Homework 1: Computing Distance Matrix and the Covariance Matrix of Data

James Baldwin

1 Solution

Problem 1: Distance

compute_distance_naive(X)

I utilized a for-loop structured around the definition of the Euclidean distance, which states that the distance $\boldsymbol{d}$ between two points $\boldsymbol{a}$ and $\boldsymbol{b}$ is given by

$$\boldsymbol{d}(\boldsymbol{a}, \boldsymbol{b}) = ||\boldsymbol{a} - \boldsymbol{b}|| = \sqrt{\sum (a_i + b_i)^2}, \tag{1}$$

where a_i and b_i represent the components of the vectors that represent the points $\boldsymbol{a}$ and $\boldsymbol{b}$.

The number of rows N and the number of columns D from the input matrix is stored and used to create a placeholder matrix of size $N \times N$. Within a double for-loop, the rows of the input matrix are stored in row vectors of size $1 \times D$. The distance between all rows X_i and X_j is computed over the N^2 iterations of the double for-loop: the difference between the D elements of each row is squared, and the resulting D values are summed. The square root of this sum is taken in order to get a value for the distance between the rows. This value is stored in the $[i][j]$th element of the distance matrix M.

compute_distance_smart(X)

In contrast to the naive version, in which the sum of the rows was computed within a double for-loop, I utilized a vectorized sum to store the squared values of the summation in a single operation. This vectorized summation $X\_squared$ results in an array of shape $(N,)$, which I then reshape into a column vector of shape $N \times 1$ for use in later calculations of the distance matrix.

I utilize an expanded version of the Euclidean distance formula, given by

$$||x_i - x_j||^2 = ||x_i||^2 + ||x_j||^2 - 2 * x_i * x_j. \tag{2}$$

In essence, this version tells us that the square of the Euclidean distance can be calculated by adding the square of the elements of two rows, and then adding twice their product. The key to the smart algorithm is to recognize that we can

utilize broadcasting and a dot product with $X\_squared$ and its transpose to efficiently carry out these calculations.

The result of these calculations is done in a vectorized manner, resulting in an $N \times N$ array, which gives us the distance matrix M by simply taking the square root of the entire thing. An extra step is taken to avoid numerical instability arising from calculating the distance between two extremely close points, taking the square of negative values by taking the square root of the maximum between zero and the distance calculation.

Problem 2: Correlation

compute_correlation_naive(X)

The correlation matrix itself is $D \times D$ rather than $N \times N$. The D standard deviations of the input matrix X also need to be stored, so a placeholder array for these is created. The correlation is calculated inside a double for-loop, which first stores the columns x_i and x_j of the matrix X. The means of these column vectors across N rows is calculated and stored.

The covariance matrix $s_{i,j}$ is given by

$$s_{i,j} = \frac{1}{N-1} \sum \big[(x_i - \mu_i)(x_j - \mu_j)\big], \tag{3}$$

where μ is the mean of the column vector. This results in a $D \times D$ matrix. When $i = j$, we have a standard deviation given by

$$\sigma_i = \sqrt{s_{i,i}}. \tag{4}$$

The covariance matrix is translated into the correlation matrix by dividing its elements by the product of the i and jth standard deviations. This is done in another double for-loop which first avoids any division by zero by checking for zero standard deviations, which can occur if the elements of a column are identical.

compute_correlation_smart(X)

The smart calculation of the correlation matrix utilizes essentially the same tools used in the smart distance calculation. The means are calculated using vectorization and stored in a column vector of length D. The step in the naive calculation of subtracting the mean from the column vector - which can be thought of as a centering operation as you zero out the mean - can be done in a single operation as well, utilizing vectorization. The summation of the product of the two centerings is again a dot product operation which can be done using the transpose of our centered matrix.

The next thing to recognize is that the standard deviations are simply the diagonal elements of the covariance matrix, which can be stored in a length D array in a single operation.

Depending on the specific data, one may have undefined correlations (as discussed in the naive version), which need to be handled in order to maintain mathematical robustness. I choose to set any undefined (divide by zero) correlations to NaN, and carry out the final calculation utilizing an outer product on the standard deviation array. This results in our final $D \times D$ correlation matrix.

2 Experiments

A matrix X with N data points containing D features is a common format to confront in machine learning. Two useful things to know about this set of data is how closely the data points N are to one another and how closely correlated the features D are to one another. A distance matrix M is a useful way to quantify the former, with the $[i][j]$th element representing the distance between the ith and jth data points (rows). A correlation matrix does the same for the latter, with the $[i][j]$th element representing the correlation between the ith and jth feature (columns).

In the three experiments carried out here (general distance and correlation calculations, calculations on specific datasets) **the control** is the naive version of the algorithms against which the smart version is ultimately compared. A brief discussion of the results and observations from each experiment follows.

Distance Calculation (general)

From Figure 1 we can see the dramatic difference between the naive and smart computation methods for calculating a distance matrix M. As the number of columns in the matrix increases, we see the computation time increase along a parabolic path, which is consistent with the N^2 calculations needed to compute the distance in the naive method. Our control in this experiment is the naive method, against which the smart method is compared. It is clear from this experiment that the matrix and vectorization methods employed in the smart function have a dramatic impact on the calculation of a distance calculation.

Correlation Calculation (general)

While we can see from Figure 2 that the overall correlation calculations were much faster, the general shape remains consistent. This is in line with the $\sim D^2$ operations needed to compute the correlation matrix. The increase in general speed is in line with our expectations, given that the data sets have number of rows $>>$ number of columns. In this experiment, the control is the naive method against which the smart method is compared.

`scikit-learn` Datasets

Given the shapes of the datasets (Iris 150×4, Breast Cancer 569×30, Digits 1797×64), the results we see are consistent with the experiments run on a

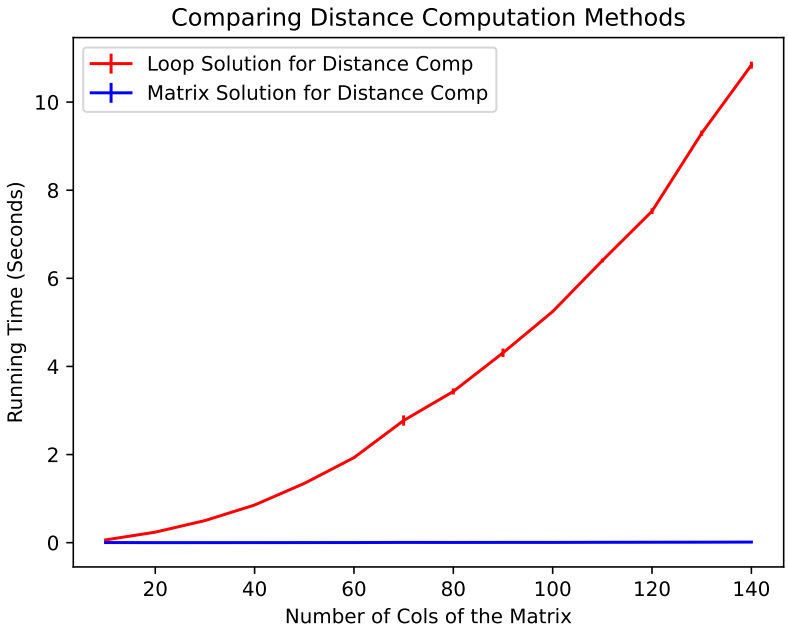

Figure 1: Comparing computation times for the naive (loop) and smart (matrix) methods of distance calculation. The naive method can take more than 10 seconds for larger matrices ($\geq$ 140 columns) while the smart method takes $<<$ 1 second. This parabolic behavior is in line with the N^2 number of operations needed in using the naive method.

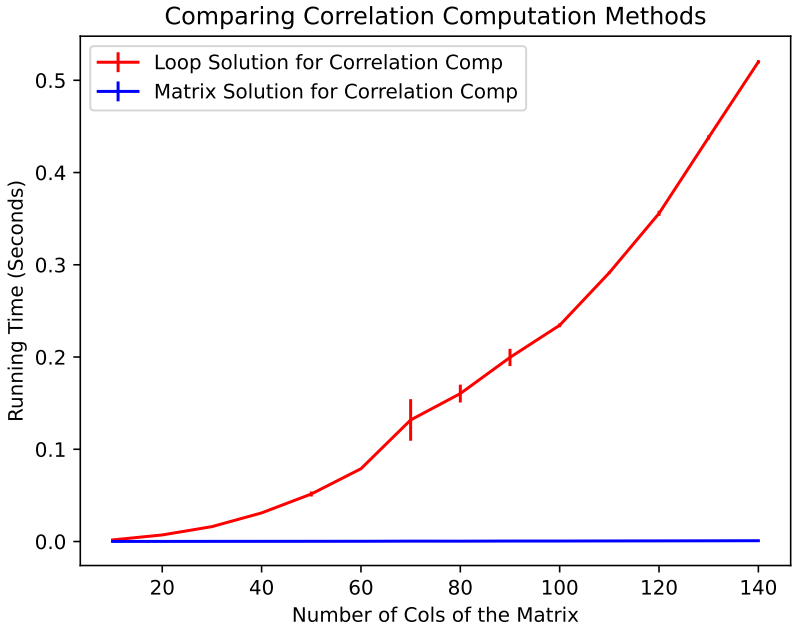

Figure 2: Comparison of computation times for the naive (loop) and smart (matrix) methods of calculating the correlation matrix for a set of data. The parabolic curve of the naive computation times is in line with the $\sim D^2$ number of operations required for calculating the correlation matrix using the naive method.

	naive dist	smart dist	naive corr	smart corr
iris	0.085735	0.000255	0.000291	0.000048
breast cancer	0.669805	0.004438	0.011505	0.000155
digits	6.630755	0.019973	0.055493	0.000487

Figure 3: Table showing the runtimes (seconds) of the different computation algorithms for the sklearn datasets Iris, Breast Cancer, and Digits.

general matrix X. The overall results are shown in table format in Figure 3 and in graphical format in Figure 4. The bar chart, showing computation times on a log scale, gives the overall picture of the effect of utilizing matrix methods to make calculations. In almost every instance, we can see the computation times improve by nearly three orders of magnitude. In this experiment, the control is again the naive method(s) used to calculate the distance and correlation matrices. The bar chart in Figure 4 makes clear the advantage of utilizing matrix and vectorization approaches in these calculations.

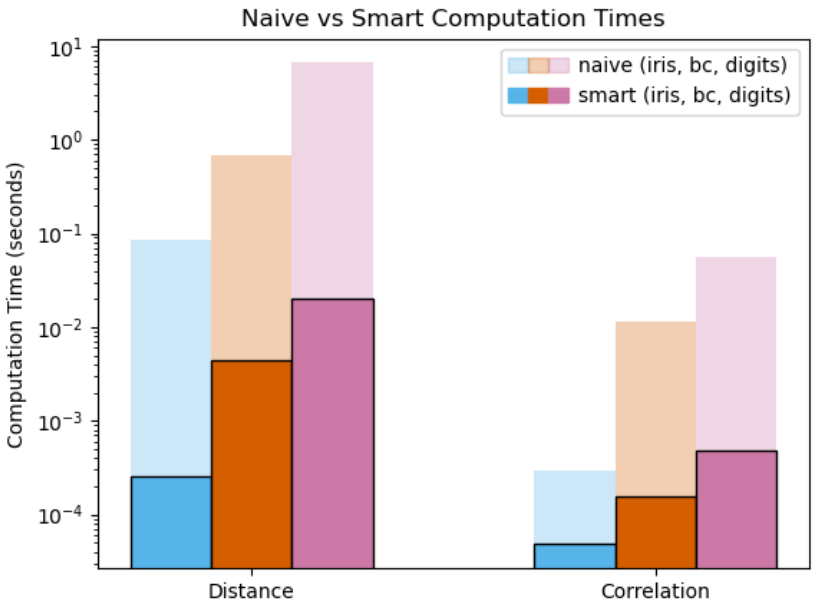

Figure 4: Bar chart showing the (log) computation times for the naive and smart methods on each of the three sklearn datasets. The faded bars represent the naive runtimes, with the full color bars representing smart method runtimes. The blue bars correspond to the iris dataset, the orange to the breast cancer (bc) dataset, and the magenta to the digits dataset.