NumPy / SciPy Recipes for Data Science: Squared Euclidean Distance Matrices

Christian Bauckhage
B-IT, University of Bonn, Germany
Fraunhofer IAIS, Sankt Augustin, Germany

***Abstract*—In this note, we explore and evaluate various ways of computing squared Euclidean distance matrices (EDMs) using *NumPy* or *SciPy*. In particular, we discuss 6 increasingly abstract code snippets where the more abstract ones require more experience but also are significantly more efficient. We conclude our discussion with suggestions as to when to use which implementation.**

I. Introduction

The need to compute squared Euclidean distances between data points arises in many data mining, pattern recognition, or machine learning algorithms. Often, we even must determine whole matrices of squared distances [1]–[5]. To be specific, if we are given an $m \times n$ data matrix

$$\boldsymbol{X} = \begin{bmatrix}\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\end{bmatrix}$$

whose n column vectors $\boldsymbol{x}_i$ are m dimensional data points, the task is to compute an $n \times n$ matrix $\boldsymbol{D} \in \mathbb{R}^{n \times n}$ where

$$D_{ij} = \left\|\boldsymbol{x}_i - \boldsymbol{x}_j\right\|^2.$$

Here, we explore different ways of how this can be done in *Python*. For simplicity, we suppose that both parameters m and n are small enough for the data to fit into memory. Still, for growing n, the matrix $\boldsymbol{D}$ quickly becomes large and practical applications will benefit from efficient implementations.

To illustrate that naïve implementations are too inefficient to allow for large scale data processing, we consider 6 different approaches and evaluate their runtime behavior in a series of experiments.

Our discussion assumes that readers are passably familiar with *NumPy* and *SciPy* [6]. Also, for each of the code snippets presented below, we require that *NumPy* and *SciPy* modules have been imported as follows:

```
# import numpy
import numpy as np
# import numpy linear algebra module
import numpy.linalg as la
# import scipy spatial module
import scipy.spatial as spt
# import scipy weave module
import scipy.weave as weave
```

II. Method 1: A naïve approach

Experienced *NumPy*/*SciPy* programmers know that *NumPy* readily provides a method for the computation of vector norms in its `linalg` module. So why not use it?

Listing 1: EDM method 1

```
def compute_squared_EDM_method1(X):
    # determine dimensions of data matrix X
    m,n = X.shape
    # initialize squared EDM D
    D = np.zeros((n,n))
    # iterate over upper triangle of D
    for i in range(n):
        for j in range(i+1,n):
            D[i,j] = la.norm(X[:,i] - X[:,j])**2
            D[j,i] = D[i,j]
    return D
```

The obvious idea is to initialize memory for matrix $\boldsymbol{D}$, to iterate over its rows i and columns j, and to compute D_{ij} as the squared norm of $\boldsymbol{x}_i - \boldsymbol{x}_j$. Using the *NumPy* method `norm`, this operation can be implemented as

```
D[i,j] = la.norm(X[:,i] - X[:,j])**2
```

We also note that distance matrices are symmetric so that $D_{ij} = D_{ji}$ and that their diagonal elements are identically zero. This allows for efficient computation by invoking `norm` only for the elements in the upper triangle of $\boldsymbol{D}$ and copying the results into the lower triangle. Taken together, all these ideas lead to an implementation as shown in listing 1.

III. Method 2: Avoiding square roots

Looking at listing 1, it appears somewhat redundant to first apply the method `norm` which computes

$$\left\|\boldsymbol{x}_i - \boldsymbol{x}_j\right\| = \sqrt{(\boldsymbol{x}_i - \boldsymbol{x}_j)^T(\boldsymbol{x}_i - \boldsymbol{x}_j)} \tag{1}$$

and then to set $D_{ij} = \left\|\boldsymbol{x}_i - \boldsymbol{x}_j\right\|^2$. As the computation of square roots is comparatively costly, it should be avoided if possible. Why not immediately compute

$$D_{ij} = (\boldsymbol{x}_i - \boldsymbol{x}_j)^T(\boldsymbol{x}_i - \boldsymbol{x}_j) \tag{2}$$

and thereby sidestep the repeated evaluation of square roots?

Using the *NumPy* method `dot` to compute inner products between vectors, we may implement (2) as

```
D[i,j] = np.dot(X[:,i]-X[:,j], X[:,i]-X[:,j])
```

Aiming at efficiency, we note that computing the difference vector $\boldsymbol{d} = \boldsymbol{x}_i - \boldsymbol{x}_j$ and setting $D_{ij} = \boldsymbol{d}^T\boldsymbol{d}$ will achieve further speed-up. This naturally leads to the code snippet in listing 2.

Listing 2: EDM method 2

```
def compute_squared_EDM_method2(X):
    # determine dimensions of data matrix X
    m,n = X.shape
    # initialize squared EDM D
    D = np.zeros((n,n))
    # iterate over upper triangle of D
    for i in range(n):
        for j in range(i+1,n):
            d = X[:,i] - X[:,j]
            D[i,j] = np.dot(d, d)
            D[j,i] = D[i,j]
    return D
```

Listing 3: EDM method 3

```
def compute_squared_EDM_method3(X):
    # determine dimensions of data matrix X
    m,n = X.shape
    # compute Gram matrix
    G = np.dot(X.T, X)
    # initialize squared Euclidean distance matrix
    D = np.zeros((n,n))
    # only iterate over upper trianlge
    for i in range(n):
        for j in range(i+1,n):
            # make use of |a-b|^2 = a'a + b'b - 2a'b
            D[i,j] = G[i,i] - 2*G[i,j] + G[j,j]
            D[j,i] = D[i,j]
    return D
```

IV. Method 3: Avoiding repeated inner products

Looking at listing 2, we observe that the *NumPy* function `dot` is called $\frac{n^2-n}{2}$ times and each time implicitly evaluates m multiplications and $m-1$ additions. Although `dot` makes use of highly tuned linear algebra packages, there seems to be potential for further speedup. In order to see how, we note that the squared Euclidean distance between a pair of vectors can be expressed as

$$\begin{aligned} D_{ij} &= (\boldsymbol{x}_i - \boldsymbol{x}_j)^T(\boldsymbol{x}_i - \boldsymbol{x}_j) \\ &= \boldsymbol{x}_i^T\boldsymbol{x}_i - 2\,\boldsymbol{x}_i^T\boldsymbol{x}_j + \boldsymbol{x}_j^T\boldsymbol{x}_j \end{aligned} \tag{3}$$

Hence, by pre-computing the *Gram matrix* $\boldsymbol{G} = \boldsymbol{X}^T\boldsymbol{X}$ with $G_{ij} = \boldsymbol{x}_i^T\boldsymbol{x}_j$, the distance in (3) can be computed in terms of only a single multiplication and two additions, i.e.

$$D_{ij} = G_{ii} - 2\,G_{ij} + G_{jj}. \tag{4}$$

Using `dot`, the computation of the gram matrix $\boldsymbol{G}$ required for this trick can be implemented as

```
G = dot(X.T, X)
```

Because of the efficiency of the underlying libraries for matrix multiplication, this is considerably faster than calling `dot` for each pair of vectors separately. These considerations lead to the implementation shown in listing 3.

V. Method 4: Avoiding for loops

Experienced *Python* programmers will still be worried when looking at our solutions so far. Each of them involves nested `for` loops and we know that, in *Python*, we should avoid `for` loops if speed is essential. Indeed, from a *NumPy* point of view, the code examples considered above are not properly *vectorized*. This is to say that we have not yet properly adopted our code to the linear algebra libraries [7]–[9] that are at the heart of *NumPy*. In other words, to fully exploit the number crunching capabilities of *NumPy*, we should strive to express whatever it is that we want to compute in terms of matrix operations only.

Luckily, for the case of squared Euclidean distance matrices, this turns out to be possible. To see how, let us assume matrix $\boldsymbol{D}$ could be computed as

$$\boldsymbol{D} = \boldsymbol{H} + \boldsymbol{K} - 2\,\boldsymbol{G} \tag{5}$$

where $\boldsymbol{G}$ is the Gram matrix from above. Then, by comparison to (4), we find that $H_{ij} = G_{ii}$ and $K_{ij} = G_{jj}$.

This means that matrix $\boldsymbol{H}$ is an $n \times n$ matrix whose columns each correspond to the diagonal of $\boldsymbol{G}$. Using the *NumPy* methods `diag` and `tile`, we can implement this as

```
H = np.tile(np.diag(G), (n,1))
```

Moreover, $\boldsymbol{K} = \boldsymbol{H}^T$. The squared EDM we are after can therefore be computed as

$$\boldsymbol{D} = \boldsymbol{H} + \boldsymbol{H}^T - 2\,\boldsymbol{G} \tag{6}$$

and the corresponding code in listing 4 does not contain `for` loops anymore.

Listing 4: EDM method 4

```
def compute_squared_EDM_method4(X):
    # determine dimensions of data matrix X
    m,n = X.shape
    # compute Gram matrix G
    G = np.dot(X.T, X)
    # compute matrix H
    H = np.tile(np.diag(G), (n,1))
    return H + H.T - 2*G
```

Listing 5: EDM method 5

```
def compute_squared_EDM_method5(X):
    V = spt.distance.pdist(X.T, 'sqeuclidean')
    return spt.distance.squareform(V)
```

VI. Method 4: Resorting to build-in functions

Distance matrix computation is such a common problem that *SciPy* readily provides various methods for this task. The *SciPy* module `spatial` contains a sub-module `distance` which provide functions for distance matrix computation for various distance measures. If we resort to these functions, we may compute an EDM as shown in listing 5.

VII. Method 6: Using inline C code

Our final solution requires basic familiarity with C. `Weave` is a *SciPy* module that allows for including C/C++ code in *Python* scripts. This code is compiled and linked at run-time the very first time the corresponding script is executed. For quick reuse, the compiled C code is cached on disk and executed inline whenever the script is called again.

Listing 6: EDM method 6

```
def compute_squared_EDM_method6(X):
    m,n = X.shape
    G = np.dot(X.T, X)
    D = np.zeros((n,n))

    code = \
    """
    int i, j;
    for (i = 0; i < n; i++) {
      for (j = i+1; j < n; j++) {
        D(i,j) = G(i,i) - 2*G(i,j) + G(i,j);
        D(j,i) = D(i,j);
      }
    }
    """
    D = weave.inline(code, ['G', 'D', 'n'],
                     type_converters=weave.converters.blitz,
                     compiler='gcc')

    return D
```

As compiled C code is typically very efficient in iterating over blocks of memory, in-lining C code within *Python* scripts may provide significant performance gains in tasks that require processing of large arrays of numbers.

Listing 6 demonstrates the use of `weave` methods for EDM computation. This solution is basically a variant of method 3 where the *Python* `for` loops are replaced by corresponding C code. Likely the most important point to note about listing 6 is that elements of a `weave` array are referenced using *parentheses* rather than *brackets*.

VIII. Evaluation

Next, we discuss results obtained from practical evaluations of the runtime behavior of the 6 methods considered above. All experiments were carried out on an Intel i5 CPU running Ubuntu 12.04; the *NumPy* and *SciPy* versions considered in our tests were 1.8 and 0.13, respectively.

To evaluate dependencies between runtime and data dimensionality, we randomly created different $m \times n$ data matrices $\boldsymbol{X}$ where $n = 1000$ and $m \in \{2^i\}_{i=1}^{8}$. For each of the resulting matrices and each of the above methods, we performed 10 distance matrix computations and determined the respective average runtimes.

Figure 1 plots average runtimes versus data dimensionality. Apparently, the impact of data dimensionality on the runtime of EDM computations is rather insignificant. Moreover, we recognize a clear distinction between methods 1, 2, and 3 and methods 4, 5, and 6. While the former require up to several seconds for processing, the latter perform the tasks at hand in mere fractions of a second.

To evaluate the relation between runtime behavior and the number of data points, we randomly created different $m \times n$ data matrices $\boldsymbol{X}$ where the data dimensionality m was set to 8 and the number of data points n was chosen from the set $\{50, 100, 200, 400, 800, 1600, 3200, 6400\}$. Again, we performed 10 distance matrix computations for each of the resulting matrices using each of the above methods and determined the corresponding average runtimes.

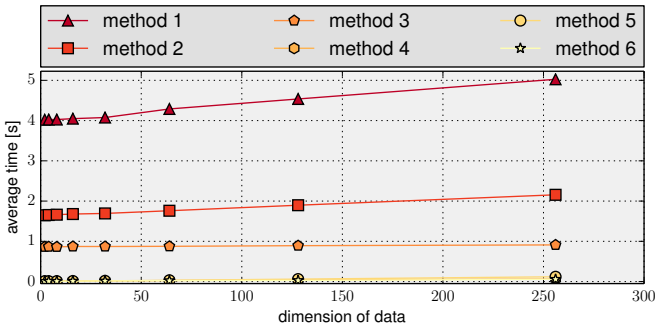


Fig. 1: Average computation times for $n = 1000$ data vectors of dimensions $m \in \{2, 4, 8, 16, 32, 64, 128, 256\}$.

Our results are shown in Figures 2(a) and 2(b). Here, we observe distinctively increasing runtimes for growing values of n. This was, of course, to be expected, since the size of a distance matrix grows quadratically in n.

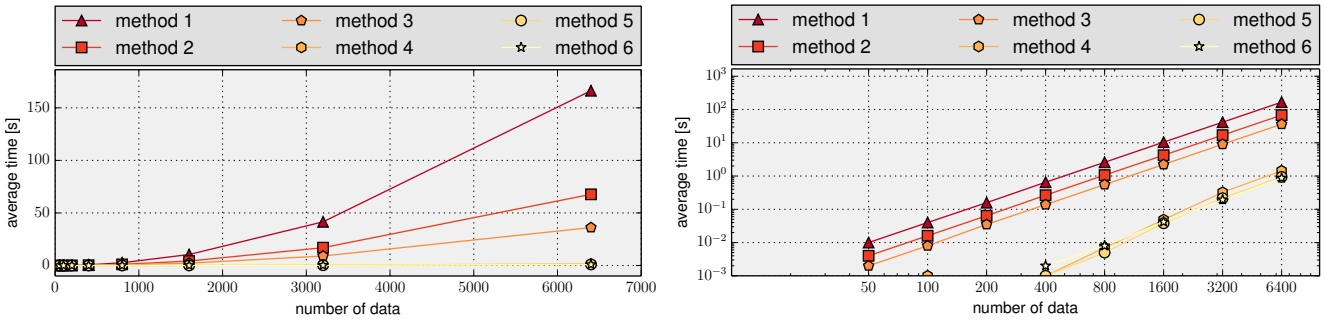
Again, we observe pronounced differences in the runtime behavior of methods 1, 2, and 3 and methods 4, 5, and 6. Especially for larger values of n, the former perform orders of magnitude worse than the latter. In particular, methods 1 and 2 appear to be infeasible for large scale problems.

Interestingly, the performance differences between methods 4, 5, and 6 seem minuscule. In fact, they only become apparent from looking at the logarithmic plot in Fig. 2(b). This plot, however, suggests that, for $n > 1600$, method 4 is slightly less efficient than method 5 which, in turn, is slightly less efficient than method 6.

IX. Summary and Suggestions

We discussed and evaluated 6 different methods for computing squared Euclidean distance matrices using *NumPy* or *SciPy*. Our experimental results underlined that the efficiency of squared EDM computation critically depends on the number n of data points. In particular, we saw that, for growing n, the average runtimes of the different methods can differ by several orders of magnitude. With respect to practical applications, these observations lead to the following suggestions:

- **Method 1** and **method 2** should be avoided at all costs (no pun intended!). The corresponding implementations are too naïve and far too slow to tackle modern, large scale data processing problems.
- **Method 3** is faster than the previous two but having to compute the Gram matrix introduces memory overhead. An advantage of this method is that it involves rather well readable code so that it might indeed be considered for problems of moderate size (say $n \leq 1000$).
- If computational efficiency is pivotal, neither of these three methods should be considered!
- **Method 4** is the most "numpythonic" way of computing a squared Euclidean distance matrix. The code in listing 4 is properly vectorized, i.e. it avoids `for` loops but resorts to highly efficient matrix computations. Also, it does not require special purpose *SciPy* modules but only depends



(a) (b)

Fig. 2: Average computation times for $n \in \{50, 100, 200, 400, 800, 1600, 3200, 6400\}$ data vectors of dimension $m = 8$.

on *NumPy* so that it can be more easily ported between platforms. Disadvantages are the memory overhead due to the computation of matrices $\boldsymbol{G}$ and $\boldsymbol{H}$ and the reduced readability of the code because of its vectorized form.

- **Method 5** is very fast and the corresponding code is comparatively well readable. A disadvantage consists in its dependency on specialized *SciPy* modules so that the code in listing 5 may pose problems with respect to portability between different computing platforms.
- If speed is of utmost importance, then **method 6**, i.e. the use of inline C code, appears to be appropriate. This, however, requires proficiency with C and seems to defy the purpose and the advantages of *Python* programming in data science.
- Overall, **methods 4** and **5** appear to be most appropriate for the task of computing squared Euclidean distances between a large number of data points. In fact, these implementation reach the efficiency of special purpose languages for scientific computing and underline that *NumPy* and *SciPy* are adequate choices for implementing data mining, pattern recognition, and machine learning algorithms.

References

[1] A. Jain, R. Duin, and J. Mao, "Statistical Pattern Recognition: a Review," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 22, no. 1, 2011.

[2] C. Faloutsos and K.-I. Lin, "FastMap: A Fast Algorithm for Indexing, Data-mining and Visualization of Traditional and Multimedia Datasets," in *Proc. SIGMOD*. ACM, 1995.

[3] C. Thurau, K. Kersting, and C. Bauckhage, "Yes We Can – Simplex Volume Maximization for Descriptive Web-Scale Matrix Factorization," in *Proc. CIKM*. ACM, 2010.

[4] N. Krislock and H. Wolkowicz, "Euclidean Distance Matrices and Applications," in *Handbook on Semidefinite, Conic and Polynomial Optimization*, M. Anjos and J. Lasserre, Eds. Springer, 2012.

[5] C. Thurau, K. Kersting, M. Wahabzada, and C. Bauckhage, "Descriptive Matrix Factorization for Sustainability: Adopting the Principle of Opposites," *Data Mining and Knowledge Discovery*, vol. 24, no. 2, 2012.

[6] T. Oliphant, "Python for Scientific Computing," *Computing in Science & Engineering*, vol. 9, no. 3, 2007.

[7] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," *ACM Trans. on Mathematical Software*, vol. 5, no. 3, 1979.

[8] J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff, "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Trans. on Mathematical Software*, vol. 16, no. 1, 1990.

[9] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed. SIAM, 1999.