

José Hisse [home](#)

Transformando modelo preditivo em produto com R

2020-08-13

O objetivo deste artigo é transformar em produto um modelo desenvolvido com a linguagem R, no caso, criando uma API Rest preditiva. Para isso vamos utilizar uma base contendo dados de pessoas diabéticas, cuja as variáveis de entradas serão pré determinadas e teremos como output uma probabilidade daquela pessoa ter ou não diabete. Vamos utilizar também a plataforma Docker para treinarmos nosso modelo com o RStudio e transformamos em container nossa API.

Base de dados PIMA

Vamos utilizar a Pima Indians Diabetes Database, assim como usamos neste [artigo abordando o deploy de modelos em Python](#). A base de dados do PIMA contém dados de pessoas do sexo feminino acima de 21 anos diabéticas ou não.

- **pregnant**: quantidade de vezes que esteve grávida;
- **glucose**: nível de glicose após 2 horas do teste de intolerância à glicose;
- **pressure**: Pressão arterial diastólica (mmHg);
- **triceps**: Espessura da dobra de pele do tríceps (mm);
- **insulin**: Quantidade de insulina após 2 horas do teste de intolerância à glicose;
- **mass**: Índice de massa corporal ($IMC = \text{peso(kg)} / (\text{altura(m)} * \text{altura(m)})$);
- **pedigree**: Função que retorna um score com base no histórico familiar;
- **age**: Idade (anos);
- **diabetes**: Indicativo se a pessoa é diabética (pos/neg).

As 8 primeiras variáveis serão usadas como entrada do nosso modelo preditivo e a última será nossa saída.

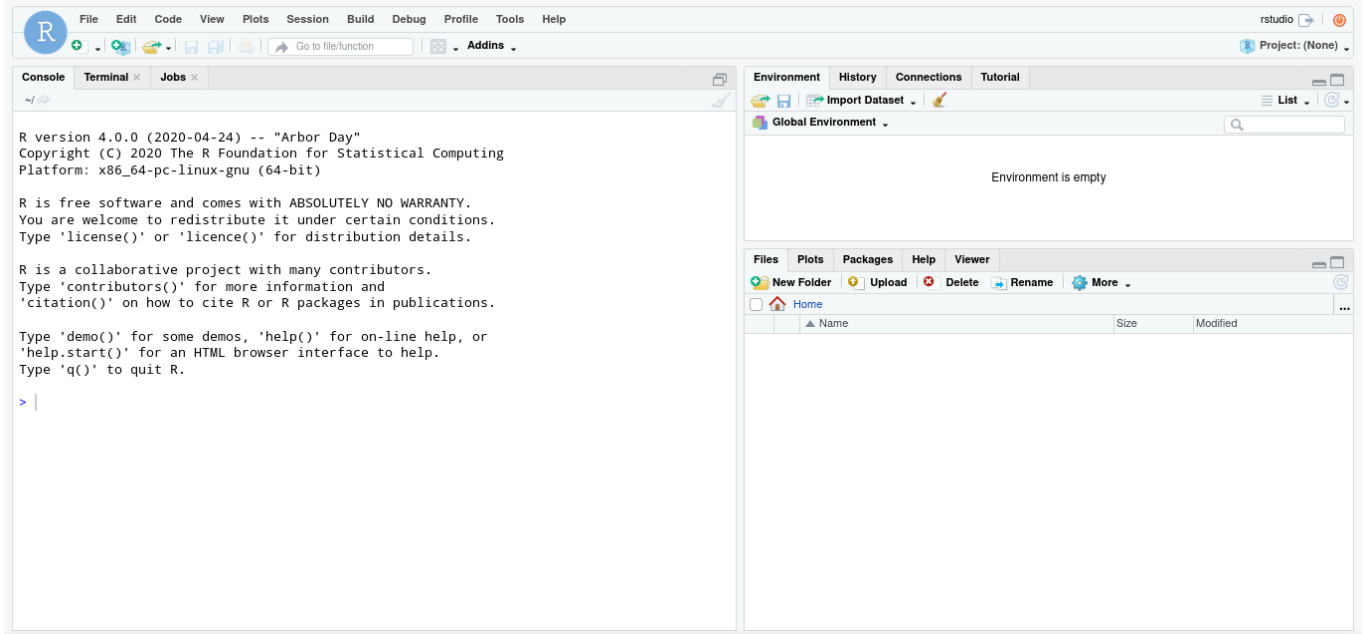
Iniciando com RStudio

Para desenvolvermos nosso modelo iremos utilizar a famosa IDE para a linguagem de programação R chamada RStudio. O RStudio é uma interface de desenvolvimento baseada na web, ou seja, podemos acessá-la através do nosso navegador.

Como o foco do artigo é utilizarmos o Docker para facilitar a padronização do nosso ambiente de desenvolvimento, vamos começar por iniciar uma imagem básica já contendo nossa IDE.

```
docker run --rm -d -p 8787:8787 --name rstudio-pima -e DISABLE_AUTH=true rocker/rstudio:4.0.
```

Ao acessar o navegador no endereço `localhost:8787` teremos acesso à interface do RStudio.



A imagem padrão do rstudio contém os pacotes básicos para executar nossos scripts em R, porém como iremos trabalhar com modelos de machine learning será necessário instalar algumas bibliotecas.

É de boa prática criarmos uma imagem personalizada quando estamos utilizando o Docker para a padronização de ambientes. Por isso não iremos instalar direto na interface do RStudio que vimos anteriormente, vamos criar uma definição de imagem onde iremos executar um comando de instalação dos pacotes R que serão necessários.

Vamos parar o RStudio que iniciamos para evitarmos conflito de portas em nosso host:

```
docker stop rstudio-pima
```

Personalizando uma imagem R

Como vimos mais acima, a imagem do RStudio que vamos utilizar é a [rocker/rstudio:4.0.0](https://github.com/rstudio/rstudio-docker), então vamos começar criando um diretório modelo-r para ser a nossa pasta de trabalho e dentro desta pasta vamos criar o arquivo de definição de imagem chamado `Dockerfile`:

```
FROM rocker/rstudio:4.0.0
```

O arquivo de definição acima apenas cria um novo layer a partir da imagem base *rocker/rstudio:4.0.0*, porém precisamos instalar as bibliotecas *mlbench*, *caret* e suas dependências. A imagem base que estamos utilizando disponibiliza um script R que nos ajuda nessa tarefa, o nome dele é [install2.r](#). Então vamos ao nosso Dockerfile:

```
FROM rocker/rstudio:4.0.0

RUN apt-get update \
    && apt-get install -y --no-install-recommends \
        libxml2-dev \
        libz-dev \
    && rm -rf /var/lib/apt/lists/*

RUN install2.r -s -d TRUE --error \
    mlbench \
    caret \
    plumber
```

Vamos construir nossa imagem definindo o nome dela como *minhaimagemr*:

```
docker build -t minhaimagemr .
```

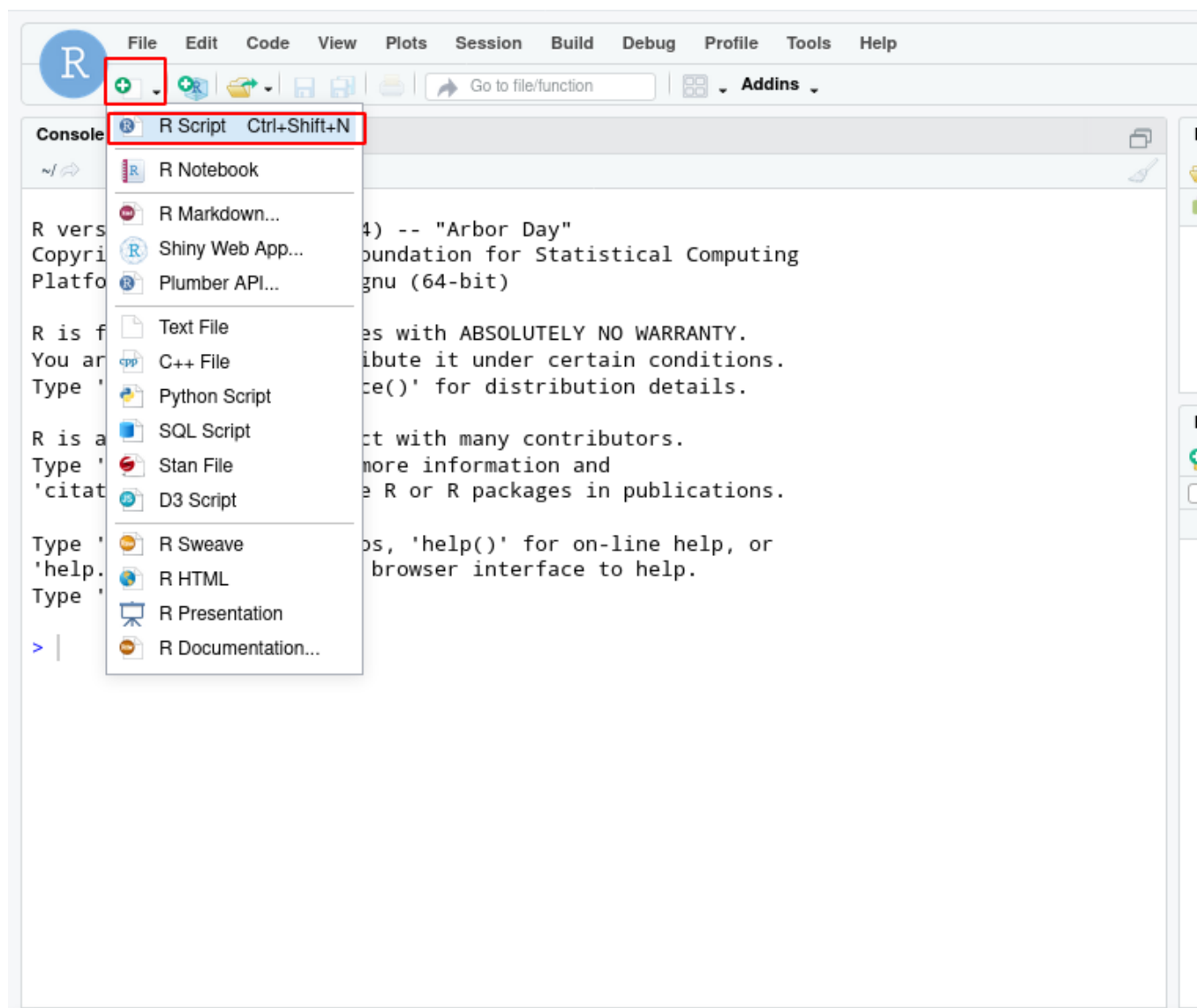
E após isso vamos executar nosso container novamente, só que agora com as bibliotecas e suas dependências já instaladas:

```
docker run --rm -d -p 8787:8787 --name rstudio-pima -e DISABLE_AUTH=true minhaimagemr
```

Observação: caso a senha seja requisitada, basta colocar “rstudio” no campo username e password.

Desenvolvendo nosso modelo

Verificando o funcionamento de nossa imagem em `localhost:8787`, vamos criar um novo arquivo no rstudio e carregar nosso script contendo o treinamento do modelo.



A descrição dos comandos antecede os mesmos no código abaixo.

```
# Biblioteca contendo o algoritmo
library(mlbench)

# Base de dados que iremos utilizar para treinar o modelo
data('PimaIndiansDiabetes')

# Verificando os dados que temos no dataset
head(PimaIndiansDiabetes,10)

# Dimensão de nosso dataset, número de linhas e colunas
dim(PimaIndiansDiabetes)

# Biblioteca necessária para particionar o modelo em conjunto de
# treinamento e conjunto de teste
library(caret)

# Ajusta o randomizador para reproduzirmos os mesmos resultados
set.seed(12345)

# Particiona o dataset em dois conjuntos, 80% para treinamento e 20% para teste
trainIndex <- createDataPartition(PimaIndiansDiabetes$diabetes,p=0.8,list=FALSE)
```

```
# Separa o dataset de treinamento do dataset de teste em objetos distintos
trainset <- PimaIndiansDiabetes[trainIndex,] # 80% dos dados para treinamento
testset <- PimaIndiansDiabetes[-trainIndex,] # 20% dos dados para teste

# Verifica as dimensões dos datasets
dim(trainset)
dim(testset)

#
typeColNum <- grep('diabetes',names(PimaIndiansDiabetes))
typeColNum

# Treinar o modelo com o conjunto de treinamento
glm_model <- glm(diabetes~.,data=trainset, family=binomial)

# Verificar os coeficientes que o modelo gerou
summary(glm_model)

# Testar o modelo com o conjunto de teste
glm_prob <- predict.glm(glm_model,testset[, -typeColNum],type='response')

head(glm_prob, 10)

#
contrasts(PimaIndiansDiabetes$diabetes)

# Efetua previsões
glm_predict <- rep('neg',nrow(testset))
glm_predict[glm_prob>.5] <- 'pos'

head(glm_predict, 10)

# Matrix de colisão
table(pred=glm_predict,true=testset$diabetes)

# Acurácia do modelo
mean(glm_predict==testset$diabetes)

# Salvar o modelo para uso da API
saveRDS(glm_model, "./glm_model.rds")
```

Vale destacar que ao final do nosso script iremos salvar o modelo treinando em um objeto do R. Sendo assim, poderemos efetuar o download do mesmo e utilizarmos posteriormente em outro script.

Criando nossa api em R

Para disponibilizarmos uma interface para nosso modelo preditivo vamos criar uma API. Essa será construída com a ajuda de uma [biblioteca R chamada Plumber](#). A Plumber irá permitir a construção de nossa API sem que tenhamos que alterar o código R existente, precisamos apenas adicionar decorators a nossa função de predição. ([Para entender mais](#)

sobre [decorators](#).)

Primeiro devemos criar um novo arquivo no RStudio e carregar nosso modelo salvo anteriormente. Após o o modelo treinado estar carregado em um objeto do R, vamos definir nossa API, usando os decorators na função de previsão. Os decorators irão expandir nossa função à transformando em uma espécie de núcleo do nosso endpoint da API.

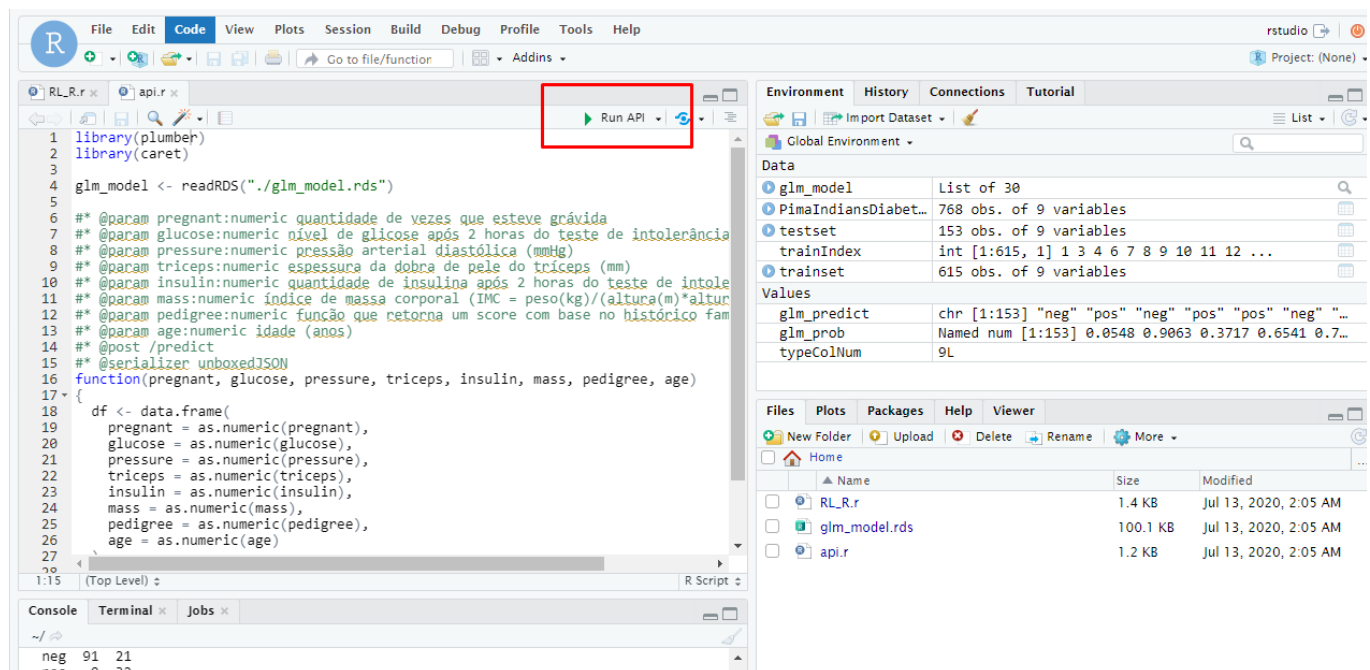
Além dos elementos descritivos do endpoint preditivo, vamos dizer que aquela função será invocada pelo [método post do protocolo http](#) pelo decorator `## @post /predict` e que o [serializador unboxedJSON](#), descrito com `## @serializer unboxedJSON` será o método que converte o retorno da função para um formato de saída desejado, em nosso caso, em json.

```
# Importando a biblioteca plumber
library(plumber)

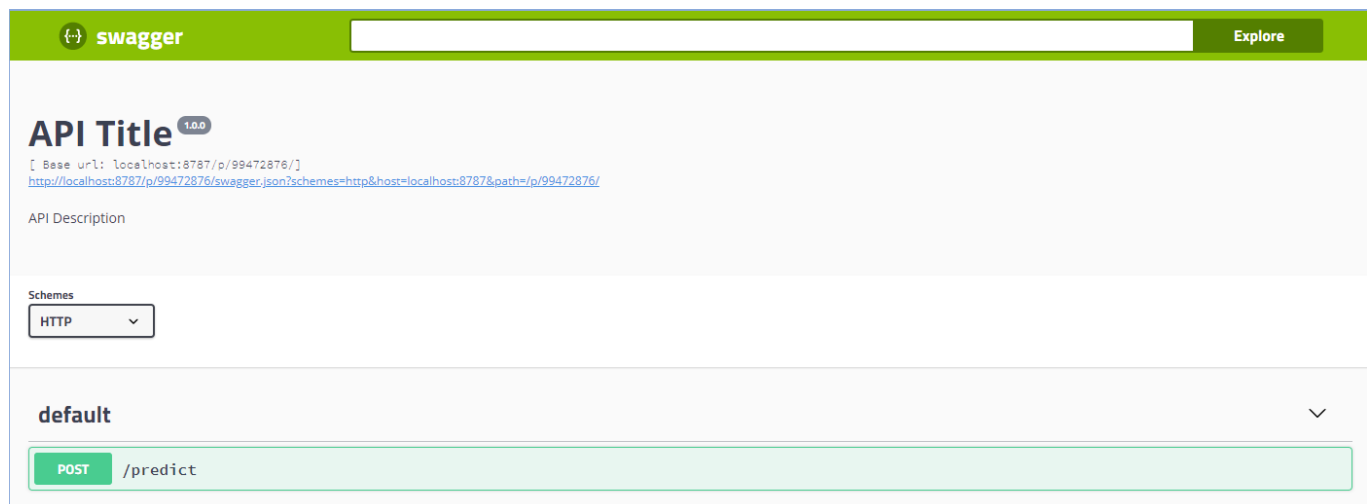
# Carregamento do modelo para um objeto
glm_model <- readRDS("./glm_model.rds")

## @param pregnant:numeric quantidade de vezes que esteve grávida
## @param glucose:numeric nível de glicose após 2 horas do teste de intolerância à glicose
## @param pressure:numeric pressão arterial diastólica (mmHg)
## @param triceps:numeric espessura da dobra de pele do tríceps (mm)
## @param insulin:numeric quantidade de insulina após 2 horas do teste de intolerância à gli
## @param mass:numeric índice de massa corporal (IMC = peso(kg)/(altura(m)*altura(m)))
## @param pedigree:numeric função que retorna um score com base no histórico familiar
## @param age:numeric idade (anos)
## @post /predict
## @serializer unboxedJSON
function(pregnant, glucose, pressure, triceps, insulin, mass, pedigree, age)
{
  df <- data.frame(
    pregnant = as.numeric(pregnant),
    glucose = as.numeric(glucose),
    pressure = as.numeric(pressure),
    triceps = as.numeric(triceps),
    insulin = as.numeric(insulin),
    mass = as.numeric(mass),
    pedigree = as.numeric(pedigree),
    age = as.numeric(age)
  )
  output <- list(prob = predict.glm(glm_model, df, type='response'))
  return(output)
}
```


Podemos executar nossa API no próprio RStudio, assim teremos um ambiente de testes antes de empacotarmos em uma imagem Docker. Para isso vamos clicar no símbolo de play verde no canto superior direito do RStudio e uma nova janela irá se abrir.



Na janela que foi aberta quando iniciamos a API, irá aparecer uma interface do Swagger. O Swagger é um utilitário que permite documentar nossa aplicação de forma amigável para o usuário. Graças ao pacote plumber teremos essa interface já implementada.



Na interface principal do Swagger podemos inserir valores de testes e verificarmos o resultado, como no exemplo a seguir.

 swagger

Explore

API Title ^{1.0.0}

[Base url: localhost:8787/p/99472876/]
<http://localhost:8787/p/99472876/swagger.json?schemes=http&host=localhost:8787&path=/p/99472876/>

API Description

Schemes

HTTP

default

POST /predict

Parameters

Cancel

Name	Description
age number (query)	idade (anos) <input type="text" value="50"/>
pedigree number (query)	função que retorna um score com base no histórico familiar <input type="text" value="0.627"/>
mass number (query)	índice de massa corporal (IMC = peso(kg)/(altura(m)*altura(m))) <input type="text" value="33.6"/>
insulin number (query)	quantidade de insulina após 2 horas do teste de intolerância à glicose <input type="text" value="0"/>
triceps number (query)	espessura da dobra de pele do tríceps (mm) <input type="text" value="35"/>
pressure number (query)	pressão arterial diastólica (mmHg) <input type="text" value="72"/>
glucose number (query)	nível de glicose após 2 horas do teste de intolerância à glicose <input type="text" value="148"/>
pregnant number (query)	quantidade de vezes que esteve grávida <input type="text" value="6"/>

Execute

Clear

Responses

Response content type application/json

Curl

```
curl -X POST "http://localhost:8787/p/99472876/predict?age=50&pedigree=0.627&mass=33.6&insulin=0&triceps=35&pressure=72&glucose=148&pregnant=6" -H "accept: application/json"
```

Server response

Code	Details
200 <i>Undocumented</i>	<div>Response body</div> <pre>{ "prob": 0.6965 }</pre> <div>Response headers</div> <pre>connection: close content-length: 15 content-type: application/json date: Mon, 13 Jul 2020 05:55:00 GMT, Mon, 13 Jul 2020 05:55:00 AM GMT server: RStudio x-content-type-options: nosniff</pre>

Responses	
Code	Description
default	Default response.

Empacotando a API preditiva

Neste ponto vamos ter como objetivo empacotar nossa API preditiva em container, ou seja, vamos criar uma receita para que nossa API seja facilmente replicada.

Em nossa abordagem vamos inserir o modelo treinado no container junto com código da API, diferente da abordagem adotada no [artigo em que utilizamos o framework serverless e o S3](/blog/api-modelos-machine-learning).

Vamos a nossa estrutura de diretórios:

- raiz/
- Dockerfile
- api.r
- glm_model.rds

Primeiro vamos copiar o modelo já treinado, *glm_model.rds*, e o código da API, *api.r* para dentro do diretório raiz. A seguir vamos criar o arquivo chamado *Dockerfile* com o seguinte conteúdo:

```
FROM r-base:4.0.0

# Instalando dependências
RUN apt-get update \
    && apt-get install -y --no-install-recommends \
    libxml2-dev \
    libz-dev \
    git-core \
    libssl-dev \
    libcurl4-gnutls-dev \
    && rm -rf /var/lib/apt/lists/*

# Instalando o Plumber com o install2.r
RUN install2.r -s -d TRUE --error plumber

# Instalando o caret com o install2.r
RUN install2.r -s -d TRUE --error caret

# Setando o usuário e grupo
USER 1000:1000

# Definindo diretório de trabalho
WORKDIR /app

# Copiando arquivos
COPY api.r .
```

```
COPY glm_model.rds .
```

```
EXPOSE 8080
```

```
# Definindo comando a ser executado, ativando o endpoint do swagger
```

```
ENTRYPOINT ["R", "-e", "pr <- plumber::plumb('/app/api.r'); pr$run(host='0.0.0.0', port=8080)"]
```

A definição do Dockerfile acima habilita a portabilidade de nossa aplicação em container.

Para efetuarmos o build de nossa API vamos executar o seguinte comando no terminal:

```
docker build -t api-r .
```

Obs.: Não esqueça do ponto ao final do comando.

Após o build já podemos executar nossa aplicação com o modelo treinado. No terminal digite:

```
docker run -p 8080:8080 api-r
```

Agora podemos acessar a interface do swagger no browser, `localhost:8080/__swagger__` como imagem abaixo.

API Title 1.0.0

[Base url: localhost:8080/]
<http://localhost:8080/swagger.json?schemes=http&host=localhost:8080&path=/>

API Description

Schemes
HTTP

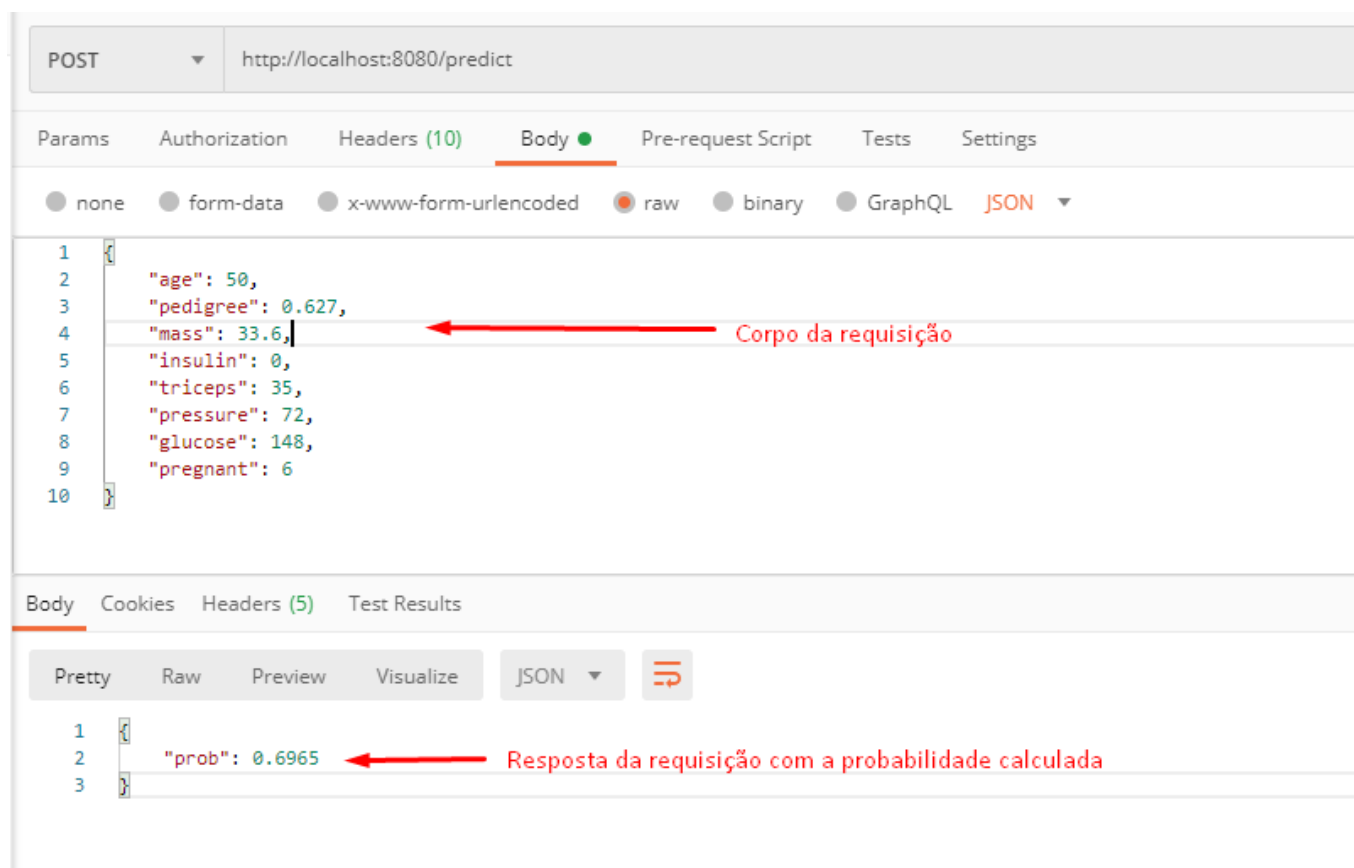
default

POST /predict

Parameters Try it out

Name	Description
age number (query)	idade (anos)
pedigree number (query)	função que retorna um score com base no histórico familiar
mass number (query)	índice de massa corporal (IMC = peso(kg)/(altura(m)*altura(m)))
insulin number	quantidade de insulina após 2 horas do teste de intolerância à glicose

Ou podemos fazer uma requisição HTTP como na imagem abaixo.



Conclusão

Este artigo buscou demonstrar ao leitor uma maneira de produtizar um modelo preditivo utilizando a linguagem R, tão comum entre os data scientists. Todos os códigos utilizados aqui podem ser encontrados neste [repositório do GitHub](#).