Today in [Machine Learning Explained](http://enhancedatascience.com/tag/machine-learning-explained/), we will tackle a central (yet under-looked) aspect of Machine Learning: **vectorization**. Let’s say you want to compute the sum of the values of an array. The naive way to do so is to loop over the elements and to sequentially sum them. This naive way is slow and tends to get even slower with large amounts of data and large data structures.  
With vectorization these operations can be seen as matrix operations which are often more efficient than standard loops. Vectorized versions of algorithm are several orders of magnitudes faster and are easier to understand from a mathematical perspective.

**A basic exemple of vectorization**

**Preliminary exemple – Python**

Let’s compare the naive way and the vectorized way of computing the sum of the elements of an array. To do so, we will create a large (100,000 elements) Numpy array and compute the sum of its element 1,000 times with each algorithm. The overall computation time will then be compared.

import numpy as np

import time

W=np.random.normal(0,1,100000)

n\_rep=1000

The naive way to compute the sum iterates over all the elements of the array and stores the sum:

start\_time = time.time()

for i in range(n\_rep):

loop\_res=0

for elt in W:

loop\_res+=elt

time\_loop = time.time() - start\_time

:

start\_time = time.time()

for i in range(n\_rep):

one\_dot=np.ones(W.shape)

vect\_res=one\_dot.T.dot(W)

time\_vect = time.time() - start\_time

Finally, we can check that both methods yield the same results and compare their runtime. The vectorized version run approximately 100 to 200 times faster than the naive loop.

print(np.abs(vect\_res-loop\_res)<10e-10)

print(time\_loop/time\_vect)

Note: The same results can be obtained with *np.sum*.The numpy version has a very similar runtime to our vectorized version. Numpy being very optimized, this show that our vectorized sum is reasonably fast.

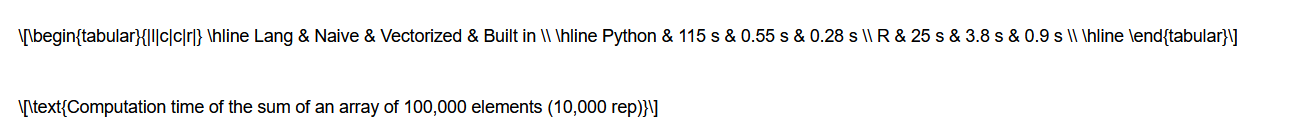
start\_time = time.time()

for i in range(n\_rep):

vect\_res=np.sum(W)

time\_vect\_np = time.time() - start\_time

**Preliminary exemple – R**



The previous experiments can be replicated in R:

##Creation of the vector

W=matrix(rnorm(100000))

n\_rep=10000

#Naive way:

library(tictoc)

tic('Naive computation')

for (rep in 1:n\_rep)

{

res\_loop=0

for (w\_i in W)

res\_loop=w\_i+res\_loop

}

toc()

tic('Vectorized computation')

# vectorized way

for (rep in 1:n\_rep)

{

ones=rep(1,length(W))

res\_vect= crossprod(ones,W)

}

toc()

tic('built-in computation')

# built-in way

for (rep in 1:n\_rep)

{

res\_built\_in= sum(W)

}

toc()

In R, the vectorized version is only an order of magnitude faster than the naive way. The built-in way achieves the best performances and is an order of magnitude faster than our vectorized way.

**Preliminary exemple – Results**

Vectorization divides the computation times by several order of magnitudes and the difference with loops increase with the size of the data. Hence, if you want to deal with large amount of data, rewriting the algorithm as matrix operations may lead to important performances gains.

**Why vectorization is (often) faster**

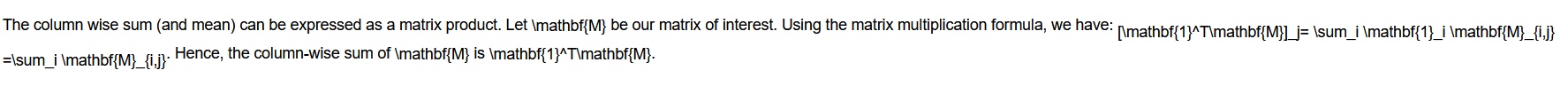
* R and Python are **interpreted language**, this means that your instructions are analyzed and interpreted at each execution. Since they are not statically typed, the loops have to assess the type of the operands at each iteration which leads to a computational overhead.
* R and Python linear algebra relies on **optimized back-end for matrix operations and linear algebra**. These back-end are written in C/C++ and can process loops efficiently. Furthermore, while loops are sequential, these back-end can run operations in parallel which improves the computation speed on modern CPU.

**Note 1**: Though vectorization is often faster, it requires to allocate the memory of the array. If your amount of RAM is limited or if the amount of data is large, loops may be required.  
**Note 2**: When you deal with large arrays or computationally intensive algebra ( like inversion, diagonalization, eigenvalues computations, ….) computations on GPU are even order of magnitudes faster than on CPU. To write efficient GPU code, the code needs to be composed of matrix operations. Hence, having vectorized code maked it easier to translate CPU code to GPU (or tensor-based frameworks).

**A small zoo of matrix operations**

The goal of this part is to show some basic matrix operations/vectorization and to end on a more complex example to show the thought process which underlies vectorization.

**Column-wise matrix sum**

.

**Python code:**

def colWiseSum(W):

ones=np.ones((W.shape[0],1))

return ones.T.dot(W)

**R code:**

colWiseSum=function(W)

{

ones=rep(1,nrow(W))

t(W)%\*%ones

}

**Row-wise matrix sum**

.

**Python code:**

def rowWiseSum(W):

ones=np.ones((W.shape[1],1))

return W.dot(ones)

**R code:**

rowWiseSum=function(W)

{

ones=rep(1,ncol(W))

W%\*%ones

}

**Matrix sum**

.

**Python code:**

def matSum(W):

rhs\_ones=np.ones((W.shape[1],1))

lhs\_ones=np.ones((W.shape[0],1))

return lhs\_ones.T.dot(W).dot(rhs\_ones)

**R code:**

matSum=function(W)

{

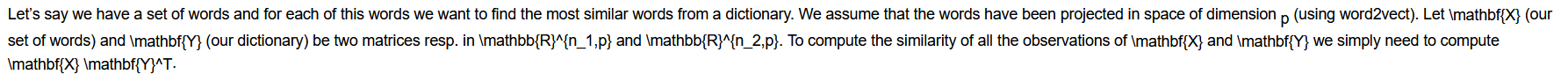
rhs\_ones=rep(1,ncol(W))

lhs\_ones=rep(1,nrow(W))

t(lhs\_ones) %\*% W%\*% rhs\_ones

}

**Similarity matrix (Gram matrix)**

.

**Python code:**

def gramMatrix(X,Y):

return X.dot(Y.T)

**R code:**

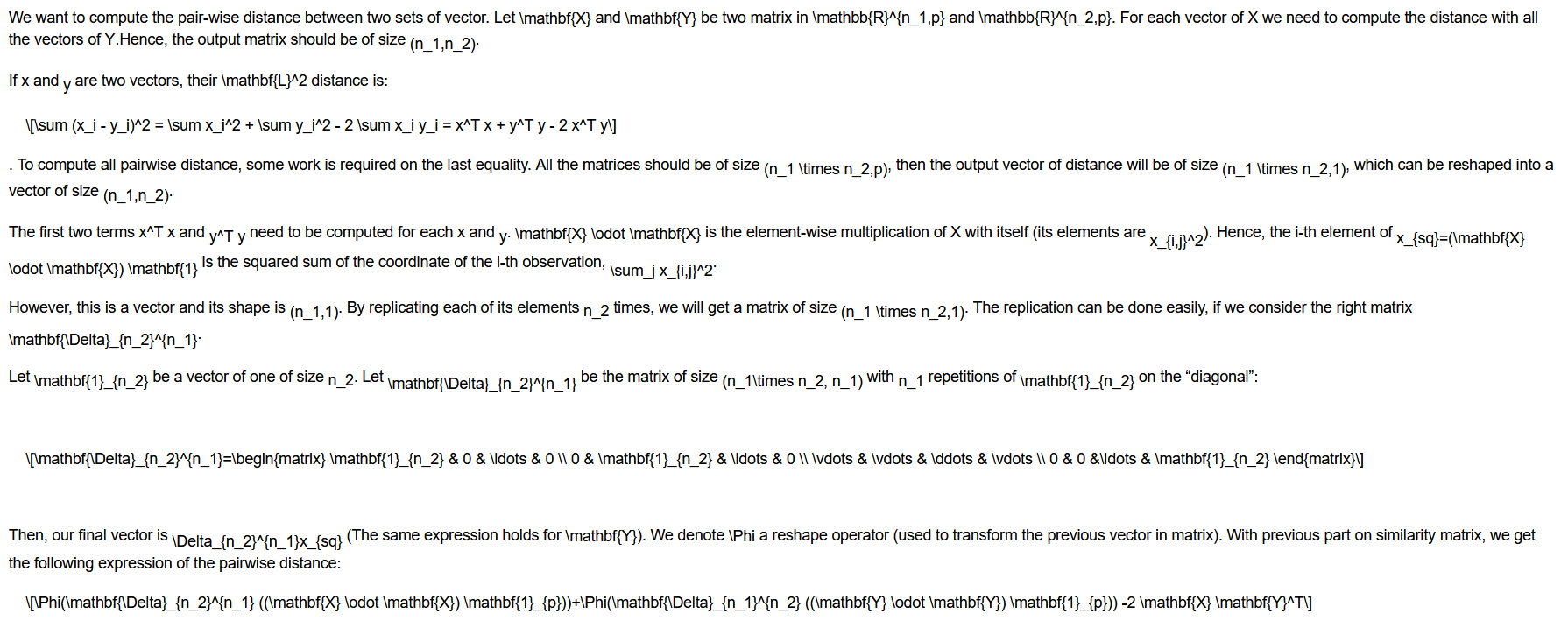
gramMatrix=function(X,Y)

{

X %\*% t(Y)

}

**L2 distance**



The previous expression can seem complex, but this will help us a lot to code the pairwise distance. We only have to do the translation from maths to Numpy or R.

**Python code:**

def L2dist(X,Y):

n\_1=X.shape[0]

n\_2=Y.shape[0]

p=X.shape[1]

ones=np.ones((p,1))

x\_sq=(X\*\*2).dot(ones)[:,0]

y\_sq=(Y\*\*2).dot(ones)[:,0]

delta\_n1\_n2=np.repeat(np.eye(n\_1),n\_2,axis=0)

delta\_n2\_n1=np.repeat(np.eye(n\_2),n\_1,axis=0)

return np.reshape(delta\_n1\_n2.dot(x\_sq),(n\_1,n\_2))+np.reshape(delta\_n2\_n1.dot(y\_sq),(n\_2,n\_1)).T-2\*gramMatrix(X,Y)

**R Code:**

L2dist=function(X,Y)

{

n\_1=dim(X)[1]

n\_2=dim(Y)[1]

p=dim(X)[2]

ones=rep(1,p)

x\_sq=X\*\*2 %\*% ones

x\_sq=t(matrix(diag(n\_1) %x% rep(1, n\_2) %\*% x\_sq, n\_2,n\_1))

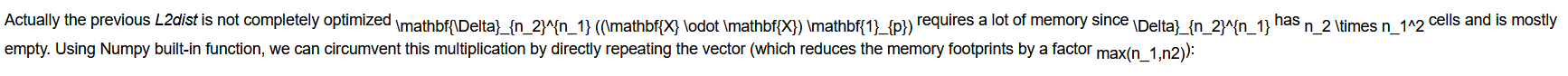
y\_sq=Y\*\*2 %\*% ones

y\_sq=matrix(diag(n\_2) %x% rep(1, n\_1) %\*% y\_sq,n\_1,n\_2)

x\_sq+y\_sq-2\*gramMatrix(X,Y)

}

**L2 distance (improved)**



**Python code:**

def L2dist\_improved(X,Y):

n\_1=X.shape[0]

n\_2=Y.shape[0]

p=X.shape[1]

ones=np.ones((p,1))

x\_sq=(X\*\*2).dot(ones)[:,0]

y\_sq=(Y\*\*2).dot(ones)[:,0]

##Replace multiplication by a simple repeat

X\_rpt=np.repeat(x\_sq,n\_2).reshape((n\_1,n\_2))

Y\_rpt=np.repeat(y\_sq,n\_1).reshape((n\_2,n\_1)).T

return X\_rpt+Y\_rpt-2\*gramMatrix(X,Y)

**R code:**

L2dist\_improved=function(X,Y)

{

n\_1=dim(X)[1]

n\_2=dim(Y)[1]

p=dim(X)[2]

ones=rep(1,p)

x\_sq=X\*\*2 %\*% ones

x\_sq=t(matrix(rep(x\_sq,each=n\_2),n\_2,n\_1))

y\_sq=Y\*\*2 %\*% ones

y\_sq=matrix(rep(y\_sq,each=n\_1),n\_1,n\_2)

x\_sq+y\_sq-2\*gramMatrix(X,Y)

}

**L2 distance – benchmark**

To show the interest of our previous work, let’s compare the computation speed of the vectorized L2 distance, the naive implementation and the scikit-learn implementation. The experiments are run on different size of dataset with 100 repetitions.

Computation time of the L2 distance

The vectorized implementation is 2 to 3 orders of magnitude faster than the naive implementation and on par with the scikit implementation.