Example of Moving Machine-Based Environmental Models to Cloud-Based Computational Resources

Graham Dean

Introduction

This document describes the manual process of converting a locally stored environmental model written in some programming language to a cloud-based resource accessible via a webservice. A simple example is worked through to give a sense of what is required to manually convert local code to an API service that may be deployed on a variety of platforms. In this way we can work through what is required to inform tool design to automate or assist in this 'cloudifying' process.

The example model chosen is TopModel that is available as an R package from the CRAN repository[1]. TopModel is a physically based, distributed watershed model that simulates hydrologic fluxes of water through a watershed. There is also a useful PDF document explaining the various functions available in the package and provides an example of data for the Huagrahuma catchment area in Ecuador[2].

API Development

The first stage is to look through the R package and identify what functions to expose via a webservice API. These may be functions directly available in the R package, bespoke wrapper functions that provide higher-level functionality and/or re-written functions into particular programming perspectives, e.g. a functional style. In this example we will expose a single *topmodel* function.

The next stage is to identify a program language specific mechanism to support a webservice APIs. An approach suitable for the R language is to use function decorators provided by the Plumber package[3]. To expose the *topmodel* function we write a wrapper function and decorate that using the Plumber http verb (get/post/patch/delete) mechanisms:

```
# File: TopModel.R
library(topmodel)
#* @post /topmodel
model <- function(parameters, topidx, delay, rain, ET0) {
    Qsim <- topmodel(as.vector(parameters), as.matrix(topidx),
        as.matrix(delay), as.matrix(rain), as.matrix(ET0))
    return(Qsim)
}
```

This short piece of R code exposes a number of issues involved in exposing a function as a webservice API. The type and structure of the function parameter data needs to be identified and a suitable mechanism used to

[1] https://cran.r-project.org/web/packages/topmodel/index.html
[2] https://cran.r-project.org/web/packages/topmodel/topmodel.pdf
[3] https://cran.r-project.org/web/packages/plumber/index.html

communicate this data across the webservice interface. As may be seen in the R code above, the parameter data is structured as R vectors and matrices. The JSON protocol is commonly used to transfer data between applications and is used here as the default data interchange format in the Plumber package.

As mentioned, the *topmodel* R package contains example data for the Huagrahuma catchment area. This data is exported from the R system to the local filesystem using the *jsonlite* package in JSON format.

```
# File: Export.R
library(topmodel)
data(huagrahuma)
write(jsonlite::toJSON(huagrahuma, digits=NA), file="Huagrahuma.json")
```

A service wrapper is now written to load up the API exported through the TopModel.R file:

```
#!/usr//bin/env Rscript
# File: modelcloud
library(plumber)
r <- plumb("TopModel.R")
r$run(port=8000)
```

The service is run:

```
$ ./modelcloud &
...
$ Starting server to listen on port 8000
```

We can test out the service from the command line using the exported data 'Huagrahum.json' and the curl command:

```
$ curl localhost:8000/topmodel -d @Huagrahuma.json
...
$ ,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
```

This successfully calls the *topmodel* webservice but the returned values do not have the expected values. The floating point numbers have lost their precision values and have been rounded down to 0. In order to overcome this we need to construct our own custom json serialiser for the *topmodel* package and add in the "digits=NA" parameter option to get the maximum precision:

```
#!/usr//bin/env Rscript
# File: modelcloud
library(plumber)
hip_json <- function(){
  function(val, req, res, errorHandler){
    tryCatch({
      json <- jsonlite::toJSON(val,digits=NA)

      res$setHeader("Content-Type", "application/json")
      res$body <- json

      return(res$toResponse())
    }, error=function(e){
      errorHandler(req, res, e)
    })
  }
}
```

```
addSerializer("hip_json",hip_json)
r <- plumb("TopModel.R")
r$run(port=8000)
```

The exported function is now decorated to make use of our new custom serialiser:

```
# File: TopModel.R
library(topmodel)
#* @serializer hip_json
#* @post /topmodel
model <- function(parameters, topidx, delay, rain, ET0) {
    Qsim <- topmodel(as.vector(parameters), as.matrix(topidx),
        as.matrix(delay), as.matrix(rain), as.matrix(ET0))
    return(Qsim)
}
```

We can now test the the service again from the command line and it gives us numbers with the correct amount of precision in exponent format:

```
$ curl localhost:8000/topmodel -d @Huagrahuma.json
...
$ ,1.74533777750147e-05,1.74390801517205e-05,1.74248059392824e-05]
```

API Consumption

The *topmodel* API can now be used by any applications, command line utilities or programming languages that can access a REST API and support the JSON data interchange format.

Python example:

```
import requests
import json
json_file = open("Huagrahuma.json")
json_data = json.load(json_file)
r = requests.post("http://localhost:8000/topmodel", json=json_data)
r.json()
...
, 1.74533788300472e-05, 1.74390812066411e-05, 1.742480699409e-05]
```

An example application written in Python that makes use of the exposed API endpoints may be found at http://scc-dean1.lancs.ac.uk/topmodel.

A command line example to find the maximum value in the *topmodel* prediction:

```
curl localhost:8000/topmodel -d @Huagrahuma.json | jq '. | max'
0.000356525103992418
```

API Deployment

The API can be deployed on to a remote machine directly and requires that R and any dependent packages are installed. An example deployment can be found at http://scc-dean1.lancs.ac.uk/api where the R API service sits behind an nginx proxy server.

```
$ curl scc-dean1.lancs.ac.uk/api/topmodel -d @Huagrahuma.json
...
, 1.74533788300472e-05, 1.74390812066411e-05, 1.742480699409e-05]
```

Alternatively a container deployment strategy might be used via Docker or Rkt.

Conclusions

The process taken to create a *topmodel* webservice:

1. Find an appropriate package that implements TopModel.
2. Decide which functions should be exposed via the webservice.
3. Use a language specific mechanism to expose the API endpoints (function decorators via the R *plumber* package in this case).
4. Decide upon a suitable data interchange format (JSON in this example).
5. Check API via command line to ensure correct results are being obtained.
6. Make any changes required to take account of data conversions.
7. Deploy as a microservice.
8. Consume via applications, command line utilities or programming languages.

This gives us a basic microservice that provides a TopModel service. A more sophisticated implementation would take into account service registration / discovery, integration into continuous deployment and careful design of the webservice API to address usability issues.