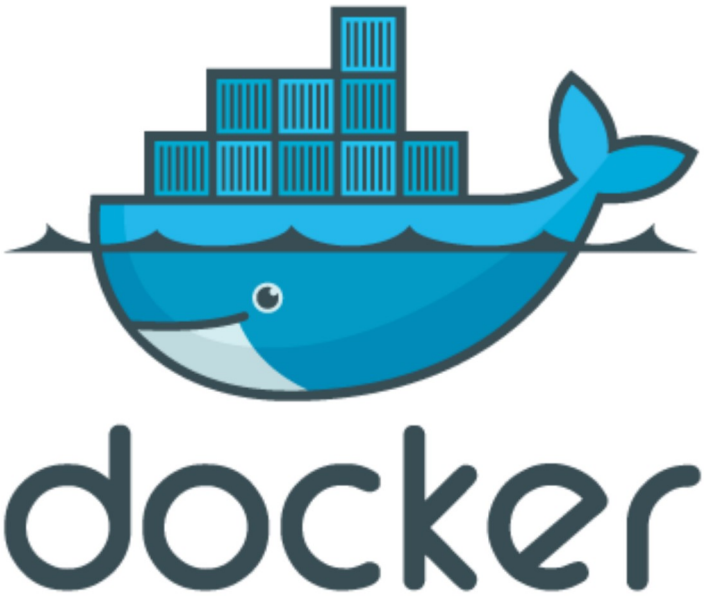
How to scale up RESTful APIs using Docker

Our last post talked about creating RESTful APIs for machine learning credit models in R using Plumber. A single instance of a Plumber app will break if there are many requests coming at a same time. We dedicate two posts on solving this scalability issue for R web APIs. This one presents ways to scale up your RESTful APIs using Docker, a recent technology that has become popular since its birth, the next one will talk about improving performance even further by using Amazon Web Service (AWS). For readers without a technical background, these are going to be a bit technical but the message is simple - using container and cloud technologies, we can improve performance of RESTful APIs easily even when the underlying languages and packages for the APIs did not consider scalability so much.

0 - 1. Docker In Brief

For those who are not familiar with Docker, it basically allows you to run multiple operating systems (OS) in one machine. Before Docker, developers make applications, configure remote servers and deploy the apps to the servers. After Docker, developers write a configuration file (called Dockerfile) to automatically configure and deploy, which has improved efficiency and reduced chances of errors during operation.



Docker
Please find this page by Docker Inc. if you would like to know more about Docker.

0 - 2. Experiment Setup

1. We will first deploy the decision tree credit model in our last post to one Docker container and see it works
2. Then, we will bombard (i.e. make a lot of requests at a same time) the API and observe symptoms
3. Lastly, we will make several Docker containers running the API and remedy the situation.

1. Baseline: One Docker Container

First, start with creating a Docker image of our credit model.

```
$ docker build -t knowru/plumber_example https://github.com/Knowru/pl
```

Then, let us run it.

```
$ docker run -p 8000:8000 -d knowru/plumber_example
```

This will create a Docker container running our Plumber app listening to port 8000.

Let us check here it works.

```
$ curl --data "@data.json" localhost:8000/predict
{"default.probability":0.3058}
```

Hooray! It worked!

Let us measure performance of our current deployment (a single container) using Siege (It is a piece of software by Joe Dog Software and allows to make many requests to a URL to see how the web application would perform in the wild reality).

```
$ siege -H 'Content-Type:application/json' "http://localhost:8000/pre
```

This is what we get:

```
SIEGE 3.0.5
Preparing 1 concurrent users for battle.
The server is now under siege..      done.
Transactions:                 100 hits
Availability:                 100.00 %
Elapsed time:                 4.83 secs
Data transferred:             0.00 MB
Response time:                0.05 secs
Transaction rate:             20.70 trans/sec
Throughput:                   0.00 MB/sec
Concurrency:                  1.00
Successful transactions:      100
Failed transactions:          0
Longest transaction:          0.08
Shortest transaction:         0.04
```

It takes about 50 ms on average to calculate default likelihood of a potential German customer if they all come one after one.

2. Symptom of Many Simultaneous Requests

Now let us make many 5 requests at a same time to see how our deployment can handle the situation.

```
$ siege -H 'Content-Type:application/json' "http://localhost:8000/pre
SIEGE 3.0.5
Preparing 1 concurrent users for battle.
The server is now under siege..      done.
Transactions:               500 hits
Availability:               100.00 %
Elapsed time:               21.97 secs
Data transferred:           0.01 MB
Response time:              0.22 secs
Transaction rate:           22.76 trans/sec
Throughput:                 0.00 MB/sec
Concurrency:                4.98
Successful transactions:    500
Failed transactions:        0
Longest transaction:        0.34
Shortest transaction:       0.09
```

We see that the average and longest response time have increased significantly:

- Response time: 0.05 secs 0.22 secs (450% degradation)
- Longest transaction: 0.08 secs 0.34 secs (325% degradation).

Now let us try to solve the problem by using multiple Docker containers.

3. Scaling Up Using Docker

Let us create 4 more Docker containers, each of which listens to a different port.

```
$ docker run -p 8001:8000 -d knowru/plumber_example
$ docker run -p 8001:8000 -d knowru/plumber_example
$ docker run -p 8001:8000 -d knowru/plumber_example
$ docker run -p 8001:8000 -d knowru/plumber_example
```

Because they are listening to different ports, we need nginx to load balance among the 5 containers.

```
$  docker run -v /home/spark/plumber_example/nginx.conf:/etc/nginx/co
```

Let us check here if our nginx works. Note that we do not use the port number 8000 anymore because our nginx listens to port 80 and distributes load to our Plumber apps running ports 8000-8004.

```
$ curl --data "@data.json" localhost/predict
{"default.probability":0.3058}
```

Good.

Let us see if we earned any performance gain:

```
$ siege -H 'Content-Type:application/json' "http://localhost/predict
 SIEGE 3.0.5
 Preparing 5 concurrent users for battle.
 The server is now under siege..      done.
 Transactions:              500 hits
 Availability:              100.00 %
 Elapsed time:              8.07 secs
 Data transferred:          0.01 MB
 Response time:             0.08 secs
 Transaction rate:          61.96 trans/sec
 Throughput:                0.00 MB/sec
 Concurrency:               4.96
 Successful transactions:   500
 Failed transactions:       0
 Longest transaction:       0.14
 Shortest transaction:      0.04
```

Yeah! Now our deployment handles multiple requests much better:

- Response time: 0.22 secs 0.08 secs (63% improvement)
- Longest transaction: 0.34 secs 0.14 secs (59% improvement).

Let us compare the results:

Concurrency

Number of requests per each thread
Total number of requests
Number of Docker containers
Average response time
Longest response time
1 100 100 1 0.05 0.08
5 100 500 1 0.22 0.34
5 100 500 5 0.08 0.14

4. Conclusion

So far, we deployed our machine learning credit model using Docker containers and observed performance improvements. In the simple situation where 5 users concurrently make 100 requests, we could reduce both average and maximum response times to less than half by utilizing five Docker containers. This looks very promising - data scientists can deploy their R models in a native R environment and do not have to worry about not being able to keep up with an increased volume!

Nevertheless, Docker containers run in a single host and a server is usually limited by memory size. Also, in some cases, you want to distribute applications to multiple servers so that even when one server fails, your overall system does not fail. In our next post, we will deploy the same model to multiple AWS servers each of which runs multiple Docker containers to further gain scalability and measure performance improvement.

How is your journey so far? Was it straight-forward or complicated? Even though we make an API ourselves, still creating, scaling and maintaining one can be resource-intensive and daunting tasks for data scientists. If you are looking for a more convenient and reliable way, please find the following resources we offer: