Racking Mac Pros

HACKERMONTHLY

Issue 62 July 2015

Curator
Lim Cheng Soon

Contributors
Simon Kuhn
Miguel Cardona
Stanislaw Pitucha
Marek Majkowski
Mike Solomon
Artem Khurshudov
Radim Rehurek
Pete French

Proofreader
Emily Griffin

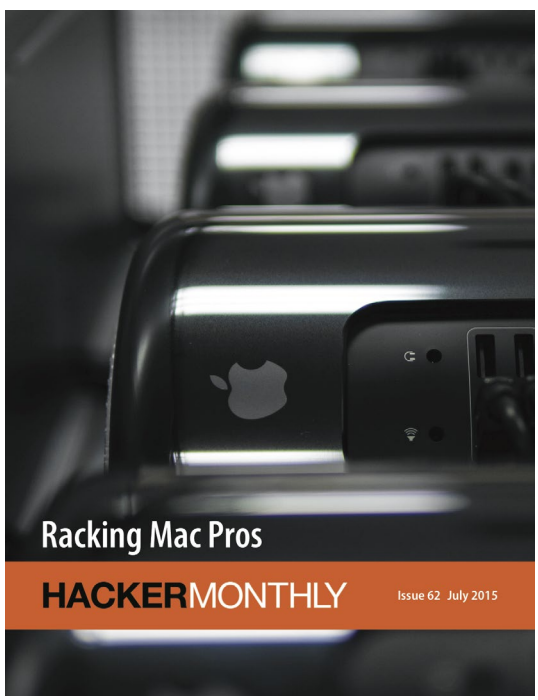
Printer
Blurb

HACKER MONTHLY is the print magazine version of Hacker News — *news.ycombinator.com*, a social news website wildly popular among programmers and startup founders. The submission guidelines state that content can be "anything that gratifies one's intellectual curiosity." Every month, we select from the top voted articles on Hacker News and print them in magazine format.
For more, visit *hackermonthly.com*

Advertising
ads@hackermonthly.com

Contact
contact@hackermonthly.com

Published by
Netizens Media
46, Taylor Road,
11600 Penang,
Malaysia.



Contents

For links to Hacker News dicussions, visit *hackermonthly.com/issue-62*

Racking Mac Pros

imgix's second generation image rendering system

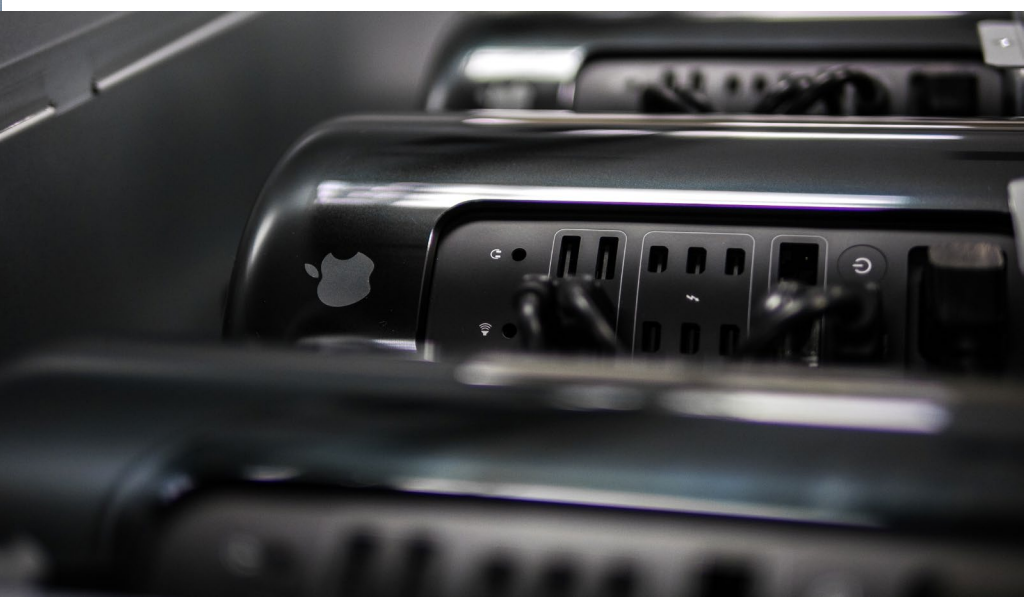
By SIMON KUHN & MIGUEL CARDONA

IMGIX [IMGIX.COM] IS AN image processing and delivery service that provides a supremely flexible, high performance, ultra–reliable solution to the problem of serving images on the modern internet. We operate our own hardware, run our own datacenters, and manage our own network infrastructure. At imgix's scale, maximizing efficiency and performance in image processing is critical for success. For this reason, we decided to incorporate Mac Pros in planning the build of our next generation image renderers. Because no existing Mac Pro server rack suited our needs, we designed and built our own.

Image Rendering At Scale

All of the images served by imgix pass through our image rendering servers. Parts of our technology are built using OS X's graphics frameworks, which offer high quality output and excellent performance. Our current rack and system design (R1) is built with Mac Minis, rack mounts from MK1 and integration by Racklive. [racklive.com]

R1 racks currently handle all of our production traffic. The design was appropriate for our needs — with excellent reliability, maintainability, and performance — but we're always looking for ways to improve upon it. In particular, we wanted greater power density and network port utilization, and we have reached the limits of the R1 design for both.



Power Density & Port Utilization

By improving these two metrics, we can minimize our fixed costs per rack, which allows us to more directly and efficiently scale based on actual customer requirements. For a redesign to be worthwhile, it needed to offer substantial improvements above R1 in these two areas. Our targets were aggressive: 70% of maximum power draw at peak and 80% network port utilization. Our R1 solution produces 37% draw and 66% port utilization.

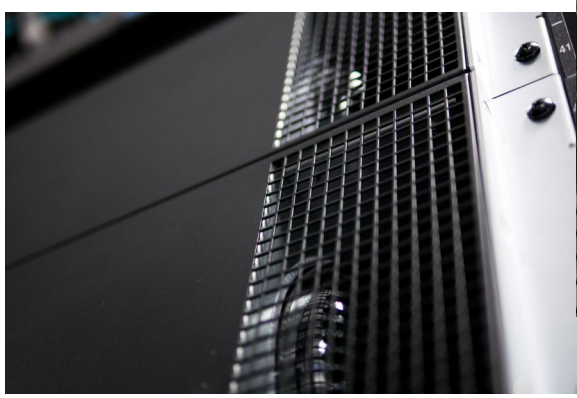
Building on OS X technologies means we're dependent on Apple hardware for this part of the service, but we aren't necessarily limited to Mac Minis. Apple's redesigned Mac Pro seemed like an ideal replacement, as long as we could reliably operate it in a datacenter environment. This was uncharted territory when we started this project, and even today is pretty uncommon in comparison to the Mac Mini.

Design

We considered various off–the–shelf products and spoke to manufacturers, but it became apparent that no one was designing for the type of usage we envisioned. Installing cylindrical systems in a 19–inch rack built for rectangles with high density and proper airflow is harder than it might seem. We had enough internal expertise to get started, but the project truly gained momentum when Racklive came onboard. They rose to the challenge, and the finished product is unlike any other Mac Pro solution.

Chassis

The R2 design consists of a metal chassis which houses four Mac Pros in a horizontal, sideways orientation with separate hot and cold air compartments. This chassis allows us to mount Mac Pros as we would any other server: on rails, in a rectangular enclosure, and with front and rear port access. The chassis itself is completely passive (although it could be adapted for fans in poorly ventilated sites). Each system within the chassis operates independently of the others.



Ports

Pigtail cables connect the ports on each system to ports on the outside of the chassis, so that there is no tangled mess lurking inside. Power and Ethernet are both routed to ports on the rear, with room for a second set of Ethernet cables if necessary. The rack design also incorporates room for a second network switch.

Each system has its own remotely controllable power outlet, which provides us with a basic level of out–of–band management. Graphics and USB may be routed to either the front or rear of the chassis. In our case, we chose the front for a better operator experience, because it can get pretty unpleasant standing in the hot aisle.

Air Flow

Positioning the Mac Pros sideways proved to be the key to obtaining the density we wanted, but that's all moot if the systems can't run reliably due to insufficient cooling. The inclusion of split hot and cold air chambers inside the chassis ensures that each system receives enough airflow to operate within acceptable ranges.

A single large channel provides cold air to all four systems, and air is forced through the hosts by their own fans as well as our datacenter's pressurized environment. The only way for air to pass through the rack is via the Mac Pro's fan, and the channel is sufficiently large to provide as much air as the four Mac Pros can intake.

Only a small portion of each system is positioned in the cool chamber, just enough for the air intakes to be exposed only to cold air. The rest of each Mac Pro is housed in the hot chamber, blocked off with a metal plate and fitted with gaskets to prevent air leakage.

Since the Mac Pro ventilates only from the top, and the hot chamber has a very large vent to the datacenter row's hot aisle, this allows hot air to quickly escape from the chassis.

Physical

The chassis is very heavy duty with excellent rigidity, since it must securely hold nearly 50 pounds of equipment. Each Mac Pro is clamped down to prevent unwanted movement. The chassis itself may be removed from the rack on sliding rails, although it isn't possible to remove just one Mac Pro — the entire group of four systems must be taken offline for maintenance.

Our inability to remove single systems is the main drawback to this design, but it's an acceptable tradeoff for density because of the way imgix's service is architected. Each chassis represents 9% of the rack's total capacity, well within our failure tolerances. The service is engineered to handle such a loss without noticeable impact.

Rack

The R2 design uses our standard 46U rack, common to all of imgix's deployments. 11 chassis fit into a rack, 4 Mac Pros per chassis, and our CDUs and network switch are mounted in the rear. The design incorporates our rapid deployment methodology: all of the systems, chassis and cables are integrated and assembled before they reach imgix's datacenter. Once a rack hits the datacenter floor, it can be processing images in as little as two hours.

We were able to develop this completely new design without making a single change to our existing datacenters because of our flexible and scalable datacenter architecture.

Once the first R2 rack has been put through its paces, we expect that racks of the R2 design will power imgix's image rendering for the next few years.

And what of our two target metrics? The R1 rack design was at 37% power draw and 66% network port utilization; the R2 rack design is currently at 81% power draw and 91% port utilization. By reducing our overhead costs, we can respond better and faster to service demand, and ultimately deliver a better product to our customers.

Conclusion

There are a number of different ways to deploy Mac Pros in a datacenter environment, but only imgix's design uniquely addresses the particular needs of a large scale service. We feel that the custom chassis and rack design represent an ideal intersection of flexibility, maintainability, and efficiency tailored to imgix's service needs. We're always looking to refine and improve our designs, and we're already hard at work on revisions to R2 as well as thinking about what the next big thing may involve. ■

Simon Kuhn has managed large scale Internet services at Yahoo and Dropbox; he runs imgix's worldwide datacenter, networking and systems infrastructure.

Miguel Cardona is imgix's lead designer, and was formerly a Visiting Professor at the School of Design at Rochester Institute of Technology.

imgix delivers dynamic images to your users with unparalleled flexibility, uncompromising quality and incredible speed.

Reprinted with permission of the original author.
First appeared in *hn.my/macpros* (imgix.com)

I Wrote a Website in Rust and Lived to Tell the Tale

By STANISŁAW PITUCHA

I WANTED TO CREATE a website for a personal project. This is usually a great opportunity to learn — no time pressure, no external requirements, etc. That meant I could choose the language I wanted to try out in advance (Rust) and take it for a spin. Here's a short summary of the experience.

The state of webdev in Rust

Rust environment has some support for web development, but it's still very basic. It's not a discovery — it's well known fact, even documented on "Are we web yet?" [arewewebyet.com] However unless you need a lot of pre-packaged components, you can already write some services.

I didn't have huge requirements: a read-only database-backed website with a 2-page admin panel. The website is not complete yet, but I've done most of the pages and now it just needs more typing, not thinking. So here's the usual: the good, the bad, and the ugly.

What worked well

The basics are there. I used the Iron framework [ironframework.io], which provides the server part with routers, static file handling, taking care of connections, etc. It's an ecosystem of its own. Jonathan Reem manages most of the GitHub bits with at least one contribution every day for the last year — impressive! Most of the useful Iron elements are in his repos and now it looks like HTTP2parser is on the way.

The router does its job. It supports getting values from the URL, and it can be composed (that is, both the `Router` and the `Chain` are `Handlers`). It doesn't support regex matching or casting parameters to the right type, but it's functional. Same goes for logging and static file handling. No thrills, they work.

There aren't that many template libraries to choose from yet, but handlebars-iron does the job.

The feeling that when the code compiles, it will not explode at runtime with some silly error is really, really good. Actually the compiler checks cover most of the things I'd normally unit-test, so this is probably the only non-trivial project I wrote without checks, and I'm OK with that. The only things I'd like to unit-test would probably get a package of their own and not stay in the webapp itself.

The code seems quite compact. There are some parts which are verbose and they're described later. But ~500 LOC include all initialization and config, DB entities and operations, around 7 routes, verifying arguments and passing them to templates. That's about as much as I'd expect from similar projects in Python.

What didn't work very well

API restrictions, lifetimes…

Some things are just harder in Rust. There were moments when I wanted to do basic refactoring, and the design of the libraries was simply against me. I realize that happens for a good reason typically, but there are also really odd cases.

One of those is described in an r2d2 issue — unfortunately it's not possible to return the connection from a function without either creating a new type, which will also

keep a ref-counted connection manager, or mut-borrowing the whole request. Of course the latter prevents getting other things from the request, like URL parameters. Issues like that suddenly throw a spanner in the works and leave you analyzing lifetimes, browsing docs, trying to figure out if you're wrong or if the library design really isn't compatible with what you're trying to do.

On the other hand, you end up learning a lot about lifetimes and borrows in practice.

Passing data in/out

Another bad part is Rust's JSON handling. It badly needs macros to make things easier. Using standard types results in things like:

```
let mut data = BTreeMap::new();
data.insert("events".to_string(), events.to_
json());
let page = Template::new("events_page", data);
```

When I really want it to be only something like:

```
let data = !json_obj { "events", events };
let page = Template::new("events_page", data);
```

Fortunately the `ToJson` trait handles creating more complex objects and `Vec<Event>` in this case could serialize itself.

Apparently there's maplit — while it's designed for hashmaps/btreemaps and `ToJson` can't use `&str` keys, unfortunately, it's still an improvement. Using those macros, the following works (and also gets rid of the only `mut` in the handlers — yay!):

```
let data = btreemap!{
    "events".to_string() => events.to_json(),
};
```

Compile times

Finally… the iteration time is just bad. It doesn't matter if you're writing some bigger piece of functionality, but when trying to solve some tricky compile error or just experimenting, waiting for more than 5 seconds is going to bother you at first and really irritate after the third try. A tiny project with lots of dependencies (516 LOC, 63 dependency crates) takes 13 seconds to compile — and that's in debug mode, without optimizations.

After a while I started to recognize which phase of compilation failed based on time-to-error (unknown names, wrong signatures, borrows, lifetimes, warnings) and that after 3 seconds or after any warning showed up, it's only LLVM/linker/optimizer running and there will be no error anymore.

What's just ugly

Verbose parts

Compared to many static languages, the handlers look tidy. Compared to dynamic languages, they're terrible. Starting with how to get a numeric ID out of the routed URL (unwraps are safe here — if they fail that's Router's implementation issue, not bad data):

```
let id_str = req.extensions.get::<Router>().
unwrap().find("id").unwrap();
let id = match id_str.parse::<i32>() {
    Err(_) => return not_found(),
    Ok(result) => result
};
```

And sure, this could be something like:

```
let id = try!(get_url_parameter::<i32>(&req,
"id"));
```

But unless you write that function, it isn't. Same goes for the database connection mentioned earlier, which could be a macro, but cannot be a function, or not easily anyway:

```
let pool = req.get::<Read<Database>>().ok().
expect("database component not initialized");
let connection = pool.get().unwrap();
```

These really needs to be more developer-friendly before people start using it daily.

Weird interfaces

Some interfaces need to be easier for developers before web development in Rust becomes more common. Figuring out a plugin architecture based on compile-time hashmap using types with associated values can be complicated. If your goal is just "get me the URL parameter," then it's needlessly annoying. It's great that it works like this under the covers, but I don't need to know about it.

Import avalanche

When working with many third party components, which is very common in webdev, the declarations on the top of the file can get rather long. For example the main file in my project contains just the initialization and route handlers (all database operations, entities, helper functions, etc., are in other modules), yet it still has 30 extern/mod/use lines at the top. With line breaks and comments that takes over one full screen. And that's when using deduplicated

```
use ...::{Something,Other,...}
```

matches on a single line.

Not the end of the world, but slightly annoying.

What's been observed

I don't know if these can be classified as good or bad, but they do give me a nice feeling.

Option

APIs usually handle `Option<...>` nicely. In Json, in database connectors, in templates, it just works where it should. That means there's rarely some special casing involved — if you have some related table in the database which may or may not have an entry you're interested in, it's probably going to be an `Option<Entry>` in your handler code.

That's good, because you won't see a ladder of special cases checking to see whether you have something or not. On the other hand, you need to learn to quickly write/read lines of `.and_then(),.err_map()` and others. While they were new to me, I quite like this approach actually. For example here:

```
let event_id = event_name
    .and_then({|name| Some(get_or_create_
event(&connection, &name)) })
    .and_then({|event| Some(event.event_id) });
```

Variable `event_id` will go directly to the template and I don't care if `event_name` existed, if it had a matching event, etc. Everything's going to be fine. Even the postgres connector can translate those to a `NULL` where needed.

This is very different from the guessing game of "does this function handle null/nil/None properly" in many other frameworks.

No ORM

There's no big ORM in Rust yet. There's r2d2 for connection pools, which is very welcome. There are also fairly standard database connectors. But that's about it. And actually, I don't mind that much. You can write macros for mini-ORM (just basic SELECT, INSERT) and handle everything else via SQL. Traits like From<> help a lot, because you can just implement

```
impl<'a> From<&'a Row<'a>> for Event {
```

for your types. Then reading them back from results is only:

```
rows.iter().map(|row| Event::from(&row)).
collect()
```

It's also really easy to make it generic or throw into a macro if it repeats too many times.

Exposing to public

Iron/Hyper are not yet ready to take internet traffic directly. Since Rust doesn't have a nonblocking IO available in a stable form yet, you should put the server behind something that can handle a slow-connection-DoS — for example Nginx.

Summary

It may look like I listed a lot more negatives than positives, but that's just because it's harder to talk about good things when they're expected. I enjoyed the experience and if some other personal project comes up, I think I'll use Rust again (instead of Python/Flask as usual).

If the project gets bigger, many things will have to be implemented — logging to external collectors, forwarding detailed errors, reporting processing/query times, application/schema migration control, etc. But that's still in the future. Today, it's a small, lean project, and Iron fulfills all the needs. ■

Stanislaw is a true generalist, having done jobs around small embedded chips, web dev, server deployment automation, internet telephony and more. Currently, he is working as an HP security engineer, exploring OpenStack. In the free time, he is a swing & blues dancer, traveling the world.

Reprinted with permission of the original author.
First appeared in *hn.my/rusttale* (viraptor.info)

How to Receive a Million Packets Per Second

By MAREK MAJKOWSKI

Photo: *flickr.com/photos/mccaffrey_uk/3208129302*

Last week during a casual conversation, I overheard a colleague saying: "The Linux network stack is slow! You can't expect it to do more than 50 thousand packets per second per core!"

That got me thinking. While I agree that 50kpps per core is probably the limit for any practical application, what is the Linux networking stack capable of? Let's rephrase that to make it more fun:

On Linux, how hard is it to write a program that receives 1 million UDP packets per second?

Hopefully, answering this question will be a good lesson about the design of a modern networking stack.

First, let us assume:

- Measuring packets per second (pps) is much more interesting than measuring bytes per second (Bps). You can achieve high Bps by better pipelining and sending longer packets. Improving pps is much harder.
- Since we're interested in pps, our experiments will use short UDP messages. To be precise: 32 bytes of UDP payload. That means 74 bytes on the Ethernet layer.
- For the experiments we will use two physical servers: "receiver" and "sender."
- They both have two six core 2GHz Xeon processors. With hyperthreading (HT) enabled that counts to 24 processors on each box. The boxes have a multi-queue 10G network card by Solarflare, with 11 receive queues configured. More on that later.
- The source code of the test programs is available here: `udpsender`, `udpreceiver`.

Prerequisites

Let's use port 4321 for our UDP packets. Before we start we must ensure the traffic won't be interfered with by the iptables:

```
receiver$ iptables -I INPUT 1 -p udp --dport
4321 -j ACCEPT
receiver$ iptables -t raw -I PREROUTING 1 -p udp
--dport 4321 -j NOTRACK
```

A couple of explicitly defined IP addresses will later become handy:

```
receiver$ for i in `seq 1 20`; do \
       ip addr add 192.168.254.$i/24 dev eth2; \
          done
sender$ ip addr add 192.168.254.30/24 dev eth3
```

1. The naive approach

To start let's do the simplest experiment. How many packets will be delivered for a naive send and receive?

The sender pseudo code:

```
fd = socket.socket(socket.AF_INET, socket.SOCK_
DGRAM)
fd.bind(("0.0.0.0", 65400)) # select source port
to reduce nondeterminism
fd.connect(("192.168.254.1", 4321))
while True:
    fd.sendmmsg(["\x00" * 32] * 1024)
```

While we could have used the usual `send` syscall, it wouldn't be efficient. Context switches to the kernel have a cost, and it is be better to avoid it. Fortunately a handy syscall was recently added to Linux: `sendmmsg`. It allows us to send many packets in one go. Let's do 1,024 packets at once.

The receiver pseudo code:

```
fd = socket.socket(socket.AF_INET, socket.SOCK_
DGRAM)
fd.bind(("0.0.0.0", 4321))
while True:
    packets = [None] * 1024
    fd.recvmmsg(packets, MSG_WAITFORONE)
```

Similarly, `recvmmsg` is a more efficient version of the common `recv` syscall.

Let's try it out:

```
sender$ ./udpsender 192.168.254.1:4321
receiver$ ./udpreceiver1 0.0.0.0:4321
  0.352M pps  10.730MiB /  90.010Mb
  0.284M pps   8.655MiB /  72.603Mb
  0.262M pps   7.991MiB /  67.033Mb
  0.199M pps   6.081MiB /  51.013Mb
  0.195M pps   5.956MiB /  49.966Mb
  0.199M pps   6.060MiB /  50.836Mb
  0.200M pps   6.097MiB /  51.147Mb
  0.197M pps   6.021MiB /  50.509Mb
```

With the naive approach we can do between 197k and 350k pps. Not too bad. Unfortunately there is quite a bit of variability. It is caused by the kernel shuffling our programs between cores. Pinning the processes to CPUs will help:

```
sender$ taskset -c 1 ./udpsender
192.168.254.1:4321
receiver$ taskset -c 1 ./udpreceiver1
0.0.0.0:4321
  0.362M pps  11.058MiB /  92.760Mb
  0.374M pps  11.411MiB /  95.723Mb
  0.369M pps  11.252MiB /  94.389Mb
  0.370M pps  11.289MiB /  94.696Mb
  0.365M pps  11.152MiB /  93.552Mb
  0.360M pps  10.971MiB /  92.033Mb
```

Now, the kernel scheduler keeps the processes on the defined CPUs. This improves processor cache locality and makes the numbers more consistent, just what we wanted.

2. Send more packets

While 370k pps is not bad for a naive program, it's still quite far from the goal of 1Mpps. To receive more, first we must send more packets. How about sending independently from two threads:

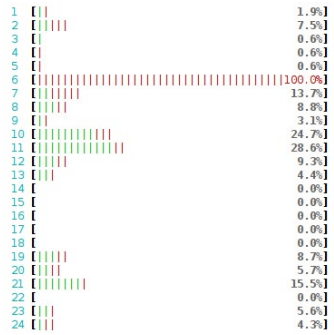
```
sender$ taskset -c 1,2 ./udpsender \
            192.168.254.1:4321
192.168.254.1:4321
receiver$ taskset -c 1 ./udpreceiver1
0.0.0.0:4321
  0.349M pps  10.651MiB /  89.343Mb
  0.354M pps  10.815MiB /  90.724Mb
  0.354M pps  10.806MiB /  90.646Mb
  0.354M pps  10.811MiB /  90.690Mb
```

The numbers on the receiving side didn't increase. `ethtool -S` will reveal where the packets actually went:

```
receiver$ watch 'sudo ethtool -S eth2 |grep rx'
     rx_nodesc_drop_cnt:    451.3k/s
     rx-0.rx_packets:       8.0/s
     rx-1.rx_packets:       0.0/s
     rx-2.rx_packets:       0.0/s
     rx-3.rx_packets:       0.5/s
     rx-4.rx_packets:     355.2k/s
     rx-5.rx_packets:       0.0/s
     rx-6.rx_packets:       0.0/s
     rx-7.rx_packets:       0.5/s
     rx-8.rx_packets:       0.0/s
     rx-9.rx_packets:       0.0/s
     rx-10.rx_packets:      0.0/s
```

Through these stats, the NIC reports that it had successfully delivered around 350kpps to RX queue number #4. The `rx_nodesc_drop_cnt` is a Solarflare specific counter saying the NIC failed to deliver 450kpps to the kernel.

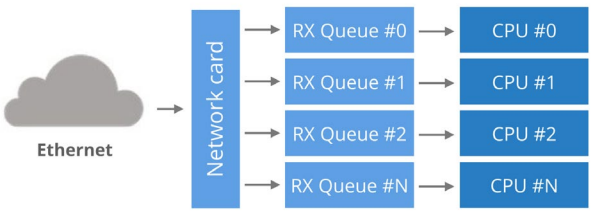
Sometimes it's not obvious why the packets weren't delivered. In our case though, it's very clear: the RX queue #4 delivers packets to CPU #4. And CPU #4 can't do any more work — it's totally busy just reading the 350kpps. Here's how that looks in `htop`:



Crash course to multi-queue NICs

Historically, network cards had a single RX queue that was used to pass packets between hardware and kernel. This design had an obvious limitation: it was impossible to deliver more packets than a single CPU could handle.

To utilize multicore systems, NICs began to support multiple RX queues. The design is simple: each RX queue is pinned to a separate CPU, therefore, by delivering packets to all the RX queues a NIC can utilize all CPUs. But it raises a question: given a packet, how does the NIC decide to which RX queue to push it?



Round-robin balancing is not acceptable, as it might introduce reordering of packets within a single connection. An alternative is to use a hash from packet to decide the RX queue number. The hash is usually counted from a tuple (src IP, dst IP, src port, dst port). This guarantees that packets for a single flow will always end up on exactly the same RX queue, and reordering of packets within a single flow can't happen.

In our case, the hash could have been used like this:

```
RX_queue_number = hash('192.168.254.30',
'192.168.254.1', 65400, 4321) % number_of_queues
```

Multi-queue hashing algorithms

The hash algorithm is configurable with `ethtool`. On our setup it is:

```
receiver$ ethtool -n eth2 rx-flow-hash udp4
UDP over IPV4 flows use these fields for computing
Hash flow key:
IP SA
IP DA
```

This reads as: for IPv4 UDP packets, the NIC will hash (src IP, dst IP) addresses. i.e.:

```
RX_queue_number = hash('192.168.254.30',
'192.168.254.1') % number_of_queues
```

This is pretty limited, as it ignores the port numbers. Many NICs allow customization of the hash. Again, using `ethtool` we can select the tuple (src IP, dst IP, src port, dst port) for hashing:

```
receiver$ ethtool -N eth2 rx-flow-hash udp4 sdfn
Cannot change RX network flow hashing options:
Operation not supported
```

Unfortunately our NIC doesn't support it. We are constrained to (src IP, dst IP) hashing.

A note on NUMA performance

So far all our packets flow to only one RX queue and hit only one CPU. Let's use this as an opportunity to benchmark the performance of different CPUs. In our setup the receiver host has two separate processor banks, each is a different NUMA node.

We can pin the single-threaded receiver to one of four interesting CPUs in our setup. The four options are:

1. Run receiver on another CPU, but on the same NUMA node as the RX queue. The performance as we saw above is around 360kpps.
2. With receiver on exactly same CPU as the RX queue we can get up to ~430kpps, but it creates high variability. The performance drops down to zero if the NIC is overwhelmed with packets.
3. When the receiver runs on the HT counterpart of the CPU handling RX queue, the performance is half the usual number at around 200kpps.
4. With receiver on a CPU on a different NUMA node than the RX queue we get ~330k pps. The numbers aren't too consistent, though.

While a 10% penalty for running on a different NUMA node may not sound too bad, the problem only gets worse with scale. On some tests I was able to squeeze out only 250kpps per core. On all the cross-NUMA tests the variability was bad. The performance penalty across NUMA nodes is even more visible at higher throughput. In one of the tests I got a 4x penalty when running the receiver on a bad NUMA node.

3. Multiple receive IPs

Since the hashing algorithm on our NIC is pretty limited, the only way to distribute the packets across RX queues is to use many IP addresses. Here's how to send packets to different destination IPs:

```
sender$ taskset -c 1,2 ./udpsender
192.168.254.1:4321 192.168.254.2:4321
```

`ethtool` confirms the packets go to distinct RX queues:

```
receiver$ watch 'sudo ethtool -S eth2 |grep rx'
     rx-0.rx_packets:     8.0/s
     rx-1.rx_packets:     0.0/s
     rx-2.rx_packets:     0.0/s
     rx-3.rx_packets:  355.2k/s
     rx-4.rx_packets:     0.5/s
     rx-5.rx_packets:  297.0k/s
     rx-6.rx_packets:     0.0/s
     rx-7.rx_packets:     0.5/s
     rx-8.rx_packets:     0.0/s
     rx-9.rx_packets:     0.0/s
     rx-10.rx_packets:    0.0/s
```

The receiving part:

```
receiver$ taskset -c 1 ./udpreceiver1
0.0.0.0:4321
  0.609M pps  18.599MiB / 156.019Mb
  0.657M pps  20.039MiB / 168.102Mb
  0.649M pps  19.803MiB / 166.120Mb
```

Hurray! With two cores busy with handling RX queues, and third running the application, it's possible to get ~650k pps!

We can increase this number further by sending traffic to three or four RX queues, but soon the application will hit another limit. This time the `rx_nodesc_drop_cnt` is not growing, but the `netstat` "receiver errors" are:

```
receiver$ watch 'netstat -s --udp'
Udp:
      437.0k/s packets received
        0.0/s packets to unknown port received.
      386.9k/s packet receive errors
        0.0/s packets sent
    RcvbufErrors:  123.8k/s
    SndbufErrors: 0
    InCsumErrors: 0
```

This means that while the NIC is able to deliver the packets to the kernel, the kernel is not able to deliver the packets to the application. In our case it is able to deliver only 440kpps, the remaining 390kpps + 123kpps are dropped due to the application not receiving them fast enough.

4. Receive from many threads

We need to scale out the receiver application. The naive approach, to receive from many threads, won't work well:

```
sender$ taskset -c 1,2 ./udpsender
192.168.254.1:4321 192.168.254.2:4321
receiver$ taskset -c 1,2 ./udpreceiver1
0.0.0.0:4321 2
  0.495M pps  15.108MiB / 126.733Mb
  0.480M pps  14.636MiB / 122.775Mb
  0.461M pps  14.071MiB / 118.038Mb
  0.486M pps  14.820MiB / 124.322Mb
```

The receiving performance is down compared to a single threaded program. That's caused by a lock contention on the UDP receive buffer side. Since both threads are using the same socket descriptor, they spend a disproportionate amount of time fighting for a lock around the UDP receive buffer.

Using many threads to receive from a single descriptor is not optimal.

5. SO_REUSEPORT

Fortunately, there is a workaround recently added to Linux: the `SO_REUSEPORT` flag. When this flag is set on a socket descriptor, Linux will allow many processes to bind to the same port. In fact, any number of processes will be allowed to bind and the load will be spread across them.

With `SO_REUSEPORT` each of the processes will have a separate socket descriptor. Therefore each will own a dedicated UDP receive buffer. This avoids the contention issues previously encountered:

```
receiver$ taskset -c 1,2,3,4 ./udpreceiver1
0.0.0.0:4321 4 1
  1.114M pps  34.007MiB / 285.271Mb
  1.147M pps  34.990MiB / 293.518Mb
  1.126M pps  34.374MiB / 288.354Mb
```

This is more like it! The throughput is decent now!

More investigation will reveal further room for improvement. Even though we started four receiving threads, the load is not being spread evenly across them:

```
10788 marek    S  0.0  0.0  0:00.00  |              `- ./udpreceiver1 0.0.0.0:4321 4 1
10792 marek    S  0.0  0.0  0:00.00  |                 `- ./udpreceiver1 0.0.0.0:4321 4 1
10791 marek    R 92.0  0.0  0:55.23  |                 `- ./udpreceiver1 0.0.0.0:4321 4 1
10790 marek    R 92.0  0.0  0:56.66  |                 `- ./udpreceiver1 0.0.0.0:4321 4 1
10789 marek    S  0.0  0.0  0:00.00  |                 `- ./udpreceiver1 0.0.0.0:4321 4 1
```

Two threads received all the work and the other two got no packets at all. This is caused by a hashing collision, but this time it is at the `SO_REUSEPORT` layer.

Final words

I've done some further tests, and with perfectly aligned RX queues and receiver threads on a single NUMA node it was possible to get 1.4Mpps. Running receiver on a different NUMA node caused the numbers to drop achieving at best 1Mpps.

To sum up, if you want a perfect performance you need to:

- Ensure traffic is distributed evenly across many RX queues and `SO_REUSEPORT` processes. In practice, the load usually is well distributed as long as there are a large number of connections (or flows).
- You need to have enough spare CPU capacity to actually pick up the packets from the kernel.
- To make the things harder, both RX queues and receiver processes should be on a single NUMA node.

While we had shown that it is technically possible to receive 1Mpps on a Linux machine, the application was not doing any actual processing of received packets — it didn't even look at the content of the traffic. Don't expect performance like that for any practical application without a lot more work. ■

After fruitful encounters with such diverse topics as high performance key value databases, distributed queueing systems, making real time web communication enjoyable and accelerating the time so that testing servers and protocols takes seconds, Marek finally settled for working on DDoS mitigation in CloudFlare London office, where he appreciates most the parking space for his motorbike.

Reprinted with permission of the original author.
First appeared in *hn.my/mpackets* (cloudflare.com)

How I Doubled my Internet Speed with OpenWRT

By MIKE SOLOMON

OpenWRT [openwrt.org] is a powerful Linux distribution for embedded devices, such as my router, and this is the story of how I used it to double my bandwidth at no extra cost to myself.

How? By doubling the number of Internet connections I have.

My Setup

My Internet

My internet is through Comcast (unfortunately).

Comcast has an initiative called Xfinity WiFi. When you rent a cable modem/router combo from Comcast (as one of my nearby neighbors apparently does), in addition to broadcasting your own WiFi network, it is kind enough to also broadcast "xfinitywifi," a second "hotspot" network metered separately from your own.

This hotspot allows Comcast customers to connect with their credentials.

My Router

My router is a Buffalo WZR-HP-AG300H. Crucially, this router 1) supports OpenWRT and 2) has two independent radios. I use one of them for my home WiFi network.

My Idea

By now, you've probably put two and two together.

I use my router's extra radio to connect to the xfinitywifi hotspot, then load balance my outbound traffic across the connection I pay for and the bonus xfinitywifi connection.

Obviously this is a pretty specific scenario, but if you have:

1. A hotspot you have credentials for within range
2. A router that supports both OpenWRT
3. That same router has a spare radio

How to set this up

1. Install OpenWRT

Find your router on OpenWRT's table of hardware and follow the instructions to install it, [hn.my/toh] getting your WiFi and network set up as usual.

2. Install multi-wan software in OpenWRT

Open your router's web interface and navigate to `/cgi-bin/luci/admin/system/packages` and install `luci-app-mwan3`. This (along with its dependencies) allows you to support multiple internet connections with round-robin load balancing between them (with connection pinning for HTTPS).

3. Authenticate a MAC address with xfinitywifi

The xfinitywifi hotspot requires authentication, not via WPA2 or other normal network security, but with a Comcast login. It remembers this login by way of your MAC address. Unfortunately, it is not very easy to authenticate directly through the router, so instead we will authenticate a MAC address through a computer, then switch

the apparent MAC address the router uses.

1. Generate a fake MAC address. Here's one: `02:67:1c:16:1f:21`
2. Spoof your MAC address (for your wireless adapter) on your computer. Be sure to find out how to do it on your Linux/Mac/Windows system. Remember to record your old MAC address.
3. With your MAC address spoofed, connect to xfinitywifi and enter your Comcast credentials

Disconnect from xfinitywifi and restore your original MAC address

4. Connect the router to xfinitywifi

In your OpenWRT web (LuCI) interface at `/cgi-bin/luci/admin/network/wireless`, press Scan on your available radio, and select Join Network for `xfinitywifi`. Name it `wan2` and add it to the `wan` firewall group. Save & Apply your settings.

Now, go to `/cgi-bin/luci/admin/network/network/wan2` and go to the Advanced Settings tab. Paste your fake and authenticated MAC address into the "Override MAC address" field. Save & Apply your settings.

5. Prepare mwan3 for a wireless WAN

In your OpenWRT web (LuCI) interface at `cgi-bin/luci/admin/network/network/wan/`, click the Advanced Settings tab and enter 10 under Use gateway metric and Save your settings.

At `cgi-bin/luci/admin/network/network/wan2/`, click the Advanced Settings tab and enter 20 under Use gateway metric and Save your settings.

In your OpenWRT web (LuCI) interface at `/cgi-bin/luci/admin/network/mwan/advanced/networkconfig`, you will see your network config file. Paste this section at the bottom, adjusting as necessary with settings from your xfinitywifi connection:

```
config route 'default_wan2'
  option interface 'wan2'
  option target '0.0.0.0'
  option netmask '0.0.0.0'
  option gateway '192.168.1.1'
  option metric '20'
```

Normally this last step is not necessary, but for some reason mwan3 seems to need it to work with wireless networks.

Submit your changes.

Check it!

Go to `cgi-bin/luci/admin/network/mwan` and you should see both networks green!

At least you will if you're the luckiest person ever. More likely you'll run into problems, check out the mwan docs [hn.my/mwan] and Google around.

Another good test is to go to What is my IP [whatismyip.com] and refresh several times and ensure you see two different IP addresses.

Good luck! ■

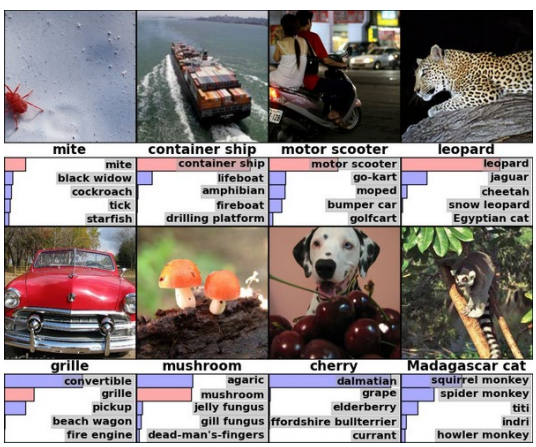
Mike Solomon is a software engineer in San Francisco. He works at Twitter (@msol) on distributed systems and backend Scala services. He writes sometimes at *msol.io*

Reprinted with permission of the original author.
First appeared in *hn.my/speed2x* (msol.io)

Suddenly, a Leopard Print Sofa Appears

By ARTEM KHURSHUDOV

If you have been around all the machine learning and artificial intelligence stuff, you surely have already seen this:



Or, if you haven't, there are some deep convolutional network result samples from ILSVRC2010, by Hinton and Krizhevsky

Let's look for a moment at the top-right picture. There's a leopard, recognized with substantial confidence, and then two much less probable choices are jaguar and cheetah.

And this is, if you think about it for a bit, kinda cool. Do you know how to tell apart those three big and spotty kitties? Because I totally don't. There must be differences, of course — maybe something subtle and specific that only a skilled zoologist can perceive, like general body shape or jaw size, or tail length — or maybe is it context/background, because leopards inhabit forests and are more likely to be found lying on a tree, when cheetahs live in savanna? Either way, for a machine learning algorithm, this looks very impressive to me. So, is that the famous deep learning approach? Are we going to meet human-like machine intelligence soon?

Well...turns out, maybe not so fast.

Just a little zoological fact

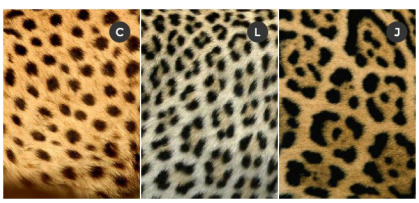
Let's take a closer look at these three kinds of big cats again. Here's the jaguar, for example:

It's the biggest cat on both Americas, which also has a curious habit of killing its prey by puncturing their skull and brain (that's not really the little fact we're looking for). It's the most massive cat in comparison with leopard and cheetah, and its other distinguishing features are dark eyes and larger jaw. Well, that actually looks pretty fine-grained.

Then, the leopard. It's a bit smaller than the jaguar and generally more elegant, considering, for example, its smaller paws and jaw. And also yellow eyes. Cute.

And the smallest of the pack, the cheetah, that actually looks quite different from the previous two. Has a generally smaller, long and slim body, and a distinctive face pattern that looks like two black tear trails running from the corners of its eyes.

And now for the part I've purposely left out: black spotty print pattern. It's not completely random, as you might think it is — rather, black spots are combined into small groups called "rosettes." You can see that jaguar rosettes are large, distinctive, and contain a small black spot inside, while leopard rosettes are significantly smaller. As for the cheetah, its print doesn't contain any, just a scatter of pure black spots.



See how those three prints actually differ.

Suspicion grows

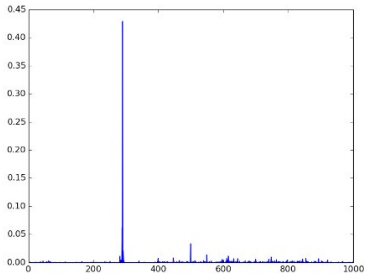
Now, I have a little bit of a bad feeling about it. What if this is the only thing our algorithm does — just treating these three pictures like shapeless pieces of texture, knowing nothing about leopard's jaw or paws, its body structure at all? Let's test this hypothesis by running a pre-trained convolutional network on a very simple test image. We're not trying to apply any visual noise, artificial occlusion or any other tricks to mess with image recognition — that's just a simple image, which I'm sure everyone who reads this page will recognize instantly.

Here it is:

We're going to use Caffe [caffe.berkeleyvision.org] and its pre-trained CaffeNet model, which is actually different from Hinton and Krizhevsky's AlexNet, but the principle is the same, so it will do just fine. And here we go:

```
import numpy as np
import matplotlib.pyplot as plt

caffe_root = '../'
import sys
sys.path.insert(0, caffe_root + 'python')

import caffe

MODEL_FILE = '../models/bvlc_reference_caffenet/deploy.prototxt'
PRETRAINED = '../models/bvlc_reference_caffenet/bvlc_reference_
caffenet.caffemodel'
IMAGE_FILE = '../sofa.jpg'

caffe.set_mode_cpu()
net = caffe.Classifier(MODEL_FILE, PRETRAINED,
                       mean=np.load(caffe_root + 'python/caffe/
imagenet/ilsvrc_2012_mean.npy').mean(1).mean(1),
                       channel_swap=(2, 1, 0),
                       raw_scale=255,
                       image_dims=(500, 500))
input_image = caffe.io.load_image(IMAGE_FILE)
prediction = net.predict([input_image])
plt.plot(prediction[0])
print 'predicted class:', prediction[0].argmax()
plt.show()
```

Here's the result:



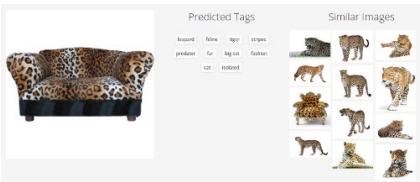

```
>> predicted class: 290
```

```
n02128385 leopard, Panthera pardus
n02128757 snow leopard, ounce, Panthera uncia
n02128925 jaguar, panther, Panthera onca, Felis onca
n02129165 lion, king of beasts, Panthera leo
n02129604 tiger, Panthera tigris
```

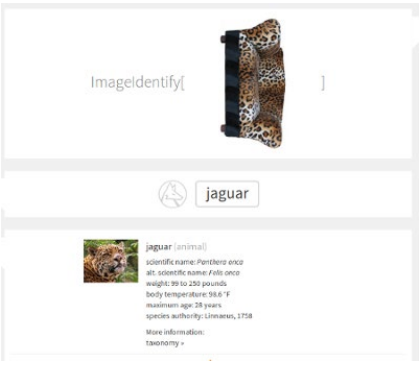
Whoops.

But wait, maybe that's just CaffeNet thing? Let's check something third-party:

- Clarifai [clarifai.com] (those guys did great on the latest ImageNet challenge)



- Brand new Stephen Wolfram's ImageIdentify [imageidentify.com]



Okay, I cheated a bit: on the last picture the sofa is rotated by 90 degrees, but that's really simple transformation that should not change the recognition output so radically. I've also tried Microsoft and Google services and nothing has beaten rotated leopard print sofa. Interesting result, considering all the "*{Somebody}'s Deep Learning Project Outperforms Humans In Image Recognition*" headlines that's been around for a while now.

Why is this happening?

Now, here's a guess. Imagine a simple supervised classifier, without going into model specifics, that accepts a bunch of labeled images and tries to extract some inner structure (a set of features) from that dataset to use for recognition. During the learning process, a classifier adjusts its parameters using prediction/recognition error, and here's when dataset size and structure matter. For example, if a dataset contains 99 leopards and only one sofa, the simplest rule that tells a classifier to always output "leopard" will result in 1% recognition error while staying not intelligent at all.

And that seems to be exactly the case, both for our own visual experience and for ImageNet dataset. Leopard sofas are rare things. There simply aren't enough of them to make difference for a classifier; and black spot texture makes a very distinctive pattern that is otherwise specific to a leopard category. Moreover, being faced with different classes of big spotted cats, a classifier can benefit from using these texture patterns, since they provide simple distinguishing features (compared with the others like the size of the jaw). So, our algorithm works just like it's supposed to. Different spots make different features, there's little confusion with other categories and the sofa example is just an anomaly. Adding enough sofas to the dataset will surely help (and then the size of the jaw will matter more, I guess), so there's no problem at all, it's just how learning works.

Or is it?

What we humans do

Remember your first school year, when you learned digits in your math class.

When each student was given a heavy book of MNIST database, hundreds of pages filled with endless hand-written digit series, 60000 total, written in different styles, bold or italic, distinctly or sketchy. The best students were also given an appendix, "Permutation MNIST," that contained the same digits, but transformed in lots of different ways: rotated, scaled up and down, mirrored and skewed. And you had to scan through all of them to pass a math test, where you had to recognize just a small subset of length 10000. And just when you thought the nightmare was over, a language class began, featuring not ten recognition categories, but twenty-six instead.

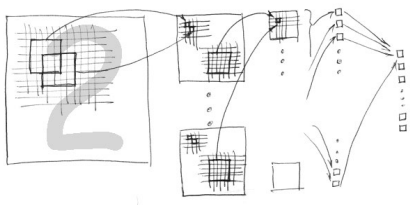
So, are you going to say that was not the case?

It's an interesting thing: looks like we don't really need a huge dataset to learn something new. We perceive digits as abstract concepts, Plato's ideal forms, or actually rather a spatial combinations of ones, like "a straight line," "a circle," "an angle." If an image contains two small circles placed one above the other, we recognize an eight; but when none of the digit-specific elements are present, we consider the image to be not a digit at all. This is something a supervised classifier never does — instead, it tries to put the image into the closest category, even if likeness is negligible.

Maybe MNIST digits is not a good example — after all, we all

have seen a lot of them in school, maybe enough for a huge dataset. Let's get back to our leopard print sofa. Have you seen a lot of leopards in your life? Maybe, but I'm almost sure that you've seen "faces" or "computers" or "hands" a lot more often. Have you actually seen such a sofa before — even once? Can't be one hundred percent confident for myself, but I think I have not. And nevertheless, despite this total lack of visual experience, I don't consider the image above a spotty cat in a slightest bit.

Convolutional networks make it worse



Deep convolutional network are long-time ImageNet champions. No wonder; they are designed to process images, after all. If you are not familiar with the concept of CNNs, here's a quick reminder: they are locally-connected networks that use a set of small filters as local feature detectors, convolving them across the entire image, which makes these features translation-invariant (which is often a desired property). This is also a lot cheaper than trying to put an entire image (represented by 1024x768=~800000 naive pixel features) into a fully-connected network. There are other operations involved in CNNs feed-forward propagation step, such as subsampling or pooling, but let's focus on convolution step for now.

Leopards (or jaguars) are complex 3-dimensional shapes with quite a lot of degrees of freedom (considering all the body parts that can move independently). These shapes can produce a lot of different 2d contours projected on the camera sensor: sometimes you can see a distinct silhouette featuring a face and full set of paws, and sometimes it's just a back and a curled tail. Such complex objects can be handled by a CNN very efficiently by using a simple rule: "take all these little spotty-pattern features and collect as many matches as possible from the entire image." CNNs local filters ignore the problem of having different 2d shapes by not trying to analyze leopard's spatial structure at all — they just look for black spots, and, thanks to nature, there are a lot of them in any leopard picture. The good thing here is that we don't have to care about object's pose and orientation, and the bad thing is that, well, we are now vulnerable to some specific kinds of sofas.

And this is really not good. CNN's usage of local features allows for transformation invariance, but this comes with the price of not knowing the object's structure nor its orientation. CNN cannot distinguish between a cat sitting on the floor and a cat sitting on the ceiling upside down, which might be good for Google image search but for any other application involving interactions with actual cats, it's not.

If that doesn't look convincing, take a look at Hinton's paper from 2011 [hn.my/hinton] where he says that convolutional networks are doomed precisely because of the same reason. The rest of the paper is about an alternative approach, his capsule theory [hn.my/capsule], which is definitely worth reading, too.

We're doing it wrong

Maybe not all wrong, and of course. Convolutional networks are extremely useful things, but think about it: sometimes it almost looks like we're already there. We're using huge datasets like ImageNet, organize competitions and challenges, where we, for example, have decreased MNIST recognition error rate from 0.87 to 0.23 (in three years) — considering that no one really knows what error rate a human brain can achieve. There's a lot of talk about GPU implementations — like it's just a matter of computational power now, and the theory is all fine. It's not. And the problem won't be solved by collecting even larger datasets and using more GPUs, because leopard print sofas are inevitable. There's always going to be an anomaly; lots of them, actually, considering all the things painted in different patterns. Something has to change. Good recognition algorithms have to understand the structure of the image and to be able to find its elements like paws or face or tail, despite the issues of projection and occlusion.

So I guess, there's still a lot of work to be done. ■

Artem is a Python developer and a Ph.D. student (machine learning and computer vision) from Krasnodar, Russia. He is currently working on currently working on *AirTribune.com*

Reprinted with permission of the original author.
First appeared in *hn.my/leopard* (rocknrollnerd.github.io)

Practical Data Science in Python

By RADIM REHUREK

The goal of this article is to demonstrate some high level, introductory concepts behind (text) machine learning. The concepts are accompanied by concrete code examples in this notebook, which you can run yourself (after installing IPython, see below), on your own computer.

The code examples build a working, executable prototype: an app to classify phone SMS messages in English (well, the "SMS kind" of English...) as either "spam" or "ham" (=not spam).

The language used throughout will be Python, a general purpose language helpful in all parts of the pipeline: I/O, data wrangling and preprocessing, model training, and evaluation. While Python is by no means the only choice, it offers a unique combination of flexibility, ease of development, and performance, thanks to its mature scientific computing ecosystem. Its vast, open source ecosystem also avoids the lock-in (and associated bitrot) of any single specific framework or library.

End-to-end example: automated spam filtering

```
In [1]:
%matplotlib inline
import matplotlib.pyplot as plt
import csv
from textblob import TextBlob
import pandas
import sklearn
import cPickle
import numpy as np
from sklearn.feature_extraction.text import
CountVectorizer, TfidfTransformer
from sklearn.naive_bayes import MultinomialNB
from sklearn.svm import SVC, LinearSVC
from sklearn.metrics import classification_
report, f1_score, accuracy_score, confusion_
matrix
from sklearn.pipeline import Pipeline
from sklearn.grid_search import GridSearchCV
from sklearn.cross_validation import StratifiedK-
Fold, cross_val_score, train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.learning_curve import
learning_curve
```

Step 1: Load data, look around

Skipping the real first step (fleshing out specs, finding out what it is we want to be doing — often highly non-trivial in practice!), let's download the dataset we'll be using in this demo. Go to *hn.my/spam* and download the zip file. Unzip it under data subdirectory. You should see a file called SMSSpamCollection, about 0.5MB in size:

```
$ ls -l data
total 1352
-rw-r--r--@ 1 kofola  staff  477907 Mar 15  2011
SMSSpamCollection
-rw-r--r--@ 1 kofola  staff    5868 Apr 18  2011
readme
-rw-r-----@ 1 kofola  staff  203415 Dec  1 15:30
smsspamcollection.zip
```

This file contains **a collection of more than 5 thousand SMS phone messages** (see the readme file for more info):

In [2]:

```
messages = [line.rstrip() for line in open('./
data/SMSSpamCollection')]
print len(messages)
```

```
5574
```

A collection of texts is also sometimes called "corpus". Let's print the first ten messages in this SMS corpus:

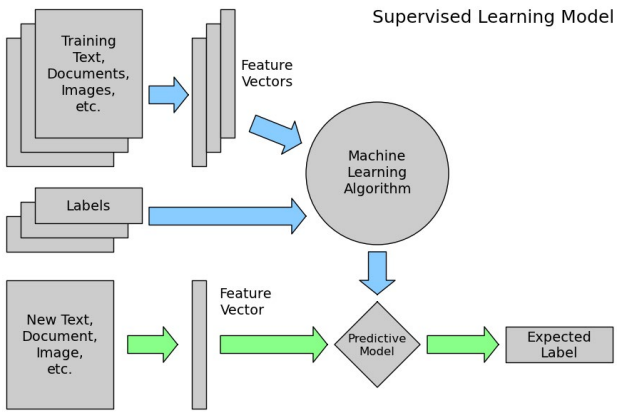
In [3]:

```
for message_no, message in
enumerate(messages[:10]):
    print message_no, message
```

```
0 ham  Go until jurong point, crazy.. Available
only in bugis n great world la e buffet... Cine
there got amore wat...
1 ham  Ok lar... Joking wif u oni...
2 spam Free entry in 2 a wkly comp to win FA
Cup final tkts 21st May 2005. Text FA to 87121 to
receive entry question(std txt rate)T&C's apply
08452810075over18's
3 ham  U dun say so early hor... U c already
then say...
4 ham  Nah I don't think he goes to usf, he
lives around here though
5 spam FreeMsg Hey there darling it's been 3
week's now and no word back! I'd like some fun
you up for it still? Tb ok! XxX std chgs to
send, £1.50 to rcv
6 ham  Even my brother is not like to speak with
me. They treat me like aids patent.
7 ham  As per your request 'Melle Melle (Oru
Minnaminunginte Nurungu Vettam)' has been set
as your callertune for all Callers. Press *9 to
copy your friends Callertune
8 spam WINNER!! As a valued network customer
you have been selected to receivea £900 prize
reward! To claim call 09061701461. Claim code
KL341. Valid 12 hours only.
9 spam Had your mobile 11 months or more? U R
entitled to Update to the latest colour mobiles
with camera for Free! Call The Mobile Update Co
FREE on 08002986030
```

We see that this is a TSV ("tab separated values") file, where the first column is a label saying whether the given message is a normal message ("ham") or "spam". The second column is the message itself.

This corpus will be our labeled training set. Using these ham/spam examples, we'll **train a machine learning model to learn to discriminate between ham/spam automatically.** Then, with a trained model, we'll be able to **classify arbitrary unlabeled messages** as ham or spam.



Instead of parsing TSV (or CSV, or Excel...) files by hand, we can use Python's pandas library to do the work for us:

In [4]:

```
messages = pandas.read_csv('./data/SMSSpamCol-
lection', sep='\t', quoting=csv.QUOTE_NONE,
                           names=["label", "message"])
print messages
```

```
     label                                message
0      ham        Go until jurong point, crazy..
1      ham         Ok lar... Joking wif u oni...
2     spam    Free entry in 2 a wkly comp to ...
3      ham              U dun say so early hor...
4      ham   Nah I don't think he goes to usf...
5     spam     FreeMsg Hey there darling it's...
6      ham     Even my brother is not like to...
7      ham   As per your request 'Melle Melle...
8     spam      WINNER!! As a valued network ...
...    ...                                   ...
5567   ham                          Huh y lei...
5568  spam      REMINDER FROM O2: To get 2.50...
5569  spam  This is the 2nd time we have trie...
5570   ham  Will ü b going to esplanade fr home?
5571   ham  Pity, * was in mood for that. So...
```

```
5572   ham  The guy did some bitching but I'd...
5573   ham  Rofl. Its true to its name

[5574 rows x 2 columns]
```

With pandas, we can also view aggregate statistics easily:

In [5]:

```
messages.groupby('label').describe()
```

Out[5]:

		message
label		
ham	**count**	4827
	unique	4518
	top	Sorry, I'll call later
	freq	30
spam	**count**	747
	unique	653
	top	Please call our customer service representativ...
	freq	4

How long are the messages?

In [6]:

```
messages['length'] = messages['message'].
map(lambda text: len(text))
print messages.head()
```

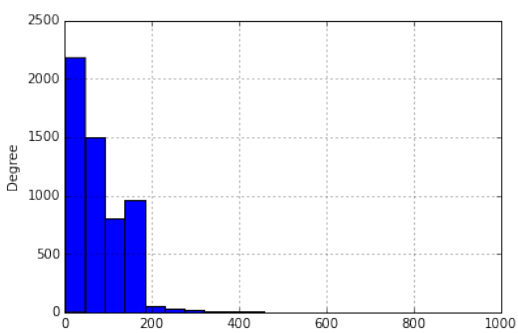
```
  label                          message  length
0   ham  Go until jurong point, crazy...     111
1   ham    Ok lar... Joking wif u oni...      29
2  spam  Free entry in 2 a wkly comp ...     155
3   ham  U dun say so early hor... U ...      49
4   ham  Nah I don't think he goes to...      61
```

In [7]:

```
messages.length.plot(bins=20, kind='hist')
```

Out[7]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x10dd7a90>
```



In [8]:

```
messages.length.describe()
```

Out[8]:

```
count    5574.000000
mean       80.604593
std        59.919970
min         2.000000
25%        36.000000
50%        62.000000
75%       122.000000
max       910.000000
Name: length, dtype: float64
```

What is that super long message?

In [9]:

```
print list(messages.message[messages.length >
900])
```

```
["For me the love should start with attraction.i
should feel that I need her every time around
<...>
she is with me.I would like to say a lot..will
tell later.."]
```

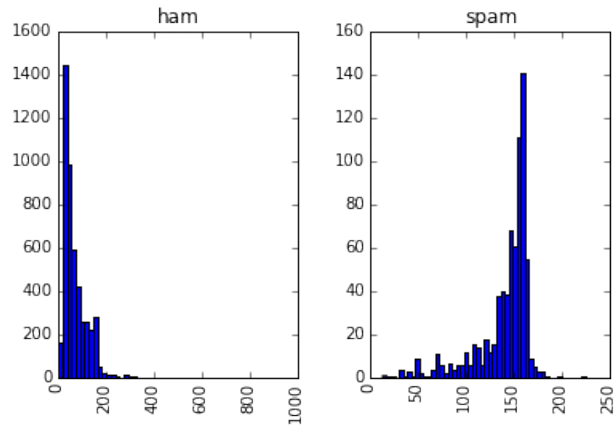
Is there any difference in message length between spam and ham?

In [10]:

```
messages.hist(column='length', by='label',
bins=50)
```

Out[10]:

```
array([<matplotlib.axes._subplots.AxesSubplot
object at 0x11270da50>,
       <matplotlib.axes._subplots.AxesSubplot
object at 0x1126c7750>], dtype=object)
```



Good fun, but how do we make a computer understand the plain text messages themselves? Or can it under such malformed gibberish at all?

Step 2: Data preprocessing

In this section we'll massage the raw messages (sequence of characters) into vectors (sequences of numbers).

The mapping is not 1-to-1; we'll use the bag-of-words approach, where each unique word in a text will be represented by one number.

As a first step, let's write a function that will split a message into its individual words:

```
In [11]:
def split_into_tokens(message):
    message = unicode(message, 'utf8')  # con-
vert bytes into proper unicode
    return TextBlob(message).words
```

Here are some of the original texts again:

```
In [12]:
messages.message.head()

Out[12]:
0    Go until jurong point, crazy...
1    Ok lar... Joking wif u oni...
2    Free entry in 2 a wkly comp to win...
3    U dun say so early hor... U c already...
4    Nah I don't think he goes to usf, he...
Name: message, dtype: object
```

...and here are the same messages, tokenized:

```
In [13]:
messages.message.head().apply(split_into_tokens)

Out[13]:
0    [Go, until, jurong, point, crazy...
1    [Ok, lar, Joking, wif, u, oni]
2    [Free, entry, in, 2, a, wkly, comp, to,
3    [U, dun, say, so, early, hor, U, c,
4    [Nah, I, do, n't, think, he, goes, to, usf,
```

NLP questions:

- Do capital letters carry information?
- Does distinguishing inflected form ("goes" vs. "go") carry information?
- Do interjections, determiners carry information?

In other words, we want to better "normalize" the text.

With textblob, we'd detect part-of-speech (POS) tags with:

```
In [14]:
TextBlob("Hello world, how is it going?").tags
# list of (word, POS) pairs

Out[14]:
[(u'Hello', u'UH'),
 (u'world', u'NN'),
 (u'how', u'WRB'),
 (u'is', u'VBZ'),
 (u'it', u'PRP'),
 (u'going', u'VBG')]
```

and normalize words into their base form (lemmas) with:

```
In [15]:
def split_into_lemmas(message):
    message = unicode(message, 'utf8').lower()
    words = TextBlob(message).words
    # for each word, take its "base form"= lemma
    return [word.lemma for word in words]

messages.message.head().apply(split_into_lemmas)

Out[15]:
0    [go, until, jurong, point, crazy,
1    [ok, lar, joking, wif, u, oni]
2    [free, entry, in, 2, a, wkly, comp, to,
3    [u, dun, say, so, early, hor, u, c,
4    [nah, i, do, n't, think, he, go, to, usf,
Name: message, dtype: object
```

Better. You can probably think of many more ways to improve the preprocessing: decoding HTML entities (those & and < we saw above); filtering out stop words (pronouns, etc.); adding more features, such as a word-in-all-caps indicator, and so on.

Step 3: Data to vectors

Now we'll convert each message, represented as a list of tokens (lemmas) above, into a vector that machine learning models can understand.

Doing that requires essentially three steps, in the bag-of-words model:

1. Counting how many times a word occurs in each message (term frequency)
2. Weighting the counts, so that frequent tokens get lower weight (inverse document frequency)
3. Normalizing the vectors to unit length to abstract from the original text length (L2 norm)

Each vector has as many dimensions as there are unique words in the SMS corpus:

```
In [16]:
bow_transformer =
CountVectorizer(analyzer=split_into_lemmas).
fit(messages['message'])
print len(bow_transformer.vocabulary_)
```

```
8874
```

Here we used `scikit-learn` (sklearn), a powerful Python library for teaching machine learning. It contains a multitude of various methods and options.

Let's take one text message and get its bag-of-words count as a vector, putting to use our new `bow_transformer`:

```
In [17]:
message4 = messages['message'][3]
print message4
```

```
U dun say so early hor... U c already then
say...
```

```
In [18]:
bow4 = bow_transformer.transform([message4])
print bow4
print bow4.shape
```

```
  (0, 1158)    1
  (0, 1899)    1
  (0, 2897)    1
  (0, 2927)    1
  (0, 4021)    1
  (0, 6736)    2
  (0, 7111)    1
  (0, 7698)    1
  (0, 8013)    2
(1, 8874)
```

So, nine unique words in message nr. 4, two of them appear twice, the rest only once. Sanity check: what are these words that appear twice?

```
In [19]:
print bow_transformer.get_feature_names()[6736]
print bow_transformer.get_feature_names()[8013]
```

```
say
u
```

The bag-of-words counts for the entire SMS corpus are a large, sparse matrix:

```
In [20]:
messages_bow = bow_transformer.
transform(messages['message'])
print 'sparse matrix shape:', messages_bow.shape
print 'number of non-zeros:', messages_bow.nnz
print 'sparsity: %.2f%%' % (100.0 * messages_
bow.nnz / (messages_bow.shape[0] * messages_bow.
shape[1]))
```

```
sparse matrix shape: (5574, 8874)
number of non-zeros: 80272
sparsity: 0.16%
```

And finally, after the counting, the term weighting and normalization can be done with TF-IDF, using scikit-learn's TfidfTransformer:

```
In [21]:
tfidf_transformer = TfidfTransformer().
fit(messages_bow)
tfidf4 = tfidf_transformer.transform(bow4)
print tfidf4
```

```
  (0, 8013)    0.305114653686
  (0, 7698)    0.225299911221
  (0, 7111)    0.191390347987
  (0, 6736)    0.523371210191
  (0, 4021)    0.456354991921
  (0, 2927)    0.32967579251
  (0, 2897)    0.303693312742
  (0, 1899)    0.24664322833
  (0, 1158)    0.274934159477
```

What is the IDF (inverse document frequency) of the word "u"? Of word "university"?

```
In [22]:
print tfidf_transformer.idf_[bow_transformer.
vocabulary_['u']]
print tfidf_transformer.idf_[bow_transformer.
vocabulary_['university']]
```

```
2.85068150539
8.23975323521
```

To transform the entire bag-of-words corpus into TF-IDF corpus at once:

```
In [23]:
messages_tfidf = tfidf_transformer.
transform(messages_bow)
print messages_tfidf.shape
```

```
(5574, 8874)
```

There are a multitude of ways in which data can be proprocessed and vectorized. These two steps, also called "feature engineering," are typically the most time consuming and unsexy parts of building a predictive pipeline, but they are very important and require some experience. The trick is to evaluate constantly: analyze model for the errors it makes, improve data cleaning and preprocessing, brainstorm for new features, evaluate....

Step 4: Training a model, detecting spam

With messages represented as vectors, we can finally train our spam/ham classifier. This part is pretty straightforward, and there are many libraries that realize the training algorithms.

We'll be using scikit-learn here, choosing the Naive Bayes classifier to start with:

```
In [24]:
%time spam_detector = MultinomialNB().
fit(messages_tfidf, messages['label'])
```

```
CPU times: user 4.51 ms, sys: 987 µs, total:
5.49 ms
Wall time: 4.77 ms
```

Let's try classifying our single random message:

```
In [25]:
print 'predicted:', spam_detector.predict(tfidf4)
[0]
print 'expected:', messages.label[3]
```

```
predicted: ham
expected: ham
```

Hooray! You can try it with your own texts, too.

A natural question is to ask, how many messages do we classify correctly overall?

```
In [26]:
all_predictions = spam_detector.
predict(messages_tfidf)
print all_predictions
```

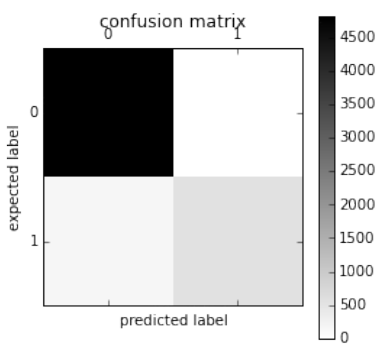
```
['ham' 'ham' 'spam' ..., 'ham' 'ham' 'ham']
```

```
In [27]:
print 'accuracy', accuracy_
score(messages['label'], all_predictions)
print 'confusion matrix\n', confusion_
matrix(messages['label'], all_predictions)
print '(row=expected, col=predicted)'
```

```
accuracy 0.969501255831
confusion matrix
[[4827    0]
 [ 170  577]]
(row=expected, col=predicted)
```

```
In [28]:
plt.matshow(confusion_matrix(messages['label'],
all_predictions), cmap=plt.cm.binary,
interpolation='nearest')
plt.title('confusion matrix')
plt.colorbar()
plt.ylabel('expected label')
plt.xlabel('predicted label')
```

```
Out[28]:
<matplotlib.text.Text at 0x11643f6d0>
```



From this confusion matrix, we can compute precision and recall, or their combination (harmonic mean) F1:

```
In [29]:
print classification_report(messages['label'],
all_predictions)
```

```
             precision    recall  f1-score   support

        ham       0.97      1.00      0.98      4827
       spam       1.00      0.77      0.87       747
avg/total       0.97      0.97      0.97      5574
```

There are quite a few possible metrics for evaluating model performance. Which one is the most suitable depends on the task. For example, the cost of mispredicting "spam" as "ham" is probably much lower than mispredicting "ham" as "spam."

Step 5: How to run experiments?

In the above evaluation, we committed a cardinal sin. For simplicity of demonstration, we evaluated accuracy on the same data we used for training. **Never evaluate on the same dataset you train on! Bad! Incest!**

Such evaluation tells us nothing about the true predictive power of our model. If we simply remembered each example during training, the accuracy on training data would trivially be 100%, even though we wouldn't be able to classify any new messages.

A proper way is to split the data into a training/test set, where the model only ever sees the training data during its model fitting and parameter tuning. The test data is never used in any way — thanks to this process, we make sure we are not cheating, and that our final evaluation on test data is representative of true predictive performance.

```
In [30]:
msg_train, msg_test, label_train, label_test = \
    train_test_split(messages['message'],
messages['label'], test_size=0.2)

print len(msg_train), len(msg_test), len(msg_
train) + len(msg_test)
```

```
4459 1115 5574
```

So, as requested, the test size is 20% of the entire dataset (1115 messages out of total 5574), and the training is the rest (4459 out of 5574).

Let's recap the entire pipeline up to this point, putting the steps explicitly into scikit-learn's Pipeline:

```
In [31]:
def split_into_lemmas(message):
    message = unicode(message, 'utf8').lower()
    words = TextBlob(message).words
    # for each word, take its "base form" =
lemma
    return [word.lemma for word in words]

pipeline = Pipeline([
    ('bow', CountVectorizer(analyzer=split_into_
lemmas)),  # strings to token integer counts
    ('tfidf', TfidfTransformer()),  # integer
counts to weighted TF-IDF scores
    ('classifier', MultinomialNB()),  # train on
TF-IDF vectors w/ Naive Bayes classifier
])
```

A common practice is to partition the training set again, into smaller subsets; for example, 5 equally sized subsets. Then we train the model on four parts and compute accuracy on the last part (called "validation set"). Repeated five times (taking a different part for evaluation each time), we get a sense of model "stability." If the model gives wildly different scores for different subsets, it's a sign something is wrong (bad data or bad model variance). Go back, analyze errors, re-check input data for garbage, re-check data cleaning.

In our case, everything goes smoothly though:

```
In [32]:
scores = cross_val_score(pipeline,  # steps to
convert raw messages into models
       msg_train,  # training data
       label_train,  # training labels
       cv=10,  # split data randomly into 10
parts: 9 for training, 1 for scoring
       scoring='accuracy',  # which scoring
metric?
       n_jobs=-1,  # -1 = use all cores = faster
       )
print scores
```

```
[ 0.93736018  0.96420582  0.94854586  0.94183445
  0.96412556  0.94382022  0.94606742  0.96404494
  0.94831461  0.94606742]
```

The scores are indeed a little bit worse than when we trained on the entire dataset (5574 training examples, accuracy 0.97). They are fairly stable, though:

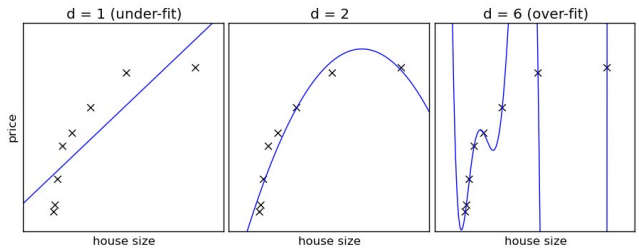
In [33]:

```
print scores.mean(), scores.std()
```

```
0.9504386476 0.00947200821389
```

A natural question is, how can we improve this model? The scores are already high here, but how would we go about improving a model in general?

Naive Bayes is an example of a high bias - low variance classifier (aka, simple and stable, not prone to overfitting). An example from the opposite side of the spectrum would be Nearest Neighbour (kNN) classifiers, or Decision Trees, with their low bias but high variance (easy to overfit). Bagging (Random Forests) as a way to lower variance, by training many (high-variance) models and averaging.



In other words:

- **high bias**: classifier is opinionated. Not as much room to change its mind with data, it has its own ideas. On the other hand, not as much room it can fool itself into overfitting either (picture on the left).
- **low bias**: classifier more obedient, but also more neurotic. Will do exactly what you ask it to do, which, as everybody knows, can be a real nuisance (picture on the right).

In [34]:

```
def plot_learning_curve(estimator, title, X, y,
ylim=None, cv=None,
                        n_jobs=-1, train_
sizes=np.linspace(.1, 1.0, 5)):
    """
    Generate a simple plot of the test and tran-
ing learning curve.

    Parameters
    ----------
    estimator : object type that implements the
"fit" and "predict" methods
        An object of that type which is cloned
for each validation.

    title : string
        Title for the chart.

    X : array-like, shape (n_samples, n_fea-
tures)
        Training vector, where n_samples is the
number of samples and
        n_features is the number of features.

    y : array-like, shape (n_samples) or (n_sam-
ples, n_features), optional
        Target relative to X for classification
or regression;
        None for unsupervised learning.

    ylim : tuple, shape (ymin, ymax), optional
        Defines minimum and maximum yvalues plot-
ted.

    cv : integer, cross-validation generator,
optional
        If an integer is passed, it is the
number of folds (defaults to 3).
        Specific cross-validation objects can be
passed, see
        sklearn.cross_validation module for the
list of possible objects

    n_jobs : integer, optional
        Number of jobs to run in parallel
(default 1).
    """
    plt.figure()
    plt.title(title)
    if ylim is not None:
        plt.ylim(*ylim)
    plt.xlabel("Training examples")
    plt.ylabel("Score")
    train_sizes, train_scores, test_scores =
learning_curve(
        estimator, X, y, cv=cv, n_jobs=n_jobs,
train_sizes=train_sizes)
    train_scores_mean = np.mean(train_scores,
axis=1)
    train_scores_std = np.std(train_scores,
axis=1)
    test_scores_mean = np.mean(test_scores,
axis=1)
```

```
    test_scores_std = np.std(test_scores,
axis=1)
    plt.grid()

    plt.fill_between(train_sizes, train_scores_
mean - train_scores_std,
                     train_scores_mean + train_
scores_std, alpha=0.1,
                     color="r")
    plt.fill_between(train_sizes, test_scores_
mean - test_scores_std,
                     test_scores_mean + test_
scores_std, alpha=0.1, color="g")
    plt.plot(train_sizes, train_scores_mean,
'o-', color="r",
             label="Training score")
    plt.plot(train_sizes, test_scores_mean,
'o-', color="g",
             label="Cross-validation score")

    plt.legend(loc="best")
    return plt
```

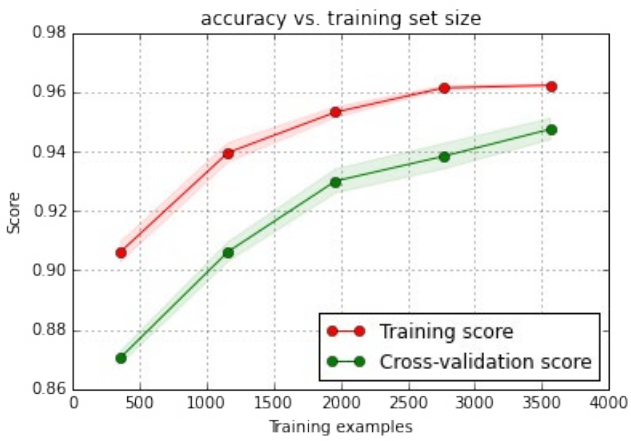
In [35]:

```
%time plot_learning_curve(pipeline, "accuracy
vs. training set size", msg_train, label_train,
cv=5)
```

```
CPU times: user 382 ms, sys: 83.1 ms, total: 465 ms
Wall time: 28.5 s
```

Out[35]:

```
<module 'matplotlib.pyplot' from '/Volumes/work/
workspace/vew/sklearn_intro/lib/python2.7/site-
packages/matplotlib/pyplot.pyc'>
```



(We're effectively training on 64% of all available data: we reserved 20% for the test set above, and the 5-fold cross validation reserves another 20% for validation sets => 0.8*0.8*5574=3567 training examples left.)

Since performance keeps growing, both for training and cross validation scores, we see our model is not complex/flexible enough to capture all nuance, given little data. In this particular case, it's not very pronounced, since the accuracies are high anyway.

At this point, we have two options:

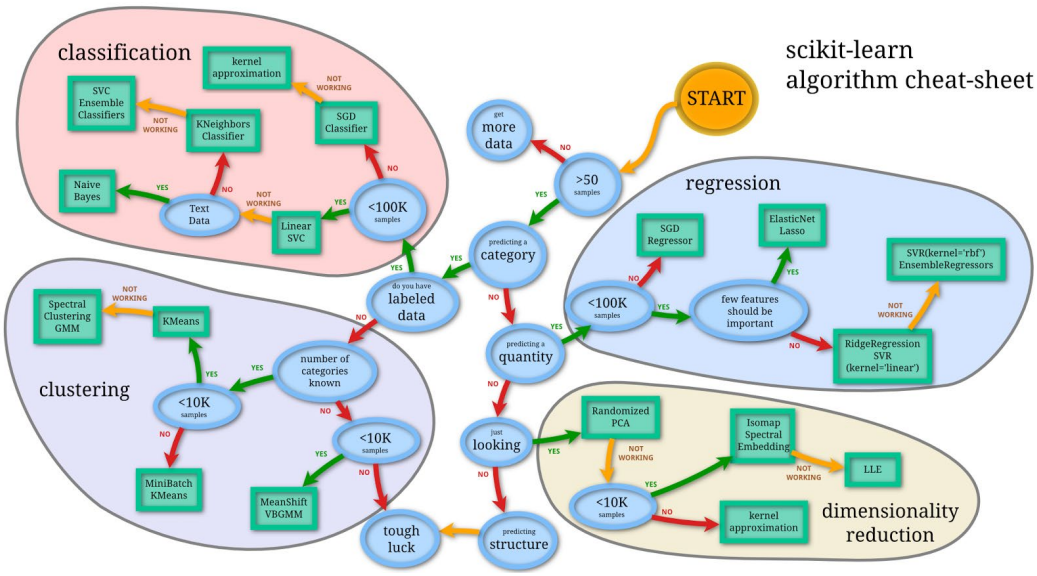
1. Use more training data to overcome low model complexity
2. Use a more complex (lower bias) model to start with in order to get more out of the existing data

Over the last few years, as massive training data collections become more available, and as machines get faster, approach #1 is becoming more and more popular (simpler algorithms, more data). Straightforward algorithms, such as Naive Bayes, also have the added benefit of being easier to interpret (compared to some more complex, black-box models, like neural networks).

Knowing how to evaluate models properly, we can now explore how different parameters affect the performance.

Step 6: How to tune parameters?

What we've seen so far is only a tip of the iceberg: there are many other parameters to tune. One example is what algorithm to use for training.



We've used Naive Bayes above, but scikit-learn supports many classifiers out of the box: Support Vector Machines, Nearest Neighbours, Decision Trees, Ensamble methods...

We can ask: What is the effect of IDF weighting on accuracy? Does the extra processing cost of lemmatization (vs. just plain words) really help?

Let's find out:

```
In [37]:
params = {
    'tfidf__use_idf': (True, False),
    'bow__analyzer': (split_into_lemmas, split_
into_tokens),
}

grid = GridSearchCV(
    pipeline,  # pipeline from above
    params,  # parameters to tune via cross
validation
    refit=True,  # fit using all available data at
the end, on the best found param combination
    n_jobs=-1,  # number of cores to use for
parallelization; -1 for "all cores"
    scoring='accuracy',  # what score are we
optimizing?
    cv=StratifiedKFold(label_train, n_folds=5),
# what type of cross validation to use)
```

```
In [38]:
%time nb_detector = grid.fit(msg_train, label_
train)
print nb_detector.grid_scores_

CPU times: user 4.09 s, sys: 291 ms, total: 4.38 s
Wall time: 20.2 s
[mean: 0.94752, std: 0.00357, params: {'tfidf__
use_idf': True, 'bow__analyzer': <function
split_into_lemmas at 0x1131e8668>}, mean:
0.92958, std: 0.00390, params: {'tfidf__use_
idf': False, 'bow__analyzer': <function split_
into_lemmas at 0x1131e8668>}, mean: 0.94528,
std: 0.00259, params: {'tfidf__use_idf': True,
'bow__analyzer': <function split_into_tokens
at 0x11270b7d0>}, mean: 0.92868, std: 0.00240,
params: {'tfidf__use_idf': False, 'bow__ana-
lyzer': <function split_into_tokens at
0x11270b7d0>}]
```

(best parameter combinations are displayed first: in this case, use_idf=True and analyzer=split_into_lemmas take the prize).

A quick sanity check:

In [39]:

```
print nb_detector.predict_proba(["Hi mom, how
are you?"])[0]
print nb_detector.predict_proba(["WINNER! Credit
for free!"])[0]
```

```
[ 0.99383955  0.00616045]
[ 0.29663109  0.70336891]
```

The predict_proba returns the predicted probability for each class (ham, spam). In the first case, the message is predicted to be ham with > 99% probability, and spam with < 1%. So if forced to choose, the model will say "ham":

In [40]:

```
print nb_detector.predict(["Hi mom, how are
you?"])[0]
print nb_detector.predict(["WINNER! Credit for
free!"])[0]
```

```
ham
spam
```

And overall scores on the test set, the one we haven't used at all during training:

In [41]:

```
predictions = nb_detector.predict(msg_test)
print confusion_matrix(label_test, predictions)
print classification_report(label_test,
predictions)
```

```
[[973   0]
 [ 46  96]]
             precision    recall  f1-score   support

        ham       0.95      1.00      0.98       973
       spam       1.00      0.68      0.81       142
avg/total       0.96      0.96      0.96      1115
```

This is then the realistic predictive performance we can expect from our spam detection pipeline, when using lowercase with lemmatization, TF-IDF and Naive Bayes for classifier.

Let's try with another classifier: Support Vector Machines (SVM). SVMs are a great starting point when classifying text data, getting state of the art results very quickly and with pleasantly little tuning (although a bit more than Naive Bayes):

In [42]:

```
pipeline_svm = Pipeline([
    ('bow', CountVectorizer(analyzer=split_into_
lemmas)),
    ('tfidf', TfidfTransformer()),
    ('classifier', SVC()),  # <== change here
])

# pipeline parameters to automatically explore
and tune
param_svm = [
  {'classifier__C': [1, 10, 100, 1000], 'classi-
fier__kernel': ['linear']},
  {'classifier__C': [1, 10, 100, 1000], 'clas-
sifier__gamma': [0.001, 0.0001], 'classifier__
kernel': ['rbf']},
]

grid_svm = GridSearchCV(
    pipeline_svm,  # pipeline from above
    param_grid=param_svm,  # parameters to tune
via cross validation
    refit=True,  # fit using all data, on the best
detected classifier
    n_jobs=-1,  # number of cores to use for
parallelization; -1 for "all cores"
    scoring='accuracy',  # what score are we
optimizing?
    cv=StratifiedKFold(label_train, n_folds=5),
# what type of cross validation to use
)
```

In [43]:

```
%time svm_detector = grid_svm.fit(msg_train,
label_train) # find the best combination from
param_svm
print svm_detector.grid_scores_
```

```
CPU times: user 5.24 s, sys: 170 ms, total: 5.41
s
Wall time: 1min 8s
[mean: 0.98677, std: 0.00259, params: {'clas-
sifier__kernel': 'linear', 'classifier__C': 1},
mean: 0.98654, std: 0.00100, params: {'classi-
fier__kernel': 'linear', 'classifier__C': 10},
<...>
mean: 0.97040, std: 0.00587, params: {'classi-
fier__gamma': 0.0001, 'classifier__kernel': 'rbf',
'classifier__C': 1000}]
```

So apparently, linear kernel with C=1 is the best parameter combination.

Sanity check again:

```
In [44]:
print svm_detector.predict(["Hi mom, how are you?"])[0]
print svm_detector.predict(["WINNER! Credit for free!"])[0]

ham
spam

In [45]:
print confusion_matrix(label_test, svm_detector.predict(msg_test))
print classification_report(label_test, svm_detector.predict(msg_test))

[[965   8]
 [ 13 129]]
             precision    recall  f1-score   support

        ham       0.99      0.99      0.99       973
       spam       0.94      0.91      0.92       142
avg / total       0.98      0.98      0.98      1115
```

This is then the realistic predictive performance we can expect from our spam detection pipeline when using SVMs.

Step 7: Productionalizing a predictor

With basic analysis and tuning done, the real work (engineering) begins.

The final step for a production predictor would be training it on the entire dataset again, to make full use of all the data available. We'd use the best parameters found via cross validation above, of course. This is very similar to what we did in the beginning, but this time having insight into its behavior and stability. Evaluation was done honestly on distinct train/test subset splits.

The final predictor can be serialized to disk, so that the next time we want to use it, we can skip all training and use the trained model directly:

```
In [46]:
# store the spam detector to disk after training
with open('sms_spam_detector.pkl', 'wb') as fout:
    cPickle.dump(svm_detector, fout)

# ...and load it back, whenever needed, possibly on a different machine
svm_detector_reloaded = cPickle.load(open('sms_spam_detector.pkl'))
```

The loaded result is an object that behaves identically to the original:

```
In [47]:
print 'before:', svm_detector.predict([message4])[0]
print 'after:', svm_detector_reloaded.predict([message4])[0]

before: ham
after: ham
```

Another important part of a production implementation is performance. After a rapid, iterative model tuning and parameter search as shown here, a well-performing model can be translated into a different language and optimized. Would trading a few accuracy points give us a smaller, faster model? Is it worth optimizing memory usage, perhaps using mmap to share memory across processes?

Note that optimization is not always necessary; always start with actual profiling.

Other things to consider here, for a production pipeline, are robustness (service failover, redundancy, load balancing), monitoring (including auto-alerts on anomalies) and HR fungibility (avoiding "knowledge silos" of how things are done, arcane/lock-in technologies, black art of tuning results). These days, even the open source world can offer viable solutions in all of these areas. All the tools shown today are free for commercial use under OSI-approved open source licenses. ■

Radim cut his IT teeth on C64 (BASIC 2.0 and assembly) in the early 90s. After the usual journey through Windows (.NET competition, C#), he ended up using and developing for UNIXy systems — mostly Debian and OS X. His fondness of low-level optimization (graphics and AI game programmer, C++) eventually transitioned into high-level optimization of business solutions and processes. Radim has been running his own freelance & consulting business [radimrehurek.com], helping companies develop scalable systems for search and text analysis. Radim holds a Project Management and Situational Leadership management certificates from the Center for Leader Studies in Prague.

Reprinted with permission of the original author.
First appeared in *hn.my/dspy* (radimrehurek.com)

Under Pressure

By PETE FRENCH

So, Sunday morning, a time for relaxing after going out the night before, right? Márcia and I had indeed gone out last night. Some very kind friends volunteered to babysit Tiago for the evening so we could go and see The Birthday Massacre play, as we hadn't been out in a while. They entertained him with some science demos using dry-ice, which he loved, and they even left some of it behind so that we could do some more this morning for fun. Indeed, he was very keen, and as soon as he had finished breakfast he said "Daddy, can we do some more experiments please?"

"Sure," I said, and went to get the dry-ice. At which point I started to realize that I may have done something a little foolish!

Dry-ice is solid carbon dioxide. At minus seventy nine degrees Celsius, it is a little on the chilly side, so to keep it from vanishing overnight I had put the remaining crystals into a small thermos flask. I went to get this, and because some might have turned to gas overnight I went onto the balcony to release the pressure before opening it entirely. But the little valve was jammed. "Odd," I thought, and tried to loosen the whole top. Again, no luck. At which point I realized that either the extreme cold, or the pressure inside the flask had jammed the top.

OK, so the top is jammed. Is this a problem? Quick calculation in my head: dry-ice expands by about 850 times when it turns to gas. Based on the amount I put in, and the size of the flask, that's going to be roughly a hundred atmospheres when it all sublimes. Which is inevitable as the flask isn't going to keep it at minus seventy nine forever. The chances of a small domestic thermos flask being able to resist a hundred atmospheres of pressure without rupturing? Well, that's pretty much zero.

I looked at the flask with that awful sinking feeling you get when you realize you have created something which is inevitably going to explode at some point in the future, and there's nothing you can do about it.

So, let's ring the dry ice people...

Well, I can't be the first person to do this, surely? Indeed a quick google shows that the thermos' webpage explicitly says to not put dry ice in one as it may cause the top "to eject forcefully" — a lovely piece of understatement there. So I found a dry-ice supplier in London and rang them to ask what to do.

"I dunno mate," the man on the helpline said, "You are the first person we have had who has ever done this." I asked what he would suggest.

"Try ringing the fire brigade?"

So, let's ring the fire brigade

I rang the fire brigade. "Uh, how do you expect us to be able to help you?" said the woman on the phone. I replied that I assumed the fire brigade encountered potentially exploding cylinders all the time and would have some way of handling the situation. "Oh yes," she said "We call the police to have them cordoned off, and then we get well away from them." Ah, not quite what I had expected. "Maybe you could try burying it in the garden?" she said helpfully. "I live in a flat," I replied.

So, let's ring the police

Actually, I really didn't want to do that. It occurred to me that Islington police were unlikely to have any form of containment for my small exploding flask problem and might have to call someone like the bomb squad. I baulked at escalating the problem that far unless absolutely necessary.

What I really needed was advice from a friend who would understand the physics of the situation, wouldn't be surprised in the slightest at being asked for help, and with a proven track record of patiently handling my somewhat misguided and potentially dangerous schemes.

So let's ring Steer

Ah, finally some useful advice. Someone who does the calculation in their own head, comes to the same numbers as me and the same "oh, shit!" conclusion. Yes, it's going to go bang, most likely at the valve end first. We discussed a few solutions: burying it in the window box is not a good idea, we can't get it cooled back down, he doesn't have any emergency contacts at the universities, and I wouldn't want to carry it across town anyway. I wasn't sure how much longer it would last and had no way of calculating it.

"What about throwing it into the canal?" I suggested

"It'll float," he said.

This is true. But then I thought "not if I weight it down with something," We have tiles left over from the kitchen. Maybe if I put them in a bag with the flask? Yes, that should work, as long as there's no air trapped in the bag in an odd pocket, it should sink nicely and go bang at the bottom of the canal where it can't hurt anyone.

Thus a plan was formed....

The *other* big run in London on April 26, 2015

I wrapped the flask in a Toys-R-Us bag with a load of ceramic tiles and wrapped it all round with parcel tape. It made a heavy cylindrical bundle, and I arranged it with the dangerous end pointing away from me. I then placed it in front of me, and started to run down Packington Street towards the canal.

Let's just say I was not feeling particularly calm about this by this point. The flask had been warming outside for a good hour now, and the weight of the tiles made it hard to hold away from my body, plus I had to keep zig-zagging to avoid pointing the bit which was going to explode at people coming up the street towards me. Thus, weaving left and right, I sprinted down the hill, through the Packington Estate, and to the edge of the canal, where I hurled the bag into the center of the canal with a splash.

..And there it floated...

"Noooo!!!!" Steer's words about making sure there were no air pockets trapped in the back suddenly came into my head so very clearly.

"Sink! Please sink!" I cried at the bag.

People were staring at me a bit by now. The canal is a crowded place on a Sunday morning, and I suspect my behavior was somewhat out of the ordinary. I sat by the canal. The bag was too far out to retrieve and starting to drift in the direction of the houseboats moored on the other side. Oh god, I started to realize quite how much worse the phone call to the police was now going to have to be.

But then, very slowly, one end of the bag began to bubble, the other end raised into the air, and in the style of Titanic, it slipped almost vertically below the surface of the water and sank to the bottom.

I stared at the water for a long while. I texted Steer and Márcia let them know I was OK and that it had worked. I was about to stand up and head home when a curious thing happened. The surface of the canal started bubbling over the spot where the bag had sunk and continued to do so.

More air trapped in the bag? No, it went on and on, a continuous stream of bubbles. I realized that the flask must have just ruptured, maybe a minute and a half after throwing it into the water. That's how close I was to having it explode in my hands.

I watched the bubbles until they finally finished, and (trembling a little bit I have to admit) I walked home. ■

Pete has been around on the net over the last 20 years under the nickname of "-bat.", or as "minusbat". He blogs at *minusbat.livejournal.com*

Reprinted with permission of the original author.
First appeared in *hn.my/thermos* (minusbat.livejournal.com)

Join the DuckDuckGo Open Source Community.

Create Instant Answers or share ideas and help change the future of search.

Featured IA: Regex Contributor: mintsoft
Get started at duckduckhack.com