**Report**

We are to compute pairwise Euclidean distance matrix and correlation matrix using nested loops and without loops (by vectorizing data and using matrix operations with numpy), and test computations on sklearn datasets data. We are also to compute and compare performance time for both methods. It is important that we have the fastest computational algorithm for processing large amount of data. And we are to show that without loop computation can be done much faster than computation with nested loops.

**Distance matrix** **loop** computation was done with 2 nested loops iterating through rows and columns of input array. Calculated the pairwise distance for each output element and placed it in the output distance matrix M.

**def** compute\_distance\_naive(X):  
 N = X.shape[0] *# num of rows* D = X[0].shape[0] *# num of cols* M = np.zeros([N,N])  
 **for** i **in** range(N):  
 **for** j **in** range(N):  
 xi = X[i,:] *# take ith row* xj = X[j,:] *# take jth row* dist = np.sqrt(np.sum((xi-xj)\*(xi-xj))) M[i,j] = dist  
  
 **return** M

**Distance matrix vectorized** computation was completed using Numpy matrix operations.

Input matrix is an array of vectors x0, x1,…, xN-1, each has a length D.

By the definition distance between vectors is calculated by the formula ||x-y|| = sqrt(x\*x – 2x\*y + y\*y).

So our output matrix was calculated as the square root of matrix A – 2 \* matrix B + matrix C. Where A is a matrix of size NxN which is an array of same column vectors; each column vector is consists of squared norms of vectors of the input matrix. Matrix B by the formula is an inner product matrix of input vectors size NxN. And matrix C is transposed of matrix A, since it consists of norms of the second vector in the formula.

**def** compute\_distance\_smart(X):  
 N = X.shape[0] *# num of rows* D = X[0].shape[0] *# num of cols* M = np.zeros([N, N])  
 A = np.zeros((N, N))  
 B = np.zeros((N, N))  
 C = np.zeros((N, N))  
  
 xiSum = np.sum(np.multiply(X,X), axis=1) *# row vector of length N. Sum over rows* A = np.transpose((xiSum, ) \* N) *# clone vector xisum to a matrix of size NxN* B = np.inner(X,X)  
 C = np.transpose(A)  
  
 Sum = np.add(np.add(A, -2 \* B), C)  
 M = np.sqrt(abs(Sum)) *# M = sqrt(A - 2\*B + C)* **return** M

**Correlation matrix loop** version was computed with 2 nested loops iterating through rows and columns of the input matrix.

For each 2 columns of the input matrix I computed the mean values. And used them for computing covariance at the specific row and column, and for computing variance at each row/column position. Variance was used to compute a standard deviation at that row/column position. Using that data I computed correlation matrix element at specific row and column location. It was computed for each row and column position.

**def** compute\_correlation\_naive(X):  
 N = X.shape[0] *# num of rows* D = X[0].shape[0] *# num of cols  
  
 # use X to create M* M = np.zeros([D, D])  
  
 **for** i **in** range(D):  
 **for** j **in** range(D):  
 xi = X[:, i] *# take ith column* xj = X[:, j] *# take jth column* xih = xi - np.sum(xi).astype(float) / N *# xi - mui* xjh = xj - np.sum(xj).astype(float) / N *# xj - muj* sij = (np.dot(xih, xjh)).astype(float) / (N-1) *# covariance at i row j column* stdi = np.sqrt(np.dot(xih, xih).astype(float) / (N - 1)) *# sigmai* stdj = np.sqrt(np.dot(xjh, xjh).astype(float) / (N - 1)) *# sigmaj* corr = sij.astype(float) / stdi / stdj

M[i, j] = corr  
  
 **return** M

**Correlation matrix vectorized** computation:

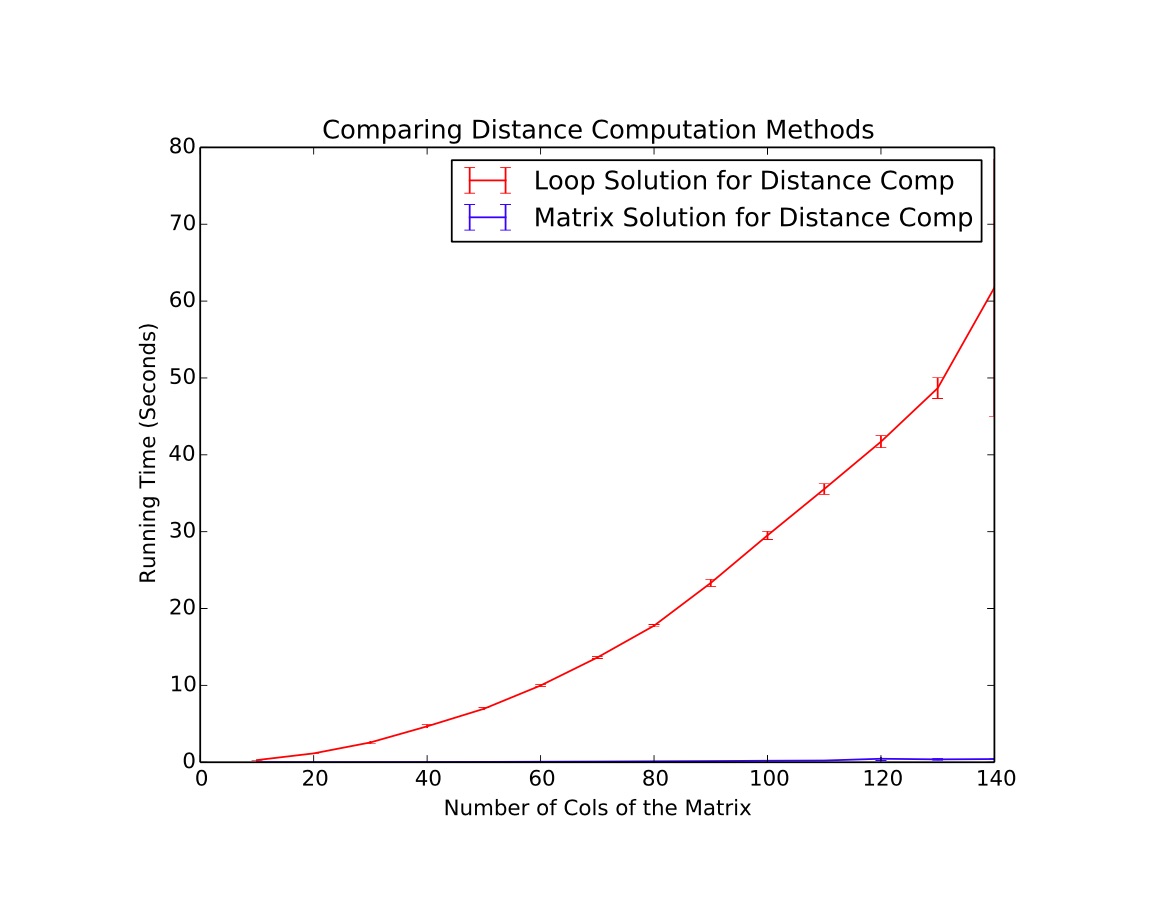
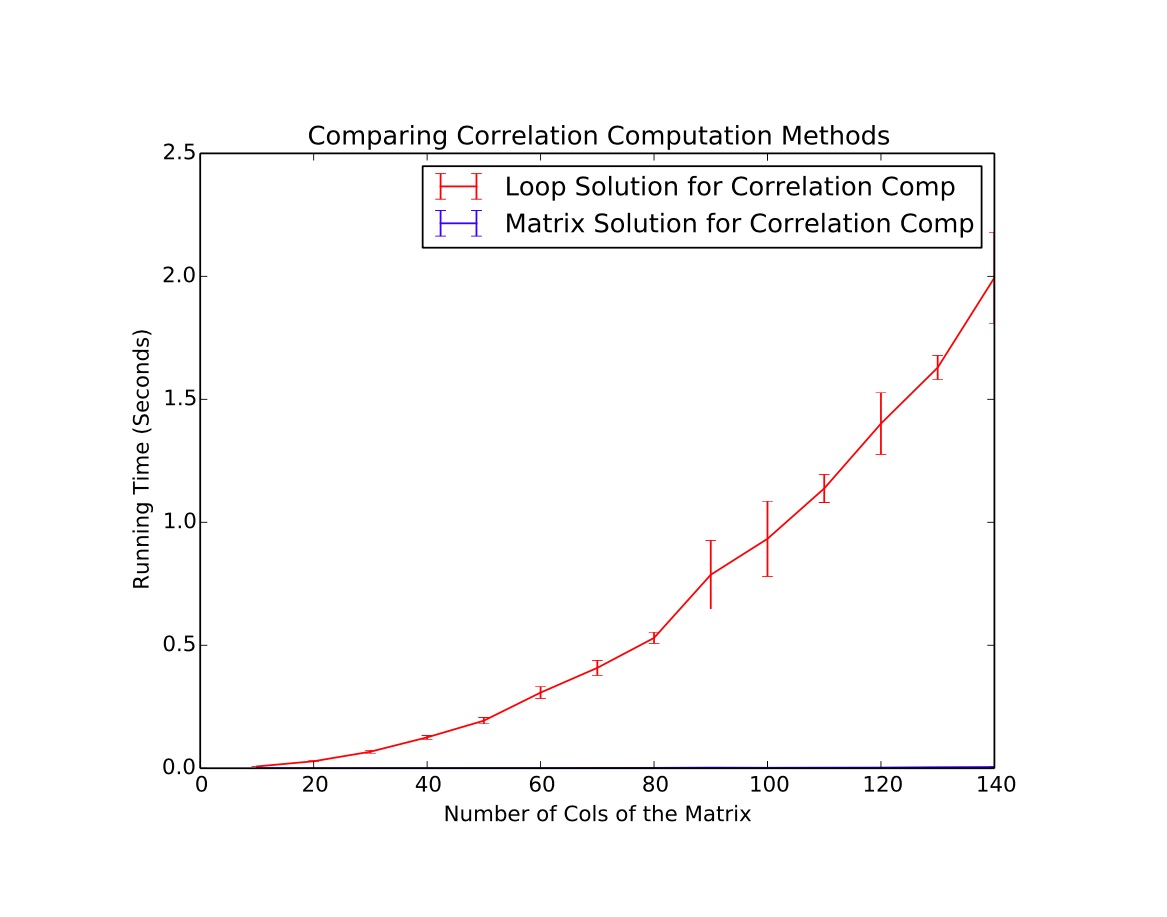
The same loop computation was performed with matrices using Numpy operations.

First computed vector of mean values, mu, and a matrix X-mu (from the formula). That matrix was used for calculating the covariance matrix S. Then vector of standard deviations was found, and it was used to compute matrix of denominators of correlation matrix. Both covariance and denominators matrices were multiplied elementwise to find the correlation matrix.

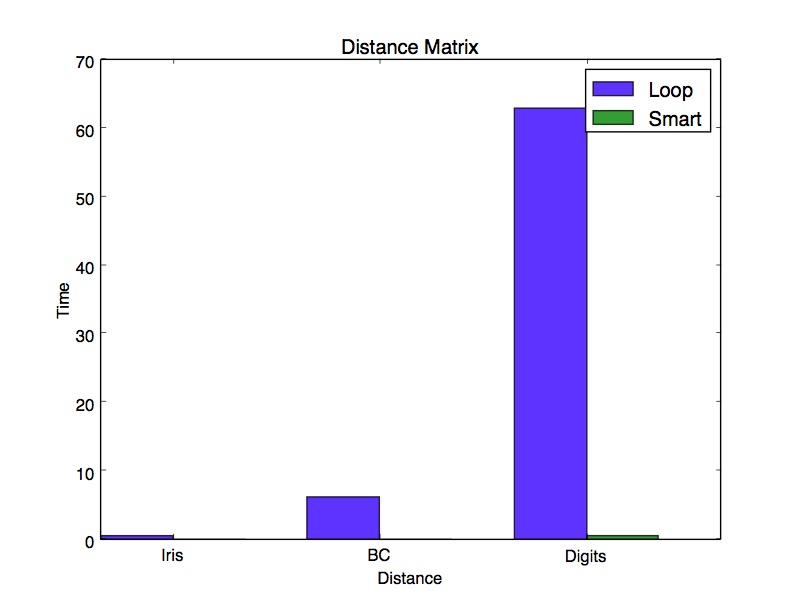
**def** compute\_correlation\_smart(X):  
 N = X.shape[0] *# num of rows* D = X[0].shape[0] *# num of cols* M = np.zeros([D, D])  
  
 mu = (np.sum(X, axis=0)).astype(float) / N *#mean vector. column sum of column vectors* MatrixMu = mu \* np.ones([N,1]) *#matrix with mu* Xmu = X - MatrixMu  
  
 S = (np.dot(np.transpose(Xmu),Xmu)).astype(float) / (N-1) *#covariance matrix* varvector = (np.sum(np.multiply(Xmu, Xmu),axis=0)).astype(float) / (N-1) *#variance vector* std = np.sqrt(varvector) *#std vector* Denom = np.outer(std, std) *#denominator matrix* M = np.multiply(S, np.power(Denom, -1))  
  
 **return** M

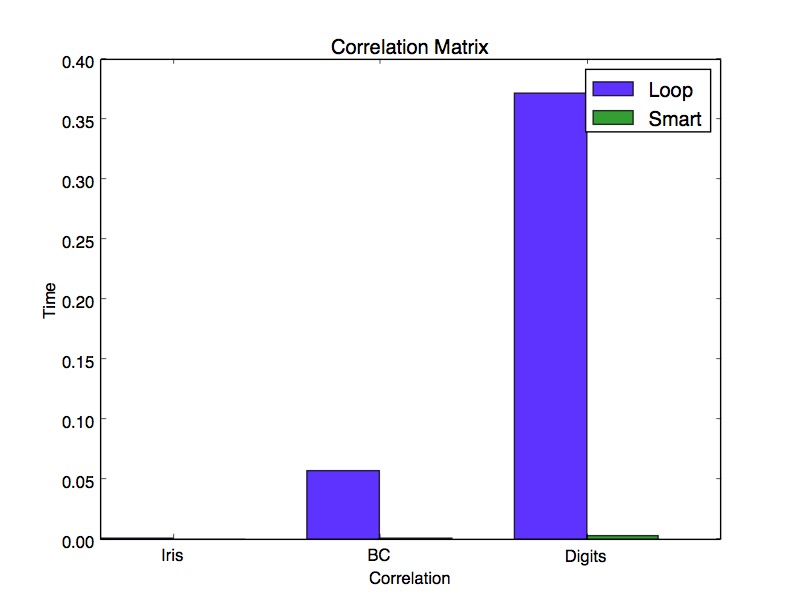
The above experiments were made to compare performance (computational time) of loop algorithms with vectorized algorithms in computing distance matrix and correlation matrix.

**The results** of computation were put in error bar charts for distance matrix and for correlation matrix. Each shows **time** growth depending on the size of input (the number of columns in input matrix). We can see that the time of vectorized computation is almost not changing with the growth of input size; meanwhile loop computation time grows exponentially.

We can see that using NumPy lets us to vectorize operations on arrays of fixed-size numeric data types. If we can successfully vectorize an operation, then it executes mostly in C, this was we avoid the overhead of the Python interpreter.

**Problem 3**

Pairwise distance matrix and correlation matrix were computed for datasets from sklearn: Iris, Breast\_cancer, Digits. The results were put in 2 bar-charts: distance matrix and correlation matrix.****

****

We can see that loop computing time is times higher than computation without loops.