“Comparing Naive and Optimized Implementations of Euclidean Distance and Correlation Matrix Computation in Python”

Prepared for

Dr. Chia-Ling Tsai
Professor, CSCI 325: Machine Learning, CUNY Queens College

By

Loyd Flores
Senior @ CUNY Queens College

Due
September 22, 2024

I. Abstract

Computing Euclidean distance and correlation matrices is fundamental to various machine learning and data science tasks. However, changes in implementations of these computations can vary in efficiency, especially for large datasets, sometimes leading to significant performance bottlenecks. This paper explores two different implementations: a naive implementation, using an iterative approach with for loops, and an optimized implementation, proposing an efficient, vectorized approach that leverages the capabilities of NumPy to reduce computational overhead. The evaluation demonstrates that the optimized method significantly improves execution, particularly when applied to high-dimensional data. These findings underscore the importance of optimized algorithms in practical applications where performance is critical.

II. Methods

1. Euclidean Distance

The euclidean distance formula is used to compute the distance between two points, It is usable for any two points even in a high-dimensional space. Euclidean distance is used in a lot of machine learning algorithms like KNN (K-Nearest-Neighbors) and Clustering. The formula given the space is 2-Dimensional:

$$d(p,q) = (q_1 - p_1)^2 + (q_2 - p_2)^2$$

1.1 Naive Implementation

To find the euclidean distance for all the rows or data points for a given matrix size (N, D), we must find the distance for row i and row j for all i and j. *N* is a variable that contains the number rows, and *D* is a variable that contains the number of columns or features in a given matrix *X*.

To do this, we need a new matrix, M, with the size (N,N), which will hold all the pairwise distances of all row pairs.

Next we iterate through all possible pairs of i and j, in code this will utilize two nested for loops, and we apply the euclidean distance formula to each i and j. M[i, j] will hold the distance between row i and row j.

1.2 Smart Implementation

In Computer Science utilizing manually iterating through matrixes is quite inefficient as it Obtains a worst case complexity of $\theta(n^2)$, meaning as your input size n increases, the The runtime of finding pairwise euclidean distances will exponentially grow as well.

There exists a library in Python called NumPy. Under the hood it is written in C and Fortran, which are both low-level programming languages that makes it much faster Then Python for numerical computations. When a NumPy function is called, it often Invokes optimized routines written in the mentioned languages, giving it a significant Performance boost compared to the standard Python loops because each iteration Involves a lot of overhead, like dynamically typed variable checks. NumPy allows for operations on entire arrays or matrices without explicit loops. This works by utilizing a concept called *Vectorization,* this refers to the process of replacing explicit loops With array operations that perform multiple operations at once, typically leveraging the CPU's ability to execute those operations in parallel. This completely removes loops out of the equation and drastically reduces the complexity of finding The euclidean distance.

The first step in finding the euclidean distance the smart way is by finding all the squared norms for all rows of X. The squared norm of each data point x_i is calculated as the sum of squares of its components which also utilizes vectorization.

Then, using NumPy to find the dot product or np.dot(X, X^T), this results in a matrix of shape (N,N), where each entry represents the dot product between two data points. After obtaining that new matrix M the squared Euclidean distance and squaring each computation to compute the pairwise distances of each point i and j.

2. **Producing the correlation matrix for all features**
   The correlation matrix is a very important tool used by Data Scientists and Data Analysts when exploring the correlation between the features of a dataset. It is a very important tool used to understand the data and problem at hand as it mathematically shows the relationships of data which provides insights that are not obvious. It utilizes the Pearson correlation formula:

$$r = \Sigma(x_i - \bar{x})(y_i - \bar{y}) / \sqrt{\Sigma(x_i - \bar{x})^2 \, \Sigma(y_i - \bar{y})^2}$$

Where r is the Pearson correlation coefficient.

2.1 Naive Implementation

Like the naive implementation of the euclidean distance formula, this also involves a manual iterative approaching, in implementation using two for loops, to find the correlation of all features.

The first step is to declare a new matrix M of size (D,D) because we are comparing all the features against one another.

The next step is to iterate over all the possible column pairs and j. As the iteration occurs, the mean and the deviation from the mean f x_i and x_j are obtained then the values are then plugged into the correlation formula until all the features have been paired against one another. M[i,j] will hold the correlation between feature i and feature j.

2.2 Smart Implementation

First step is to center the data by subtracting the mean of each column from its corresponding values.

Then the covariance matrix is calculated using the NumPy dot product. Then the standard deviation for each column needs to be collected to finally be able to use the correlation formula to obtain the correlation matrix. Again, M[i, j] will hold the correlation of feature i and feature j.

III. Results and Findings

1. Experiments

To do the comparison this study explores two experiments that involve comparing the naive implementation against the smart implementation.

a. The first experiment compares the two implementations against each other by increasing the input size in a controlled manner starting at
(N = 100, D = 10) up to (N = 1000, D = 100) to simulate different in input size

b. The second experiment involves utilizing three different data sets from sci-kit learn to simulate a more accurate representation of real-world use cases where datasets will vary in unpredictable ways. It may also provide a more stable validation technique to compare the two implementations. The datasets are the following:

1. Iris (N = 150,D = 4)
2. Breast Cancer (N = 569, D = 30)
3. Digits (N = 1797, D = 64).

1. Euclidean Distance

Figure 1.1 Distance Experiment with a predictable increase in input size

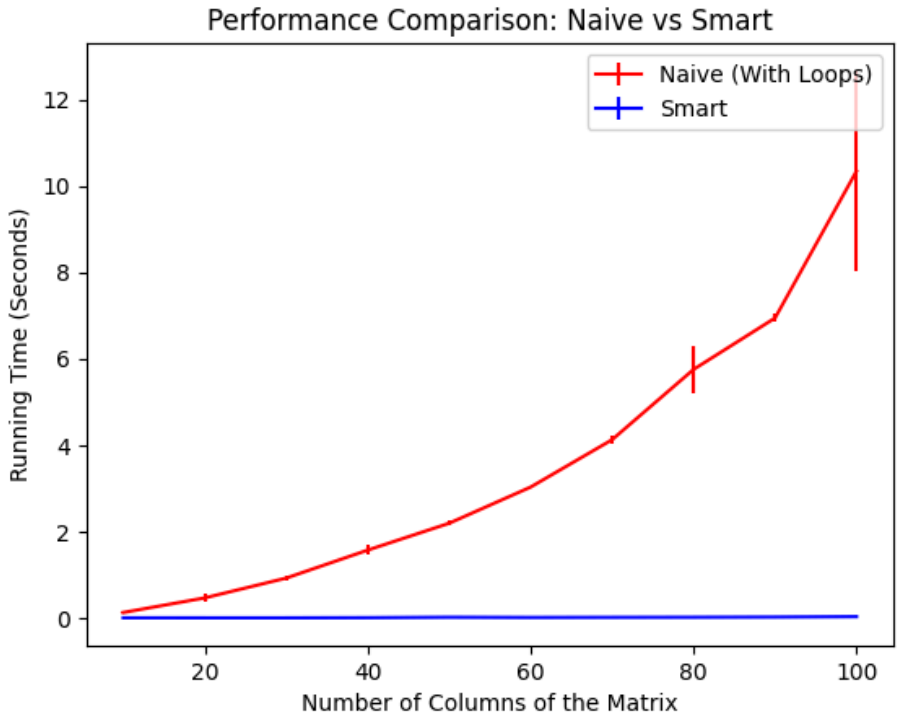
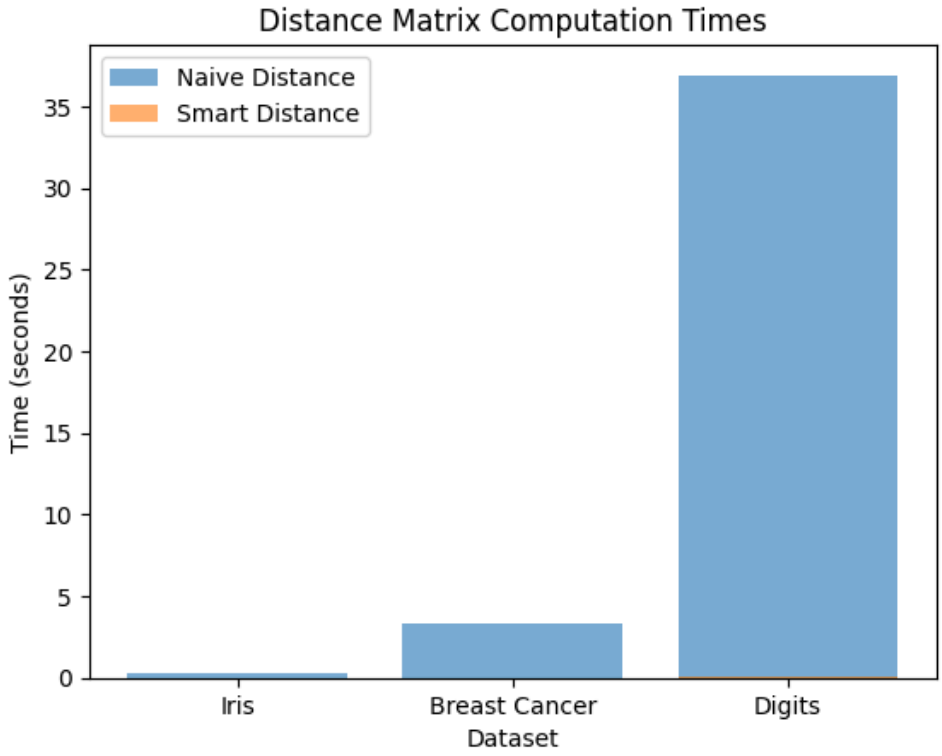


Figure 1.2 Tabulated Run time (In seconds)

Dataset	Naive Distance	Smart Distance
Iris	0.257	0.071
Breast Cancer	3.745	0.017
Digits	30.982	0.101

Figure 1.3 Distance Comparison using 3 different datasets



The experiment was run at least 5 times to ensure that results remain consistent. It is evident that that the Naive implementation with loops does in fact grow exponentially, while the smart implementation remains constant. There is an interesting finding that the smart implementation that at a specific value it becomes faster than both smaller or larger input sizes as evident in *Figure 1.2,* where the Smart implementation performed best in the breast cancer dataset being in the middle of the two other options in terms of size. It is a common assumption that a smaller input size will always Produce the best results, yet in this case somehow it is not true.

2. Finding the correlation matrix

Figure 2.1 Correlation comparison given a predictable increase in input size

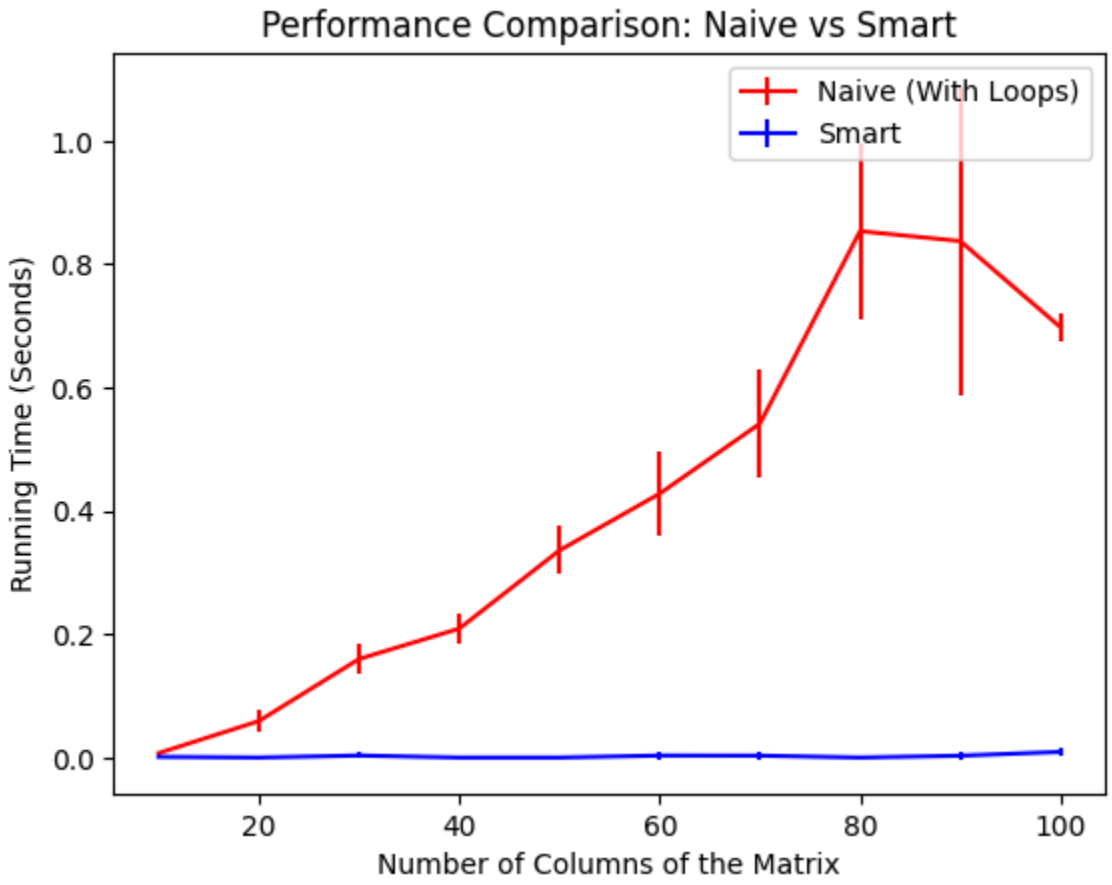
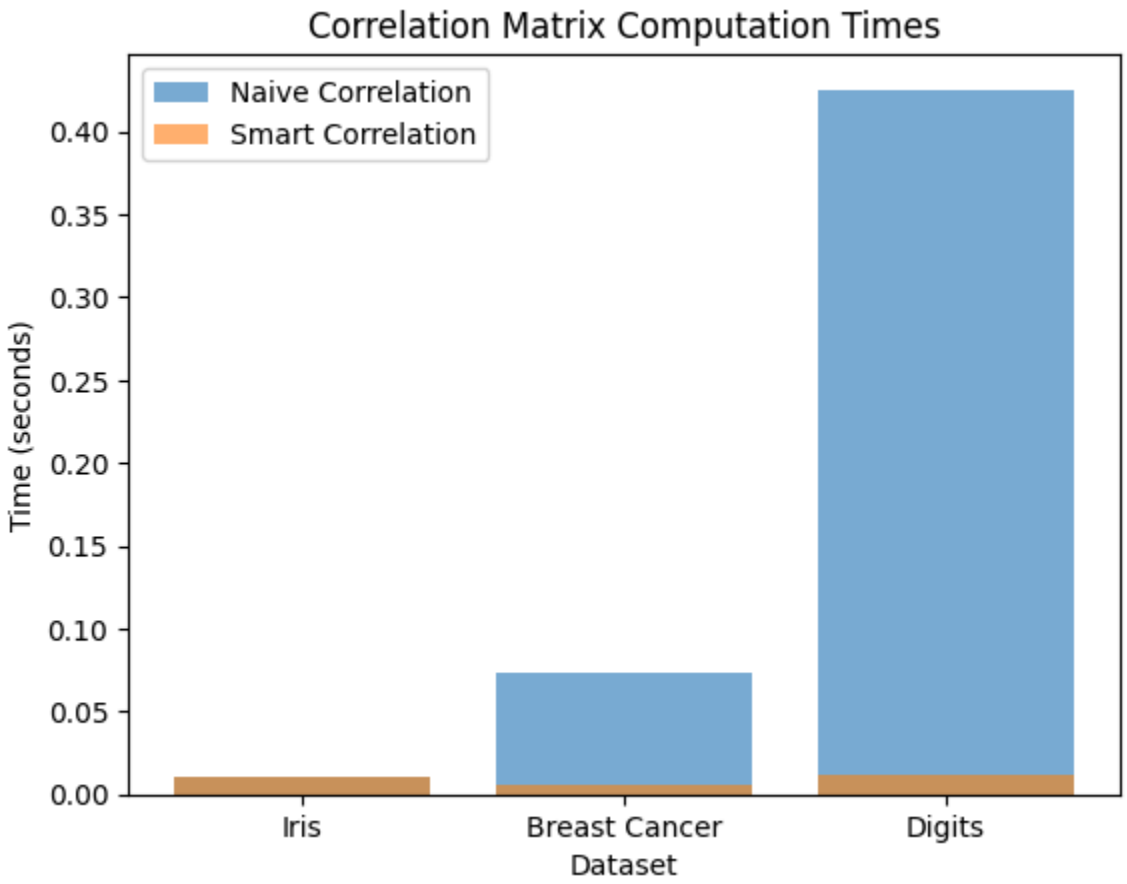


Figure 2.2 Tabulated Run time (In seconds)

Dataset	Naive Correlation	Smart Correlation
Iris	0.010	0.010
Breast Cancer	0.073	0.006
Digits	0.425	0.012

Figure 2.3 Correlation Comparison given 3 datasets



Similar to the comparison in Euclidean distance it holds that the smart implementation provides faster and more stable results independent of input size. The same is also still evident that there is an ideal input size to optimize the performance of the smart Implementation because in terms of finding the correlation the smart implementation yet again performed best when the Breast Cancer dataset was the input. Another interesting finding is that the naive implementation will reach some form of ceiling in terms of runtime and will progressively slow down as if there is a sweet spot for to perform at its worst.

IV. Conclusion

The experiments definitively answer that the smart implementation outperforms naive implementation independent of the input size. The right implementation is is just another question for the user to answer as both implementations have their tradeoffs mainly in the complexities of runtime, space utilization, and implementation. The naive implementation is definitely simpler in terms of In terms of writing out the code that implements it but also is a lot slower and less inefficient with compute resources.