Practical Machine Learning Project Hugo PERIER

Hugo PERIER 13697711

9/24/2019

```r
knitr::opts_chunk$set(echo = TRUE)
install.packages('rstudioapi', repos = "http://cran.us.r-project.org")
install.packages("Metrics", repos = "http://cran.us.r-project.org")
install.packages('smooth', repos = "http://cran.us.r-project.org")
install.packages("kableExtra", repos = "http://cran.us.r-project.org")
library(knitr)
library(kableExtra) # to print table in the report
library(smooth)
library(Metrics) # for MSE
library(rstudioapi) # to set the working directory in the file location
set.seed(13697711)

# STEP0: pre-processing datasets
# ----
#current_path <- getActiveDocumentContext()$path
#setwd(dirname(current_path ))
d <- read.csv("CBA_history_data.csv",encoding = "UTF-8")
d <-d[order(as.Date(d$X.U.FEFF.date, format = "%d/%m/%Y")),]
d<-d[-1,]
colnames(d) <- c("date","closeprices","return")
ds <- d$return
training_set = ds[c(1:299)]
test_set  = ds[c(300:399)]
```

GitHub file : https://github.com/Hugo-Perr/ML_A2.git

Practical Machine Learning Project Hugo PERIER

I decided to build a neural network for forecasting financial time series. Then, I chose real data on the following website :https://datanalysis-morningstar-com-au.ezproxy.lib.uts.edu.au/

The aim of this project is to predict the daily return of the daily share price of Commonwealth Bank (CBA). I based all this algorithm on a research article, which I you can consult here: https://pdfs.semanticscholar.org/87d3/eb3174f93abb1822343798084a50c3d18bff.pdf.

NB: I explain what I have understand from this research article in the following parts, and I used the point "." as a multiplicative in all formuleas.

Data set used :

	date	closeprices	return
399	02/02/2018	80.79	0.0097488
398	05/02/2018	79.80	-0.0122540
397	06/02/2018	77.40	-0.0300752
396	07/02/2018	76.79	-0.0078811
395	08/02/2018	76.51	-0.0036463
394	09/02/2018	76.25	-0.0033982

How I created the multi-layer perceptron

The multi-layer perceptron is made of several simple perceptrons. Several architectures are possible. To keep things simple, I suppose (I chose) that, for n inputs, we have p hidden layers of n nodes each, and then an output layer with a single node.

Values of each perceptron

For the activation function (or transfer function), I will use the most widespread one, that is the sigmoid function: f(x)=1/(1+exp(-x)) and for the derivative of this function :f'(x)=f(x)(1-f(x)).

For the ith perceptron of the layer k if there are only 2 nodes on the previous layer the value of the perceptron using the transfert function is : zi,k = w1,i,(k-1).z1,(k-1) + w2,i,(k-1).z2,(k-1). But for the example with the code I will chose 3 nodes per layer.

Weights of each perceptron

Between each pair of adjacent layers, n² weights have to be estimated. I decided to store these weights in an array (1 to n, 1 to n, 1 to p), in which wi,j,k the weight linking the output i of layer (k-1), noted zi,k,to the node j of layer k. It addition to this array, there is an one-dimensionnal array containing the weights between the last hidden layer and the output node, noted wi,1,p+1.

Important : all initial weights are chosen randomly.

The update of the weights of the last layer is made comparing the predicted value to the real value (for the training test). Then, the weight is corrected in the following way (which incorporates the gradient in the case of a sigmoid transfer function): wi <- wi - s x Zi x e With e the error : e = (y-s)y(1-y) the derivative of the error(or it is the sensitivity of the error, at the level of the output node). wi is the weight, of the neuron i, xi is its input, y the output, s the correct value, and s is what the bibliography call the learning rate (i chose s=0.5 as a starting value) it impact the speed/way of the update of the weights at each line during the learning steps.

How I update the weights at each steps

I use a backpropagation to update the weights of the other layers.

Creating the matrix e begin with the layers the closer to the output, and go backward: For the node i of the hidden layer k (with a loop for k decreasing from p to 1), the sensitivity of the output error to its input is now ei,k = f'(sum of [wl,i,k.zl,k] for l=1 to n).(sum of [wi,j,(k+1).ej,(k+1)]).

After having calculated the sensitivity for each node of each layer, update all the weights with : wi,j,k = wi,j,k - s.zi,k.ei,k

Learning phase

```r
# STEP1: MDP's parameters
# ----
n = 3  # number of nodes per hidden layer and number of inputs for the first nodes of the MDP
p = 2 # number of hidden layers
m = min(ds)
M = max(ds)
ds.w <- array(dim=c(n,n,p)) # array for weights of each nodes
ds.z <- array(dim=c(n,(p+1))) # array for values of each nodes
ds.e <- array(dim=c(n,(p+1))) # array for values of sensitivity of error
ds.wf <- array(dim=n)
learning_rate = 0.5

f <- function(x) # transfert function, the most widespread one ie the sigmoide
{
  f <- 1/(1+exp(-x))
}

# initializing weights:
for (i in 1:n)
{
  for (j in 1:n )
  {
    for(k in 1:(p))
    {
      ds.w[i,j,k] = runif(1)
      ds.wf[i]=runif(1)
    }
  }
}
# ----

# STEP2: Calculate Z
training_line <- 4 # must be up to 3 to have acces to the interest a d-1,d-2 and day-3
for (training_line in 4:length(training_set))
{

# ----
# following step repeated for all the training set
for (i in 1:n) { ds.z[i,1]=training_set[training_line-i] } # n input at the very biginning of the MDP

for (k in 2:(p+1))
{
    for(i in 1:n)
    {
      s = 0
      for (nb_node_before in 1:n)
      {
        s = s + ds.w[nb_node_before,i,(k-1)]*ds.z[nb_node_before,(k-1)]
      }
      ds.z[i,k] = f(s)
    }
}
# ----

# STEP3: Compute the final estimation
# ----
s = 0
for (i in 1:n)
{
  s = s + ds.wf[i]*ds.z[i,p]
}
y=f(s)*(M-m)+m # compute and scale the estimated return
# ----

# STEP4: Sensitivity of the error & back propagation
# ----
e = (y-training_set[training_line])*y*(1-y)
# particular cases : the last hidden layer errors k=p+1, only 1 weight per node
s = 0
for (i in 1:n)
{
  s = s + ds.wf[i]*ds.z[i,(p+1)] # first sum used to calculate de propagation of the error
}
for (i in 1:n)
{
  ds.e[i,(p+1)] = f(s)*(1-f(s))*ds.wf[i]*e
}



for (i in 1:n) # backpropagation of the error
{
  s=0
  for (l in 1:n) { s = s + ds.w[l,i,p]*ds.z[l,p] }
  ds.e[i,p] = f(s)*(1-f(s))*ds.wf[i]*e
}

# then for other nodes:
for (k in (p-1):1)
{
  for (i in 1:n)
  {
    s = 0
    sj=0
    for (l in 1:n)
    {
      s = s + ds.w[l,i,k]*ds.z[l,k]
      sj = sj + ds.w[i,l,k]*ds.e[l,k+1]
    }
    ds.e[i,k] = f(s)*(1-f(s))*sj
  }
}
# ----

# STEP5: Update W
# ----
for (i in 1:n)
{
  for (j in 1:n)
  {
    for (k in 1:p)
    {
      ds.w[i,j,k] <- ds.w[i,j,k] - learning_rate*ds.z[i,k]*ds.e[i,k]
    }
  }
}
for (i in 1:n)
{
  ds.wf[i]=ds.wf[i] - learning_rate*ds.z[i,(p+1)]*ds.e[i,(p+1)]
}
# ----

}
```

Now let's see the matrix weights :

```r
kable(ds.w)
```

V1	V2	V3	V4	V5	V6
0.8035982	0.6570317	0.2552454	0.1120703	0.2788933	0.4155853
0.9021069	0.1385585	0.5651541	0.7682733	0.8984571	0.6577997
0.1855141	0.8425236	0.8575772	0.3825001	0.7303111	0.0040999

```r
kable(ds.wf)
```

x
0.0266083
0.1360001
0.5959309

Matrix nodes values Z:

```r
kable(ds.z)
```

-0.0047733	0.4985421	0.6525814
-0.0022412	0.4991678	0.7215689
0.0001401	0.4994088	0.6312537

Matrix error E:

```r
kable(ds.e)
```

8.40e-06	3.50e-06	3.60e-06
1.26e-05	1.57e-05	1.84e-05
2.06e-05	7.99e-05	8.08e-05

Testing phase

```r
estimated_return = c()
for (testing_line in 4:length(test_set))
{
  for (i in 1:n) # initialize inputs for the current line
  {
    ds.z[i,1]=training_set[testing_line-i]
  }

  for (k in 2:(p+1)) # compute Z matrix
  {
    for(i in 1:n)
    {
      s = 0
      for (nb_node_before in 1:n)
      {
        s = s + ds.w[nb_node_before,i,(k-1)]*ds.z[nb_node_before,(k-1)]
      }
      ds.z[i,k] = f(s)
    }
  }

  s = 0
  for (i in 1:n)
  {
    s = s + ds.wf[i]*ds.z[i,p]
  }
  y=f(s)*(M-m)+m # compute and scale the estimated return
  estimated_return = c(estimated_return,y) # saving the estimate return
}

result_test = data.frame(test_set[-c(1:3)],estimated_return)
colnames(result_test) <- c("real_returns","estimated_returns")
```

Then, we obtain the following table:

```r
kable(head(result_test))
```

real_returns	estimated_returns
0.0194990	0.0227451
0.0078180	0.0226573
0.0069262	0.0226472
0.0082542	0.0227402
0.0016373	0.0227543
0.0107615	0.0227682

Evaluation of this test

MSE:

```
## [1] 0
```

There is an error to print the mean squarred error in markdown, but on my test : MSE = 0.001283608 According to the small size of the estimated values, I concur that this value is too large. In addition. I also compute the Mean absolute percentage error (MAPE) : MAPE = 0.9795205 Even if it's less than 1%, I think there is an error with the backpropagation of the error between the nodes of the multi layer perceptron. I tried to change the way I compute the E matrix (specially on the last hidden layer to be sure that the latter takes in concideration the error on the final value estimated). I also tried to change the value of n,p and the learning parameter … in vain.

The vast majority of the work I have done on this project is related to the mathematical understanding of this prediction method. This explains why I remain surprised by my results, I don't think I made any mistakes in the programming.

Discuss Reflections

The backpropagation explained above can be seen as a standard learning rule. I saw on internet that it can be ameliorated with the following methods:

- The use of Momentum: the standard learning rule is mitigated with a stationary rule. A new parameter determines to which extent the weights should be updated and to which extent the previous change of weight should be used again
- The use of a non-constant learning rates.

Quality of the data used

Concerning the quality of the data, the data I have used are free of rights and accessible on several sites. I have therefore briefly checked the accuracy of the data (from https://datanalysis-morningstar-com-au.ezproxy.lib.uts.edu.au/) on the following website: https://www.asx.com.au/asx/share-price-research/company/CBA So I can say that the data sets are strictly identical.

Social/ethical aspect of the proposed technique

If we think ethically, developing a machine learning algorithm to predict in advance the course of an action would have disastrous consequences. First, it would destroy chance, or rather stock market uncertainty for the person who owns it. Let us first admit that a small portion of the population has access to this algorithm. These would be the mathematicians and programmers behind this discovery. It would only take one of them to have less ethics than the others for the code to spread. Since large stock market profits require handling a large amount of money, a person in possession of this algorithm will have to raise this amount of money or partner with someone who is richerthan him. It is from this moment that the important consequences occur. Predicting stock market prices on the use of public historical data can generate significant profits in the first instance. But unfortunately, if the use of such an algorithm is used too frequently, on too large an amount, or by too many players, I think it could simply kill the stock market. Either the algorithm will no longer be as efficient because of the upheaval in historical data. Either, financial crises could be created because of a very hazardous distribution of wealth, favourable only to those who use the algorithm.

So, if the prospect of enormous wealth is interesting to anyone creating this type of machine learning algorithm, we must all hope that this person can reflect on the consequences of this discovery. Knowledge is an important power, but not all knowledge is necessarily good for sharing.