Internet Archaeology: Scraping time series data from Archive.org

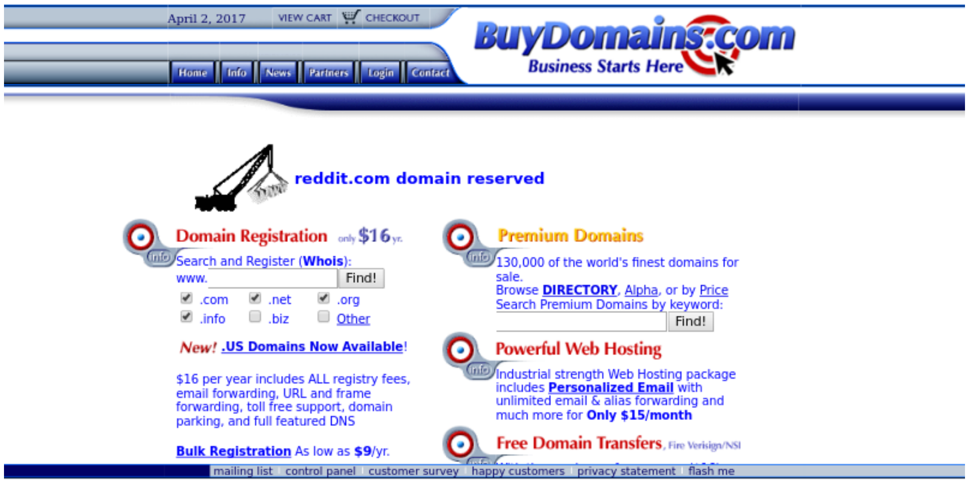
Wed, Apr 5, 2017

Skip to the Wayback Machine Scraper GitHub repo (https://github.com/sangaline/wayback-machine-scraper) if you're just looking for the completed command-line utility or the Scrapy middleware (https://github.com/sangaline/scrapy-wayback-machine). The article focuses on how the middleware was developed and an interesting use case: looking at time series data from Reddit.

Introduction

The Archive.org Wayback Machine (https://archive.org/web) is pretty awe inspiring. It's been archiving web pages since 1996 and has amassed *284 billion* page captures and over 15 petabytes of raw data. Many of these are sites that are no longer online and their content would have been otherwise lost to time. For sites that are still around, it can be absolutely fascinating to watch how they've evolved over the years.

Take Reddit (http://reddit.com) for instance. You can go back in time and watch it grow from this...



reddit.com domain reserved

Domain Registration only $16 yr.
Search and Register (Whois):
www. Find!
.com .net .org
.info .biz Other

New! .US Domains Now Available!

$16 per year includes ALL registry fees, email forwarding, URL and frame forwarding, toll free support, domain parking, and full featured DNS

Bulk Registration As low as $9/yr.

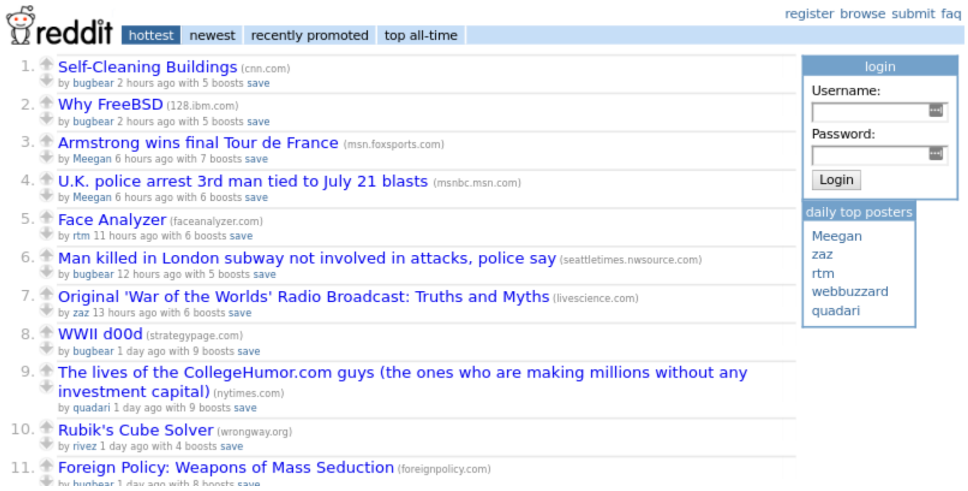
Premium Domains
130,000 of the world's finest domains for sale.
Browse DIRECTORY, Alpha, or by Price
Search Premium Domains by keyword:
Find!

Powerful Web Hosting
Industrial strength Web Hosting package includes Personalized Email with unlimited email & alias forwarding and much more for Only $15/month

Free Domain Transfers, Fire Verisign/NSI

mailing list control panel customer survey happy customers privacy statement flash me

to this...

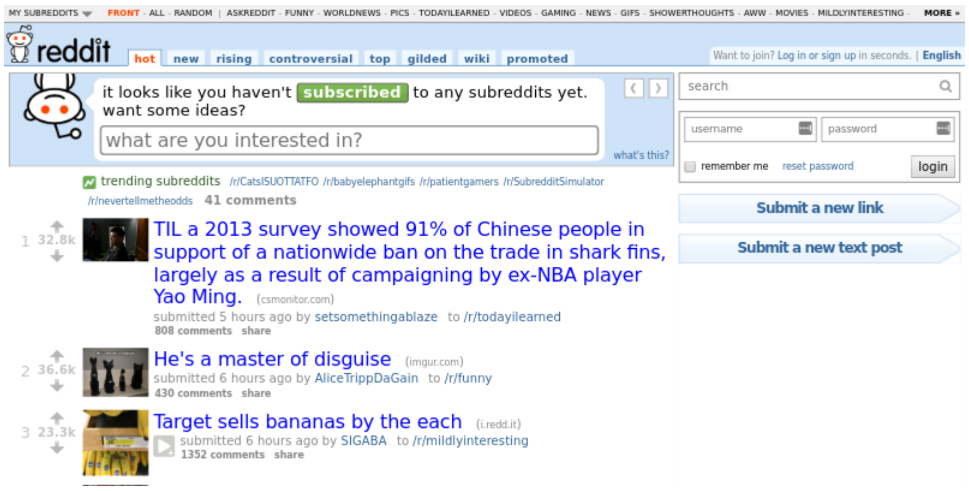


1. Self-Cleaning Buildings (cnn.com)
by bugbear 2 hours ago with 5 boosts save
2. Why FreeBSD (128.ibm.com)
by bugbear 2 hours ago with 5 boosts save
3. Armstrong wins final Tour de France (msn.foxsports.com)
by Meegan 6 hours ago with 7 boosts save
4. U.K. police arrest 3rd man tied to July 21 blasts (msnbc.msn.com)
by Meegan 6 hours ago with 6 boosts save
5. Face Analyzer (faceanalyzer.com)
by rtm 11 hours ago with 6 boosts save
6. Man killed in London subway not involved in attacks, police say (seattletimes.nwsource.com)
by bugbear 12 hours ago with 5 boosts save
7. Original 'War of the Worlds' Radio Broadcast: Truths and Myths (livescience.com)
by zaz 13 hours ago with 6 boosts save
8. WWII d00d (strategypage.com)
by bugbear 1 day ago with 9 boosts save
9. The lives of the CollegeHumor.com guys (the ones who are making millions without any investment capital) (nytimes.com)
by quadari 1 day ago with 9 boosts save
10. Rubik's Cube Solver (wrongway.org)
by rivez 1 day ago with 4 boosts save
11. Foreign Policy: Weapons of Mass Seduction (foreignpolicy.com)
by bugbear 1 day ago with 8 boosts save

login
Username:
Password:
Login

daily top posters
Meegan
zaz
rtm
webbuzzard
quadari

to this...


MY SUBREDDITS FRONT - ALL - RANDOM | ASKREDDIT - FUNNY - WORLDNEWS - PICS - TODAYILEARNED - VIDEOS - GAMING - NEWS - GIFS - SHOWERTHOUGHTS - AWW - MOVIES - MILDLYINTERESTING - MORE »
reddit
hot
new
rising
controversial
top
gilded
wiki
promoted
Want to join? Log in or sign up in seconds. | English

it looks like you haven't subscribed to any subreddits yet. want some ideas?
what are you interested in?
what's this?

search
username password
remember me reset password login
Submit a new link
Submit a new text post

trending subreddits /r/CatsISUOTTATFO /r/babyelephantgifs /r/patientgamers /r/SubredditSimulator /r/nevertellmetheodds 41 comments

1 32.8k TIL a 2013 survey showed 91% of Chinese people in support of a nationwide ban on the trade in shark fins, largely as a result of campaigning by ex-NBA player Yao Ming. (csmonitor.com)
submitted 5 hours ago by setsomethingablaze to /r/todayilearned
808 comments share

2 36.6k He's a master of disguise (imgur.com)
submitted 6 hours ago by AliceTrippDaGain to /r/funny
430 comments share

3 23.3k Target sells bananas by the each (i.redd.it)
submitted 6 hours ago by SIGABA to /r/mildlyinteresting
1352 comments share

with about *192 thousand* other stops along the way. That's just absolutely incredible to me (if you agree then please consider donating to help them keep doing what they do (https://archive.org/donate/)).

That's all well-and-good but the thing about 192 thousand web captures, let alone 284 billion, is that that's just way too much for one person to sift through by hand. There's a lot of interesting data there, but if you want to actually do something with it then you'll need some sort of scraper to collect the data from the Wayback Machine.

What might one call such a thing? Hmmm, I dunno. Maybe a...

WaybackMachineScraper

(https://github.com/sangaline/wayback-machine-scraper)

That's what I called my own reusable middleware and command-line utility (https://github.com/sangaline/wayback-machine-scraper) at least (original, right?).

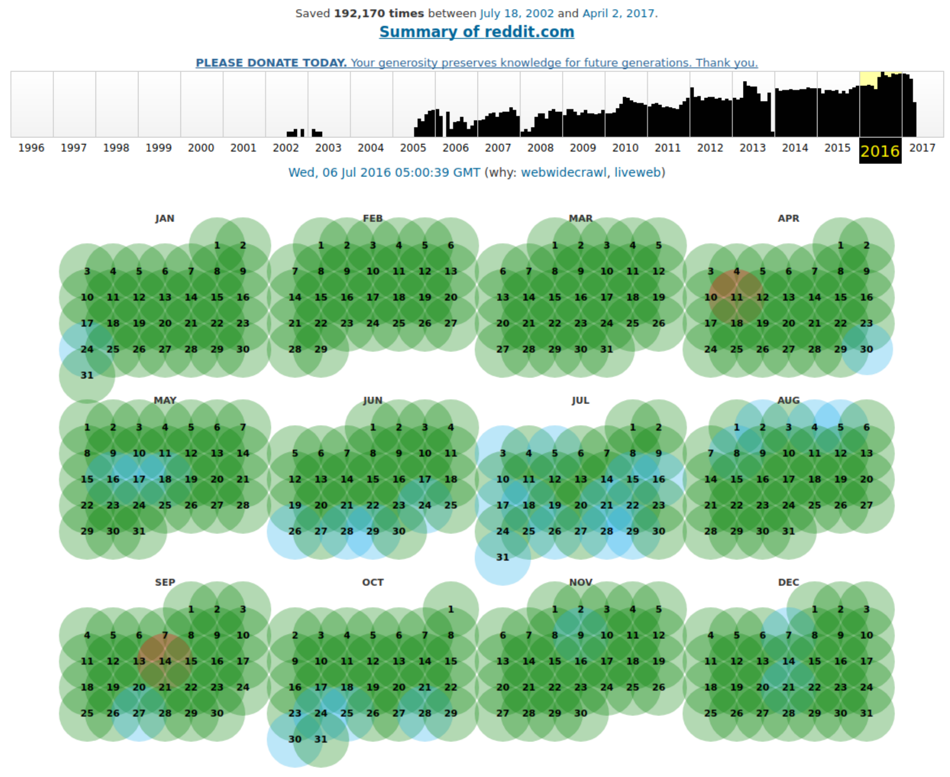
In this article, I'll walk you through the process of writing it using python and Scrapy (https://scrapy.org). I should mention that scraping archived pages from the Wayback Machine (https://archive.org/web) isn't exactly a new idea- the official Scrapy docs (https://doc.scrapy.org/en/latest/topics/practices.html) list scraping cached copies of pages under "Common Practices"- but I'm going to try to put a little bit of a twist on it.

If all you wanted to do was fetch a current or historical snapshot then you could just write something like

```
def waybackify_url(url, closest_timestamp='2017'):
    return f'https://web.archive.org/web/{closest_timestamp}/{url}'
```

and "waybackify" your URLs before crawling them. That's great if you just want to avoid rate limits and bans but it sure doesn't make for much of a web scraping tutorial! Aside from that, it doesn't really help much if you're interested in how a page changes over time. For example, I wanted to look at all of the available Hacker News (http://news.ycombinator.com) snapshots when I was writing the Reverse Engineering the Hacker News Ranking Algorithm (/post/reverse-engineering-the-hacker-news-ranking-algorithm/).

The most obvious solution in cases like this is to write a spider that crawls the archive index pages


Saved 192,170 times between July 18, 2002 and April 2, 2017.
Summary of reddit.com
PLEASE DONATE TODAY. Your generosity preserves knowledge for future generations. Thank you.
1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015 2016 2017
Wed, 06 Jul 2016 05:00:39 GMT (why: webwidecrawl, liveweb)
JAN FEB MAR APR MAY JUN JUL AUG SEP OCT NOV DEC

and extracts the timestamps from the URLs. That's definitely doable but it gets slightly more complicated than it seems at first and if you're putting this logic in your spider then chances are that reusing the code won't be trivial. Scraping time series data is a fairly general problem and so it would be nice if the code be reused with minimal modifications to spider code.

A more natural way to approach the problem is to write middleware that does the dirty work and integrates easily with existing code. I had basically already done this myself but hadn't quite cleaned it up enough that I would feel comfortable open sourcing it. After recently writing Advanced Web Scraping (/post/advanced-web-scraping-tutorial/) and receiving a lot of really positive feedback, I realized that others might find it informative if I did a little walkthrough of how I developed it and made the code available.

Just talking about middleware can get a little bit boring so I'll frame the discussion within the context of trying to analyze time series data from Reddit. The analysis there won't be particularly deep but there was something pretty interesting that popped out of the data. It should be more than enough of a starting point to do some internet archaeology of your own!

A Simple Reddit Spider

Let's begin by throwing together a very simple spider that just grabs the titles and scores of the stories on the front page of Reddit (http://reddit.com). We'll do that right after we get the boilerplate out of the way by setting up a virtualenv (https://virtualenv.pypa.io/en/stable/), installing Scrapy, and scaffolding out a default Scrapy project.

```
mkdir ~/scrapers/reddit
cd ~/scrapers/reddit
virtualenv env
. env/bin/activate
pip install scrapy
scrapy startproject reddit_scraper
cd reddit_scraper
```

If any of that stuff doesn't make sense to you then you might want to go check out The Scrapy Tutorial (https://doc.scrapy.org/en/latest/intro/tutorial.html) or The Advanced Web Scraping Tutorial (/post/advanced-web-scraping-tutorial/) (though you can probably follow along fine just knowing that that sets up a project scaffold).

Now we can add a basic spider to `reddit_scraper/spiders/reddit_spider.py`

```
from datetime import datetime as dt
import scrapy

class RedditSpider(scrapy.Spider):
    name = 'reddit'

    def start_requests(self):
        yield scrapy.Request('http://reddit.com')

    def parse(self, response):
        items = []
        for div in response.css('div.sitetable div.thing'):
            try:
                title = div.css('p.title a::text').extract_first()
                votes_div = div.css('div.score.unvoted')
                votes = votes_div.css('::attr(title)').extract_first()
                votes = votes or votes_div.css('::text').extract_first()

                items.append({'title': title, 'votes': int(votes)})
            except:
                pass

        if len(items) > 0:
            timestamp = response.meta['wayback_machine_time'].timestamp()
            return {'timestamp': timestamp, 'items': items}
```

This spider is about as simple as they come: it starts at `http://reddit.com` and doesn't crawl anywhere else from there. It uses a few CSS selectors to pull out the title and votes for each story on the front page and then attaches a timestamp to them. If we run the spider with

```
scrapy crawl reddit -o snapshots.jl
```

then it will produce an uglified version of something like

```
{
  "timestamp": 1491171571.881031,
  "items": [{
      "title": "Evidence that WSJ used FAKE screenshots",
      "votes": 32459
    }, {
      "title": "Expand the canvas? [MOD APPROVED]",
      "votes": 16305
    }, {
      "title": "My sister's cat spazzing out in his new cat tree!",
      "votes": 15815
    }, etc.
  ]
}
```

in `snapshots.jl`.

To track changes over time, we could now set up a cron job to run our scraper at regular intervals. That's a great way to collect frequently changing data but- much like in the movie Primer (www.imdb.com/title/tt0390384/)- you can't go back further than when you first turned it on. That's where the Wayback Machine (https://web.archive.org) comes in.

The Wayback Machine (https://web.archive.org) is basically a much more complicated spider that is saving the entire HTML content of each snapshot. If we can feed the historical HTML snapshots into our spider and attach the correct timestamps then it will effectively be as though we were running our scraper at those points in time.

The point that I'm trying to make here isn't the obvious one that we can use the HTML snapshots to extract the historical data. It's that if we connect the snapshots to our spider in the right way then the spider should be none the wiser and things should just work. This is in contrast to the other approach we discussed where our spider would need to be aware of the archive index pages, urls, etc.

Developing the Middleware

Writing a Scrapy Downloader Middleware (https://doc.scrapy.org/en/latest/topics/downloader-middleware.html) is generally where you'll end up whenever you need to intercept requests and responses to modify or replace them. Downloader middleware classes implement `process_request(request, spider)` and `process_response(request, response, spider)` methods that have a lot of freedom in what they can do. Let's start piecing together the middleware in `reddit_scraper/middlewares.py` and it should hopefully become clear exactly how much you can accomplish with this freedom.

We'll first add the basic initialization that loads our `WAYBACK_MACHINE_TIME_RANGE` setting and saves the crawler.

```
import json
from datetime import datetime as dt

from scrapy import Request
from scrapy.http import Response
from scrapy.exceptions import IgnoreRequest

class UnhandledIgnoreRequest(IgnoreRequest):
    pass

class WaybackMachine:
    cdx_url_template = ('http://web.archive.org/cdx/search/cdx?url={url}'
                        '&output=json&fl=timestamp,original,statuscode,digest')
    snapshot_url_template = 'http://web.archive.org/web/{timestamp}id_/{original}'

    def __init__(self, crawler):
        self.crawler = crawler

        # read the settings
        self.time_range = crawler.settings.get('WAYBACK_MACHINE_TIME_RANGE')

    @classmethod
    def from_crawler(cls, crawler):
        return cls(crawler)
```

There are a few other things in here but they aren't doing anything yet and we'll get to them shortly (so just ignore them for now). To actually turn this on, we'll have to also add a couple of settings to `reddit_scraper/settings.py`.

```
# enable the middleware
DOWNLOADER_MIDDLEWARES = {
    'reddit_scraper.middlewares.WaybackMachine': 50,
}

# only consider snapshots during the year of 2016
WAYBACK_MACHINE_TIME_RANGE = (20160101000000, 20170101000000)

# be bad but not too bad
ROBOTSTXT_OBEY = False
DOWNLOAD_DELAY = 5
```

Our middleware is now enabled but we need to implement request and response processing to actually make it useful. The request processing is the simpler of the two: we'll let any web.archive.org (https://web.archive.org) requests through without modification and for everything else we'll construct a request to the Wayback Machine's public CDX Server API (https://github.com/internetarchive/wayback/blob/master/wayback-cdx-server/README.md).

```
    def process_request(self, request, spider):
        # let any web.archive.org requests pass through
        if request.url.find('http://web.archive.org/') == 0:
            return

        # otherwise request a CDX listing of available snapshots
        return self.build_cdx_request(request)

    def build_cdx_request(self, request):
        cdx_url = self.cdx_url_template.format(url=request.url)
        cdx_request = Request(cdx_url)
        cdx_request.meta['original_request'] = request
        cdx_request.meta['wayback_machine_cdx_request'] = True
        return cdx_request
```

Returning a new request aborts the current request processing and sends the new request into the downloader pipeline. The new request passes through our `WaybackMachine` middleware unscathed this time because the URL starts with `http://web.archive.org/`. It (hopefully) makes it all the way CDX server which will provide what is basically the computer-friendly version of the archive index pages. The CDX server will specifically return a JSON file including the timestamp, URL, and statuscode of each snapshot request as well as a hash of the snapshot content. That will look something like

```
[["timestamp","original","statuscode","digest"],
["20020718215101", "http://reddit.com:80/", "200", "VNG6YBPFVMBWJPRETPQX45QEHDHXFOFD"],
["20020802023739", "http://reddit.com:80/", "200", "VNG6YBPFVMBWJPRETPQX45QEHDHXFOFD"],
["20020923101504", "http://reddit.com:80/", "200", "VNG6YBPFVMBWJPRETPQX45QEHDHXFOFD"],
etc.
]
```

and, like all responses, will make its way back through the downloader middleware. Our spider wouldn't know what to do with it though so we need to intercept the response and prevent it from making it that far. We can do this by implementing `process_response(request, response, spider)`.

```
    def process_response(self, request, response, spider):
        meta = request.meta

        # parse CDX requests and schedule future snapshot requests
        if meta.get('wayback_machine_cdx_request'):
            snapshot_requests = self.build_snapshot_requests(response, meta)

            # schedule all of the snapshots
            for snapshot_request in snapshot_requests:
                self.crawler.engine.schedule(snapshot_request, spider)

            # abort this request
            raise UnhandledIgnoreRequest

        # clean up snapshot responses
        if meta.get('original_request'):
            return response.replace(url=meta['original_request'].url)

        return response
```

We start out by grabbing the meta information on the request (you may have already noticed that we had attached some to our CDX request in `build_cdx_request(request)`). This meta information is then used to determine whether this is a response to a CDX request; if it is then we parse it to construct requests for the individual snapshots, schedule these with the Scrapy engine, and then abort the request by throwing an unhandled error that we defined earlier. The last little bit of code there is to make the response URL match that of the original request for the snapshot requests so that the spider doesn't have to know about where the snapshot responses actually came from.

The final piece of the puzzle is to implement the code for actually parsing the CDX responses and building the snapshot requests.

```
    def build_snapshot_requests(self, response, meta):
        # parse the CDX snapshot data
        data = json.loads(response.text)
        keys, rows = data[0], data[1:]
        def build_dict(row):
            new_dict = {}
            for i, key in enumerate(keys):
                new_dict[key] = row[i]
            return new_dict
        snapshots = list(map(build_dict, rows))

        # construct the requests
        snapshot_requests = []
        for snapshot in snapshots:
            # ignore snapshots outside of the time range
            if not (self.time_range[0] < int(snapshot['timestamp']) < self.time_range[1]):
                continue

            # update the url to point to the snapshot
            url = self.snapshot_url_template.format(**snapshot)
            original_request = meta['original_request']
            snapshot_request = original_request.replace(url=url)

            # attach extension specify metadata to the request
            snapshot_request.meta.update({
                'original_request': original_request,
                'wayback_machine_url': snapshot_request.url,
                'wayback_machine_time': dt.strptime(snapshot['timestamp'], '%Y%m%d%H%M%S'),
            })

            snapshot_requests.append(snapshot_request)

        return snapshot_requests
```

You can see here that our snapshot requests are built using `original_request` as a base. This means that they still have the original callbacks attached and are otherwise identical except for the extra meta data that we attach (and the temporarily different `url` property). Additionally, our `snapshot_url_template` uses a lesser known feature of the Wayback Machine API that allows us to get the original raw page content instead of the one with modified links and added content. After we switch `response.url` back to `original_request.url` in `process_response(request, response, spider)`, the response will only be distinguishable from one coming from the original server in that it has the additional `wayback_machine_url` and `wayback_machine_datetime` meta data attached. This will all come in handy when it comes time to integrate the middleware with our spider as we'll do momentarily.

Putting It All Together

We designed our middleware in such a way that pretty much any existing spider should "just work" and now we get to reap the benefits of that. No modifications of any generated requests are required and the only evidence that the responses were fetched from archive.org (https://archive.org) should be the additional meta data and some minor header differences. Indeed, we could run our scraper again now and it would successfully parse all of the available snapshots. The only problem is that the timestamps would be wrong because we're populating them with `datetime.datetime.now()`.

To remedy this, we simply replace

```
        return {
            'timestamp': dt.now().timestamp(),
            'items': items,
        }
```

with

```
        return {
            'timestamp': response.meta['wayback_machine_time'].timestamp(),
            'items': items,
        }
```

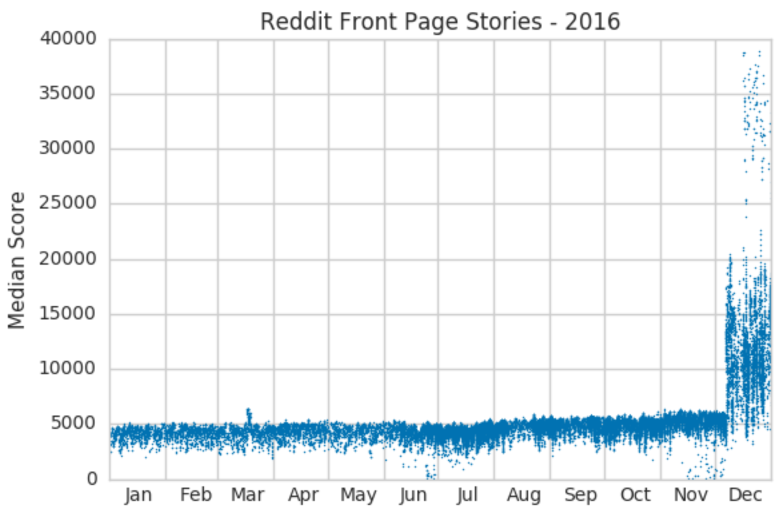
in `reddit_scraper/spiders/reddit_spider.py`. Running the crawler with `scrapy crawl reddit_scraper -o snapshots.jl` should now yield items for every front page snapshot from 2016!

We now have the data in a nice structured format and can finally get to the fun part. Let's start with loading the JSON Lines file back into python

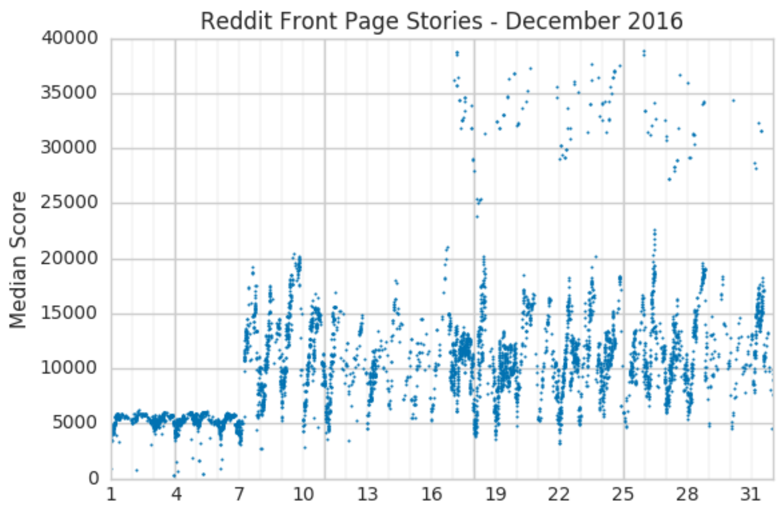
```
from datetime import datetime as dt
import json
import numpy as np

# load in the data
times, median_scores = [], []
with open('snapshots.jl', 'r') as f:
    for line in f:
        row = json.loads(line)
        scores= [item['votes'] for item in row['items']]
        time = dt.utcfromtimestamp(row['timestamp'])
        times.append(time)
        median_votes.append(np.median(scores))

# plot the data
fig = plt.figure(figsize=single_figsize)
ax = fig.add_subplot(1, 1, 1)
ax.plot(times, median_scores, 'o', ms=1)
```

We're computing the median scores here instead of the average scores in order to suppress the noise of unusually popular stories. The median score should be fairly representative of some convolution of Reddit site traffic and their scoring algorithm. Now let's take a look at how the median scores change over time by plotting them.

```
import matplotlib.pyplot as plt

# label the months
xticks = [dt(2016, i + 1, 18) for i in range(12)]
xticklabels = [date.strftime('%b') for date in xticks]
ax.set_xticklabels('')
ax.set_xticks(xticks, minor=True)
ax.set_xticklabels(xticklabels, minor=True)

# title and axis labels
ax.set_title('Reddit Front Page Stories - 2016')
ax.set_ylabel('Median Score')

# format and save it
fig.tight_layout()
fig.savefig('reddit-front-page-stories-2016.png')
```



Things are fairly stable with a small upwards trend until we hit December and the median scores go crazy! Let’s zoom in a little bit so we can see more clearly when this happened.



It looks like the median scores abruptly doubled on December 7th and the variations within each day also became more pronounced. It’s pretty clear that there was some algorithm change at Reddit that took place on that date. This was also a significant enough change that it would likely be noticed by regular users.

Let’s Google ‘Reddit vote scores “December 7th, 20016”’ and see who else noticed it. One of the first results is Reddit overhauls upvote algorithm to thwart cheaters and show the site’s true scale (http://www.entirenewslink.com/reddit-overhauls-upvote-algorithm-to-thwart-cheaters-and-show-the-sites-true-scale/) which links to a Reddit self-post by an admin named KeyserSosa (https://www.reddit.com/r/announcements/comments/5gvd6b/scores_on_posts_are_about_to_start_going_up from December 6th, 2016. Here is an abridged excerpt from that post.

> In the 11 years that Reddit has been around, we’ve accumulated a lot of rules (https://i.redd.it/oek2mm1io01y.png) in our vote tallying as a way to mitigate cheating and brigading on posts and comments Here’s a rough schematic of what the code looks like without revealing any trade secrets or compromising the integrity of the algorithm (https://i.redd.it/dastklohq01y.gif). Many of these rules are still quite useful, but there are a few whose primary impact has been to sometimes artificially deflate scores on the site (https://www.reddit.com/r/TheoryOfReddit/comments/z1sfn/one_minute_obamas_ama_karma_sco
>
> …Very soon (think hours, not days), we’re going to cut the scores over to be reflective of these new and updated tallies…
>
> **TL;DR** voting is confusing, we cleaned up some outdated rules on voting, and we’re updating the vote scores to be reflective of what they actually are. Scores are increasing by a lot.

And that’s exactly what we saw in the data. The data lets us even go a bit further and see that the new rules were relatively constant from the 7th up until the 16th or 17th when the median scores seem to fluctuate between the 5k-20k range and the 30-40k range. It’s hard to say exactly why this was happening from just the plots we’ve generated so far; maybe they were trying out new rule variations or maybe specific thresholds were being reached. If we were to dig in a little bit deeper and track individual story trajectories then we could probably make some more specific guesses (this is sadly outside the scope of this article however).

In addition to the scores, we also scraped the story titles. Let’s cycle through all of the stories again and pick out the most highly rated ones.

```
class TopStories:
    def __init__(self, N=10):
        self.stories = []
        self.N = N

    def add_story(self, new_story):
        # update any existing story with the higher score
        for story in self.stories:
            if story['title'] == new_story['title']:
                story['votes'] = max(story['votes'], new_story['votes'])
                return

        # insert a story in it's proper position
        for i in range(len(self.stories))[::-1]:
            if new_story['votes'] > self.stories[i]['votes']:
                if i == 0 or self.stories[i - 1]['votes'] > new_story['votes']:
                    self.stories.insert(i, new_story)
                    if len(self.stories) > self.N:
                        self.stories.pop()
                    return
            else:
                break

        # otherwise add it to the end of necessary
        if len(self.stories) < self.N:
                self.stories.append(new_story)

# load in the data
top_stories = TopStories()
with open('snapshots.jl', 'r') as f:
    for line in f:
        row = json.loads(line)
        time = dt.utcfromtimestamp(row['timestamp'])
        for item in row['items']:
            top_stories.add_story(item)
```

Score	Title
153759	1 dad reflex 2 children
109035	Hey Reddit, we need your help. We are small time youtubers who have recently discovered someone with 300x as many subscribers has made a near shot by shot rip off of one of our videos. The video has nearly 3x as many views as ours. Here is a side by side comparison. We don’t know what to do.
100304	TIL Carrie Fisher told her fans: “No matter how I go, I want it reported that I drowned in moonlight, strangled by my own bra.”
95621	Carrie Fisher Dies at 60
94915	Grindelwald, Switzerland
92277	Carrie Fisher dead at age 60
92192	Dog before and after being called a good boy
90257	Thanks Reddit. You saved me from potential credit card theft. Always wiggle the card reader.
88027	if you draw hands on the small McDonald’s hot cup it looks like a butt. If you poke a hole in it…
87462	Australian man waits 416 days to see what happens after his ipod timer passes 9999 hours 59 minutes and 59 seconds.

Well, now I kind of wish that we had scraped the URLs too so I could include the links. Here's 1 dad reflex 2 children (https://www.reddit.com/r/gifs/comments/5jrlw1/1_dad_reflex_2_children/) at least, it's pretty impressive.

There's a ton more that we could do here if we extracted a bit more data, but hopefully this is enough to give you a test for how easy it is to mess around with the data once we have it.

Wrap Up

Well, I hope that you enjoyed our little foray into internet archaeology here. There are countless possibilities for what you can do with time series data that you scrape from the Wayback Machine and what we've done here barely scrapes the surface. You could use the data that we scraped here to apply the techniques developed in Reverse Engineering the Hacker News Ranking Algorithm (/post/reverse-engineering-the-hacker-news-ranking-algorithm/) to Reddit or you could scrape other sites to find historical product prices, review scores, or anything else you're curious about or need to run your business.

I would love to here about whatever analysis you might undertake so feel free to reach out at evan@intoli.com (mailto:evan@intoli) (that's doubly true if you're looking for someone to help your business solve their data needs!). If you do plan to actually use the middleware that we developed then please check out the full code on the Scrapy Wayback Machine GitHub repo (https://github.com/sangaline/scrapy-wayback-machine). I skipped over some important error handling and edge cases to simplify the code a bit during this tutorial and you'll really want those in production. The repo contains those additions as well as a really useful command-line utility for scraping pages without custom parsing (which may be useful if you want to parse them in a language other than python).

Finally, please don't forget to donate to archive.org if you're scraping data from their servers (https://archive.org/donate/). They provide an awesome public server and scraping consumes a lot of their resouces. Throwing them a few bucks goes a long way in helping them provide the services that they do!

← Advanced Web Scraping: Bypassing "403 Forbidden," captchas, and more (http://sangaline.com/post/advanced-web-scraping-tutorial/)