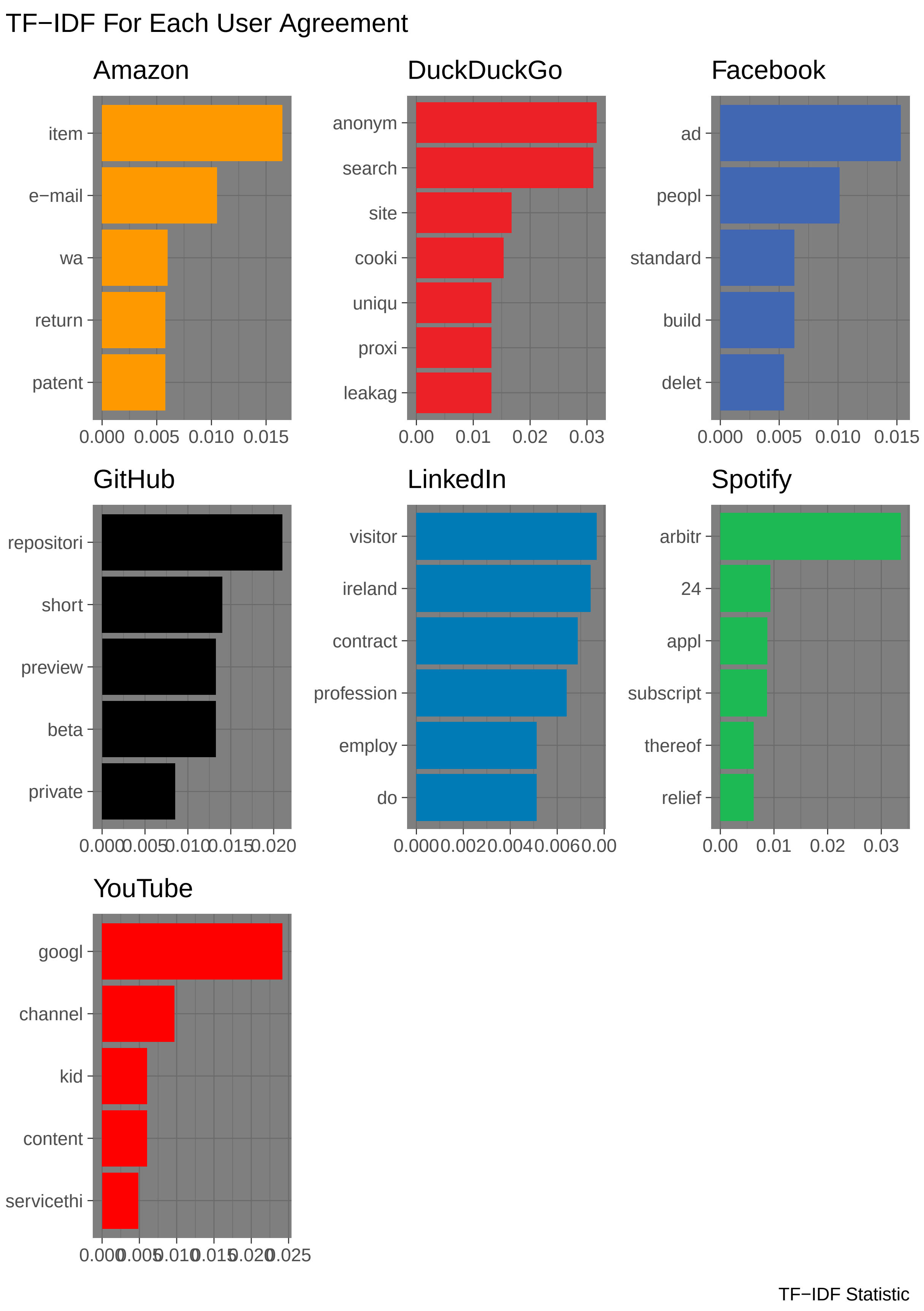
Onto the good stuff! Let's see which words are the most common:


Most Common Words Among All TOS Policies
Word
servic
content
term
agreement
includ
account
inform
user
provid
law
agre
right
product
parti
applic
access
person
search
polici
licens
time
legal
softwar
respons
notic
0
200
400
Number of Uses


Most Common Words in each User Agreement
Amazon
servic
content
softwar
includ
product
term
copyright
condit
inform
provid
right
law
properti
item
infring
0
20
40
60
DuckDuckGo
search
inform
site
engin
person
anonym
identifi
address
top
sett
cooki
term
store
privaci
data
0
20
40
60
Facebook
product
content
term
servic
peopl
ad
commun
data
share
provid
polici
account
permiss
right
delet
appli
advertis
0
20
40
60
GitHub
servic
content
account
term
user
agreement
inform
respons
repositori
licens
access
law
includ
right
provid
0
25
50
75
100
LinkedIn
servic
content
contract
agre
inform
includ
provid
law
account
right
term
polici
person
connect
post
data
consent
0
30
60
90
Spotify
servic
content
agreement
parti
applic
arbitr
user
agre
includ
law
subscript
right
term
claim
provid
paid
0
50
100
YouTube
servic
content
includ
agreement
term
provid
user
account
googl
legal
section
licens
law
access
termin
0
50
100
Number of uses

TF-IDF

A spectacular way to find differences between policies is by using the TF-IDF statistic.


TF−IDF For Each User Agreement
Amazon
item
e−mail
wa
return
patent
0.000 0.005 0.010 0.015
DuckDuckGo
anonym
search
site
cooki
uniqu
proxi
leakag
0.00 0.01 0.02 0.03
Facebook
ad
peopl
standard
build
delet
0.000 0.005 0.010 0.015
GitHub
repositori
short
preview
beta
private
0.000 0.005 0.010 0.015 0.020
LinkedIn
visitor
ireland
contract
profession
employ
do
0.000 0.002 0.004 0.006 0.00
Spotify
arbitr
24
appl
subscript
thereof
relief
0.00 0.01 0.02 0.03
YouTube
googl
channel
kid
content
servicethi
0.000 0.005 0.010 0.015 0.020 0.025
TF−IDF Statistic

We can see words like “item” are unique to Amazon, “privaci [privacy]” to DuckDuckGo, or “repositori [repository]” to GitHub. Features key to the services of these companies become clear after using the TF-IDF.

Word Embeddings

We can also compare words by measuring their similarity via the word2vec algorithm. We calculate the word probabilities and skipgram probabilities to find the normalized skipgram probability (PMI), find the singular value decomposition of the PMI matrix, and then search for similarity between words in the ‘eigen words’ matrix!

How do we do this? Word Math! Even though word2vec was not trained with a nearly large enough dataset to be trusted, we can still ask questions.

For example, what is the sum of “creepy” and “funded”?

```
## government
```

Yep.