How to create a RESTful API for a machine learning credit model in R

Even though R provides probably the most number of machine learning algorithms out there, its packages for application development are few and thus data scientists often find it difficult to push their deliverables to their organizations' production environments. As a remedy for the problem, a data scientist can create a RESTful API using open source libraries so that programmers using other languages can interact with the API. This post will explain step-by-step how to create APIs using R's open source packages.

1. What is a RESTful API?

Wait, wait, wait, RESTful what? What are APIs and RESTful APIs? Are they different?

API(Application Program Interface) is a generic term and can really mean any interface an IT service promises to other services. For instance, the below Python function written below requires to input variables a and b. The fact it takes in two variables is an API.

```
def sum(a,b):
    return a + b
```

RESTful API(REpresentational State Transfer-ful API) is a type of API. I hate the name because it does not clearly say what type of an API it is. Essentially, RESTful APIs are HTTP APIs (and I hope that HTTP sounds more familiar than RESTful). RESTful APIs use the HTTP as their common interface. At a bare minimum, the HTTP looks for two things: method and URL. For example, when you came to this web page to read our blog post, you (or your web browser conveniently) made a request to our web server with the GET method and the URL you see in your web browser. Our web server knows how to interpret your HTTP request and thus you can see our blog post at this moment.

2. How to create RESTful APIs in R using open source libraries?

1. First, create a machine learning model

To make our example a bit more interesting, let us create a machine learning model in the simplest manner possible. I have worked with an online lending company before so let us create a model that predicts whether a person is going to pay off a loan using the famous German credit data. Building a predictive model is a sophisticated process of combining arts, science and specialized business knowledge; however, in this example, we will build it without worrying too much about all these details for the sake of simplicity.

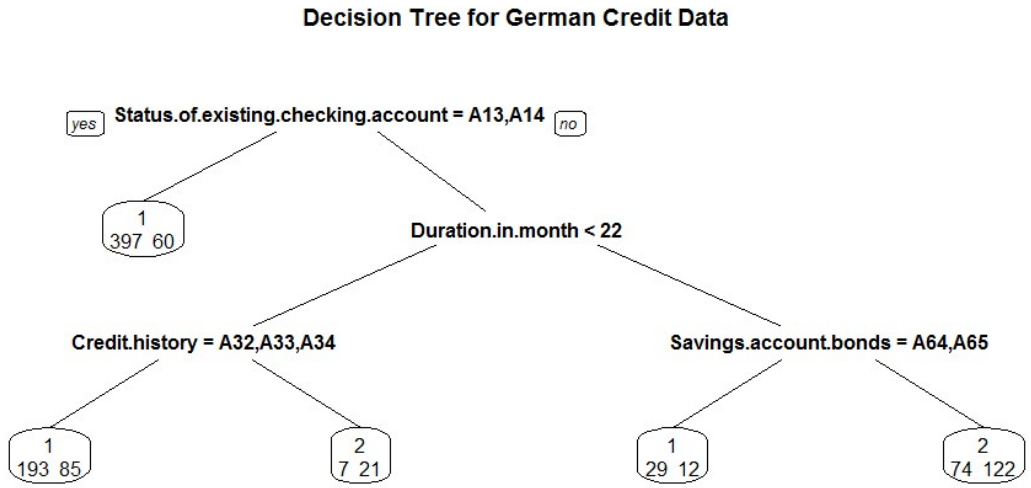
```
url <- "https://archive.ics.uci.edu/ml/machine-learning-databases/statlog/german/german.data"
col.names <- c(
  'Status of existing checking account', 'Duration in month', 'Credit history'
  , 'Purpose', 'Credit amount', 'Savings account/bonds'
  , 'Employment years', 'Installment rate in percentage of disposable income'
  , 'Personal status and sex', 'Other debtors / guarantors', 'Present residence since'
  , 'Property', 'Age in years', 'Other installment plans', 'Housing', 'Number of existing credits at this bank'
  , 'Job', 'Number of people being liable to provide maintenance for', 'Telephone', 'Foreign worker', 'Status'
  )
# Get the data
data <- read.csv(
  url
  , header=FALSE
  , sep=' '
  , col.names=col.names
)

library(rpart)
# Build a tree
# I already figured these significant variables from my first iteration (not shown in this code for simplicity)
decision.tree <- rpart(
```

```
  Status ~ Status.of.existing.checking.account + Duration.in.month + Credit.history + Savings.account.bonds
  , method="class"
  , data=data
)

install.packages("rpart.plot")
library(rpart.plot)
# Visualize the tree
# 1 is good, 2 is bad
prp(
  decision.tree
  , extra=1
  , varlen=0
  , faclen=0
  , main="Decision Tree for German Credit Data"
)
```

Then you will see something like below:



Decision Tree for German Credit Data

So I looked up what these A13, A14, A32, A33, A34, A64 and A65 mean:

- A13: Checking account balance >= 200 DM or salary assignments for at least 1 year
- A14: No checking account
- A32: Existing credits paid back duly till now
- A33: Delay in paying off in the past
- A34: Critical account / other credits existing (not at this bank)
- A64: Savings account balance >= 1000 DM
- A65: Unknown/ no savings account.

Interpreting the tree:

1. If a customer has no marginal amount of money in his checking account or does not have a checking account at all, then he is likely to be in good standing (1 means good, 2 means bad)
2. If a customer has only a small amount of money in his checking account, the loan duration is greater than or equal to 22, and also has only insignificant amount of fund in his savings account, he is more than likely to default. Does it make sense in your own terms? (By the way, the data was still using the old currency Deutsche Mark; how old is this data?)
3. The second-deep, left-most node? Some customers in this node had delays in paying off in the past or credits in other banks but will be categorized as good in this decision tree.

2. (Optional) Predict using the machine learning credit model

What's the use of a predictive model if we don't predict? Let us make some predictions using the credit model we just created:

```
new.data <- list(
  Status.of.existing.checking.account='A11'
  , Duration.in.month=20
  , Credit.history='A32'
```

```
  , Savings.account.bonds='A32'
)
predict(decision.tree, new.data)

#           1         2
# 1 0.6942446 0.3057554
```

3. Save it

Let us save the model to our hard disk so we can use it in our RESTful API:

```
save(decision.tree, file='decision_Tree_for_german_credit_data.RData')
```

4. Use the R package Plumber to create a RESTful API

So now is time to make a RESTful API for our credit model so that all German banks can use it (apology our future clients, Euro is not accepted). We will use Plumber, an open source package in R that provides easy ways to create a RESTful API for programs written in R. If you are familiar with Python or Ruby, it is an equivalent of Flask and Sinatra.

Plumber

Following their examples, we can create a script like below to create a web API for our credit model:

```
library(jsonlite)
load("decision_Tree_for_german_credit_data.RData")

#* @post /predict
predict.default.rate <- function(
  Status.of.existing.checking.account
  , Duration.in.month
  , Credit.history
  , Savings.account.bonds
) {
  data <- list(
    Status.of.existing.checking.account=Status.of.existing.checking.account
    , Duration.in.month=Duration.in.month
    , Credit.history=Credit.history
    , Savings.account.bonds=Savings.account.bonds
  )
  prediction <- predict(decision.tree, data)
  return(list(default.probability=unbox(prediction[1, 2])))
}
```

Then save the file (I named it deploy_ml_credit_model.R) and run the following command in an R environment such as the Console window in RStudio.

```
# For the first time, you many need to install the plumber package by typing:
# install.packages("plumber")
  library(plumber)
r <- plumb("deploy_ml_credit_model.R")
r$run(port=8000)
```

Then boom you just made your first RESTful API running R machine learning model! It is running on your localhost on port 8000.

5. Test to make sure it works

Let us see it works by issuing the following command in a terminal:

```
curl -X POST -d '{"Status.of.existing.checking.account": "A11", "Duration.in.month": 24, "Credit.history": "A32", "S
```

You will get:

```
{"default.probability":0.6224}
```

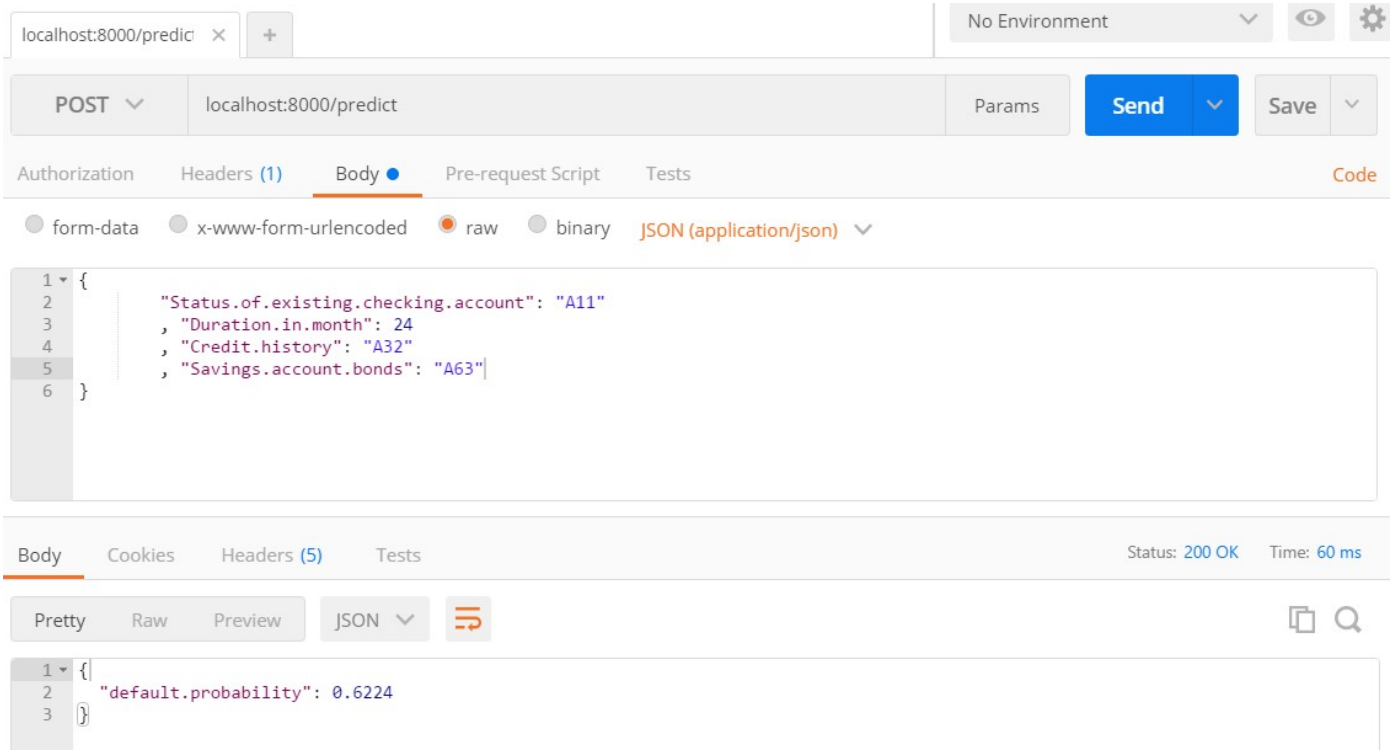
Hooray! Your decision tree model just responded!

Let us see if we can make requests to this API using another language like Python:

```
import requests
import json
response = requests.post(
    "localhost:8000"
    , headers={"Content-Type": "application/json"}
    , data=json.dumps({
        "Status.of.existing.checking.account": "A11"
    , "Duration.in.month": 24
    , "Credit.history": "A32"
    , , "Savings.account.bonds": "A63"
    })
)

print response.json()
# {u'default.probability': 0.6224}
```

Often, typing the curl command and writing scripts in Python can be cumbersome. So I found Postman as a better alternative to test RESTful APIs. Follow their instruction on installation. Then you can request to your API and get a response like below:



Some of you guys here might be wondering why we don't use Shiny. Though Shiny is a great tool for visualization, in my knowledge, as of March 1, 2017, it does not support the POST method, which is a very important HTTP method to send additional data to web applications. There are many discussions (#1, #2 and etc.) going on and perhaps it will be supported soon.

(By the way, our platform automates all the processes described in this and subsequent posts. If you would like to see a demo, please contact us.

3. Limitations

Great! We just deployed our credit model built in R. However, the deployment lacks the followings.

1. Input validation

Let us send some garbage input like this one:

```
{
    "Garbage": "Can you handle me?"
}
```

Wrong input

And you will see your web application complains:

```
{
  "error": [
    "500 - Internal server error"
  ],
  "message": [
    "argument \"Status.of.existing.checking.account\" is missing, with no default"
  ]
}
```

Server error

2. Monitoring / alerting

Did it send an email to you before it erred out?

3. Security

Thanks! Your competitors can benefit from your machine learning models too!

4. Scalability

This is probably the most difficult obstacle to solve. R by nature is a single-thread language.

How a single-threaded manager would manage employees

It means that when an order comes to a pizza store with four employees, the manager will ask only one employee to get the order, cook pizzas, deliver and receive payments because the manager just recently started her work and does not know how to manage resources otherwise. During that whole time the order is processed, the other three will be idle. (Image source: Some eCards)

When multiple requests come at a same time, R only handles one request at a time and those that are not being processed have to wait, causing significant latency.

No one likes slow pizza delivery
Because the manager uses only one employee even though there are four, if the employee is busy, the next orders have to wait and typically no one likes slow pizza delivery. (Image source: final gear)

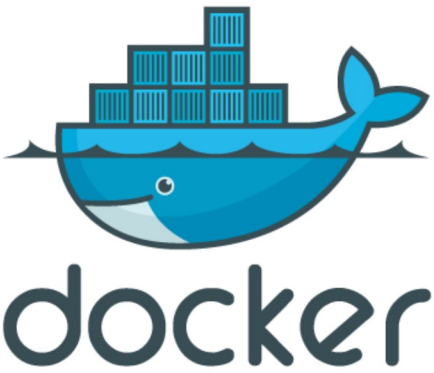
Generally, there are three ways to scale up your RESTful APIs. We will talk about these options in more detail in another post.

1. Multiple processes

It is difficult to do with R, but basically you use all four employees rather than one.

2. Docker

I had difficulty finding a good analogy here because Docker allows you to run multiple "Virtual" OS in one server and there is no such thing as a "Virtual" pizza store. How about this one? You just bring multiple containers to your pizza store front yard, decorate each like a pizza store and process orders as if multiple stores are processing. This blog "How to scale up machine learning credit model RESTful APIs using Docker" specifically talks about how to scale a machine learning model API using Docker.



Docker
Docker makes a container for both your application and also the OS the application is developed in so that when the container gets shipped, the application can run in the same environment regardless of the underlying, new infrastructure. Furthermore, you can run multiple docker containers simultaneously making your infrastructure more scalable.

3. Multiple servers

Just open more stores to handle larger volume.

For maximum throughput, you can consider having multiple servers, each of which run multiple Docker containers, each of which run multiple processes (whew). For more detailed explanations, you can find this blog on how to scale up credit model APIs using AWS.

In this blog post, we learned how to create RESTful APIs for R models. The next three posts will talk about how to scale up your RESTful APIs for larger audience. We've also streamlined the process of creating an API and you can find relevant information below:

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: