PC-2019/20 Mean shift clustering

Francesca Del Lungo
francesca.dellungo2@stud.unifi.it

Matteo Petrone
matteo.petrone@stud.unifi.it

Abstract

Mean shift represents a general non-parametric clustering procedure. This algorithm has a quadratic computational complexity, but its structure is embarrassingly parallel so it is perfectly suited to be implemented in parallel.

We will see a sequential implementation and two parallel implementations in openMP and CUDA and a further optimization to make parallel programming with CUDA even more efficient.

Finally, the experiments carried out and the results obtained will be presented.

1. Introduction

Mean shift is non-parametric clustering procedure [3, 1] and it was first proposed in 1975 by Fukunaga and Hostetler [4]. In contrast to the classic K-means clustering approach here there are no assumptions on the shape of the distribution nor the number of clusters.

Mean shift builds upon the concept of kernel density estimation (KDE), that is a method to reconstruct the probability density function (PDF) given a set of data points.

The basic idea of KDE is as follows: at each step of the procedure a kernel function is applied to each point that causes the points to shift in the direction of the nearest peak of the KDE surface (gradient ascent approach).

At the end of the procedure, stationary points correspond to the *modes* of the distribution (cluster centroids) and each *cluster* is given by all data points in the attraction basin of a mode, where the attraction basin is the region for which all trajectories lead to the same mode [6].

1.1. Kernel function

There are two important parameters for the KDE approach: the *kernel*, which specifies the shape of the distribution placed at each point, and the kernel *bandwidth*, which controls the size of the kernel at each point.

There are many different types of kernel functions [11], the most frequently used is the Gaussian kernel:

$$K(x) = e^{-\frac{x^2}{2\sigma^2}} \tag{1}$$

The standard deviation σ represents the smoothing parameter (called bandwidth). It is critical because it determines the amount of smoothing: a too small value may cause the estimator to show insignificant details (many small clusters), while a too large value causes oversmoothing of the information contained in the sample (fewer but larger clusters) [11].

1.2. Mean shift procedure

Given N data points x_i, $i = 1...N$ in the d-dimensional space R^d, the position in which each point is shifted at each step t of the algorithm is computed as a weighted average between the considered point and all the others, where the weights are computed according to the given kernel function.

The general mean shift procedure for a given point x is as follows:

1. Compute the mean shift vector $m(x^t)$:

$$m_h(x) = \frac{\sum_{i=1}^{N} x_i K\left(x - x_i\right)}{\sum_{i=1}^{N} K\left(x - x_i\right)} - x$$

2. Translate the kernel (window) by $m(x^t)$:

$$x^{t+1} = x^t + m_h(x^t)$$

3. Iterate previous steps until convergence.

2. Implementation

In the mean shift procedure we can use some different *stop condition*: as we can notice in Algorithm 1 we choose to set a fixed number of iterations, this allows us to compare the execution time of the algorithm in the different versions in a more objective way.

Furthermore, we choose the *gaussian kernel* as the kernel function and the *euclidean distance* to measure the distance between pairs of points.

The main *data structures* used in the algorithm are two vectors: `original_points` contains all the points in their original positions (it remains unchanged throughout the entire algorithm) and `shifted_points` is where new positions are stored after the shifting step.

As can be easily deduced from Algorithm 1 and Algorithm 2 the mean shift procedure has a quadratic computational complexity[1], so it is computationally (relatively) expensive and it does not scale well with dimension of feature space. Even the creators of the algorithm, Fukunaga and Hostetler [4], recognized from the very beginning that their algorithm *"may be costly in terms of computer time and storage"*.

On the other hand the mean shift algorithm is *embarrassingly parallel*, in fact each single point can be processed independently from the others.

2.1. Sequential version

Below we can see the pseudocode of the mean shift algorithm implemented (Algorithm 1) and the subroutine used to shift each single point (Algorithm 2). This sequential version has been implemented in C++.

Algorithm 1 Mean shift algorithm

```
procedure MEANSHIFT($original\_points$)
    $shifted\_points \leftarrow original\_points$
    while $iteration < N\_ITERATIONS$ do
        for $point$ in $shifted\_points$ do
            $point \leftarrow$ SHIFTPOINT($point, original\_points$)
```

Algorithm 2 Mean shift subroutine

```
procedure SHIFTPOINT($point, original\_points$)
    $num \leftarrow 0$
    $den \leftarrow 0$
    for $op$ in $original\_points$ do
        $dist \leftarrow$ COMPUTEDISTANCE($point, op$)
        $w \leftarrow$ KERNEL($dist, BANDWIDTH$)
        $num \leftarrow num + op * w$
        $den \leftarrow den + w$
    return $num/den$
```

2.2. Parallel version with openMP

OpenMP (Open Multiprocessing) [9] is an API for shared-memory parallel programming, it follows the the fork-join programming model: here the idea is speeding up parts of the application by exploiting CPU parallelism.

OpenMP does not require restructuring the serial program: in order to use it you only need to add the *compiler directives* to transform the serial program into a parallel one. As can be notice from Algorithm 3 with a single `pragma` command it is possible to switch from a sequential to a parallel version of the algorithm.

OpenMP automatically partitions the iterations of a for loop with the `parallel for` construct and by default OpenMP statically assigns loop iterations to threads: when the parallel for block is entered, it assigns each thread the set of loop iterations it is to execute. The same could be obtained with `schedule(static, chunk-size)` construct. When no chunk-size is specified, OpenMP divides iterations into chunks that are approximately equal in size and it distributes at most one chunk to each thread.

The openMP scheduling type can be modified with explicit `schedule` clause, but in our case the static scheduling assures that each thread receives the same workload and this is the best choice as the number of iterations of the for loop is fixed.

Moreover with the `num_threads(num)` we can set the number of threads in a thread team.

Algorithm 3 OpenMP mean shift algorithm

```
procedure OMPMEANSHIFT($original\_points$)
    $shifted\_points \leftarrow original\_points$
    while $iteration < N\_ITERATIONS$ do
        #pragma omp parallel for
        for $point$ in $shifted\_points$ do
            $point \leftarrow$ SHIFTPOINT($point, original\_points$)
```

Please note that in the openMP Algorithm 3 the SHIFTPOINT($point, original\_points$) function is the same as in the sequential version (see Algorithm 2).

2.3. Parallel version with CUDA

CUDA (Compute Unified Device Architecture) [8] is a parallel computing platform and application programming interface (API) model created by Nvidia.

In this section we will look at some details of the parallel implementation of the proposed algorithm that takes advantage of the parallelism of the GPU. There are two variants of mean shift algorithm implemented on CUDA: the first one is the naive version, while the second one is a more optimized version that makes use of shared memory.

In the GPU, threads are organized in blocks and blocks are organized in grids. Each thread within a thread block executes an instance of the kernel and has a thread ID and also every thread block has a block ID within its grid.

In our implementation we used a vector to store points, so both the blocks and the grid require only the x coordi-

[1] The computational complexity of mean shift can be written as $O(Tn^2)$, where T is the number of iterations and n is the number of data points. The most computationally expensive component of the mean shift procedure corresponds to identifying the neighbors of a point in a space (as defined by the kernel and its bandwidth). This problem is well-known as multidimensional range searching [2].

nate to be defined. As we can note from the pseudocode in Algorithm 4 and Algorithm 6 the block dimension is equal to a constant called $BLOCK\_DIM$ and the grid dimension (that corresponds to the number of blocks) is given by $\lceil |orig\_pts|/BLOCK\_DIM \rceil$, where the $|\cdot|$ symbol indicates the number of points of the given vector. These values defines the execution configuration of the cuda kernel.

An important aspect to underline is that in our implementation the points are stored in memory as a Structure of Arrays (SOA) (in contrast to Arrays of Structures): this allows to have contiguous ("coalesced") memory access in order to achieve high memory bandwidth. Doing so the 3d points are stored in memory as shown in equation 2.

$$[x_1, x_2, ...x_n, y_1, ...y_n, z_1, ...z_n] \tag{2}$$

2.3.1 CUDA naive implementation

In Algorithms 4 and 5 are shown a wrapper function needed to define the kernel execution configuration and to launch the kernel and in the second procedure the CUDA kernel responsible for the mean shift algorithm. The wrapper function is also necessary to call the CUDA kernel a number of times specified in the $N\_ITERATIONS$ constant.

In the pseudocode BW stands for bandwidth.

Algorithm 4 CUDA naive mean shift algorithm

```
procedure CUDAMeanShift($orig\_points, b\_dim$)
    $shifted\_points \leftarrow orig\_points$
    $b\_dim \leftarrow b\_dim$
    $blocks \leftarrow \lceil |orig\_pts|/BLOCK\_DIM \rceil$
    while $iteration < N\_ITERATIONS$ do
        NaiveKernel<<< $blocks, b\_dim$ >>>($sp, op$)
        cudaDeviceSynchronize()
```

Algorithm 5 CUDA naive kernel

```
procedure NaiveKernel($shifted\_pts, orig\_pts$)
    $idx \leftarrow threadIdx.x + blockIdx.x * blockDim.x$
    if $idx < |orig\_pts|$ then
        $sp \leftarrow shifted\_pts[idx]$
        $tot\_weight \leftarrow 0$
        $new\_spt \leftarrow 0$
        for $op$ in $orig\_pts$ do
            $dist \leftarrow$ ComputeDistance($op, sp$)
            $weight \leftarrow$ Kernel($dist, BW$)
            $new\_spt \leftarrow new\_spt + op * weight$
            $tot\_weight \leftarrow tot\_weight + weight$
        $shifted\_pts[idx] \leftarrow new\_spt/tot\_weight$
```

2.3.2 CUDA shared memory: tiling

A more efficient implementation of the cuda kernel can be obtained obtimizing the procedure in order to limit the memory accesses, this can be done making use of the shared memory [7].

Algorithm 6 CUDA mean shift algorithm with tiling

```
procedure CUDATilingMS($orig\_points, b\_dim$)
    $shifted\_points \leftarrow orig\_points$
    $b\_dim \leftarrow BLOCK\_DIM$
    $blocks \leftarrow \lceil |orig\_pts|/b\_dim \rceil$
    while $iteration < N\_ITERATIONS$ do
        TilingKernel<<< $blocks, b\_dim$ >>>($sp, op$)
        cudaDeviceSynchronize()
```

Algorithm 7 CUDA kernel with tiling

```
procedure TilingKernel($shifted\_pts, orig\_pts$)
    $tx \leftarrow threadIdx.x$
    $idx \leftarrow threadIdx.x + blockIdx.x * blockDim.x$
    $tile \leftarrow shared\_mem\_array[TILE\_WIDTH]$
    $tot\_weight \leftarrow 0$
    $new\_spt \leftarrow 0$
    $n\_tiles \leftarrow (|orig\_pts| - 1)/(TILE\_WIDTH + 1)$
    for $tile\_i$ in $n\_tiles$ do        ▷ Load data from global memory into shared memory
        $tile\_idx \leftarrow tile\_i * TILE\_WIDTH + tx$
        if $tile\_idx < |orig\_pts|$ then
            $tile[tx] \leftarrow orig\_pts[tile\_idx]$
        else
            $tile[tx] \leftarrow null\_pt$
        $\_\_syncthreads()$        ▷ Wait for loading end
        if $idx < |orig\_pts|$ then
            $sp \leftarrow shifted\_pts[idx]$
            for $i$ in $TILE\_WIDTH$ do
                $op \leftarrow tile[i]$
                $dist \leftarrow$ ComputeDistance($op, sp$)
                $weight \leftarrow$ Kernel($dist, BW$)
                $new\_spt \leftarrow new\_spt + op * weight$
                $tot\_weight \leftarrow tot\_weight + weight$
    $\_\_syncthreads()$        ▷ Wait for computation end
    if $idx < |orig\_pts|$ then
        $shifted\_pts[idx] \leftarrow new\_spt/tot\_weight$
```

There are several levels of memory on the GPU device, each with distinct characteristics.

Every thread in a thread block has access to a unified *shared memory* (in addition to the private 'local' memory of the single thread) that is shared among all threads for the life of that thread block. Moreover, all threads have access to the *global memory*, which is located off-chip.

There is a natural tradeoff in the use of device memories in CUDA: the global memory is large but slow, whereas the shared memory is small but fast. A common optimization strategy is to partition the data into subsets called *tiles* so that each tile fits into the shared memory [5] and use this memory as a cache memory so that the cost of reading from global memory can be amortized.

Using this strategy the number of accesses to the global memory for each thread are reduced from $O(n)$ to $O(n/TILE\_WIDTH)$.

In Algorithm 6 and Algorithm 7 are show the wrapper function and the cuda kernel with the tiling pattern respectively.

In particular in the Algorithm 7 we can point out some interesting aspects: first of all we can see the algorithm as consisting of different parts: during the *loading phase* each thread loads into the shared memory the point which was assigned to, and then during the *computational phase* a partial shift is computed based only on the points contained in the current tile. This process must be repeated many times, until all the points have been loaded and used to compute the shift.

We have two important indexes used in the algorithm: $tile\_idx$ indicates the index of the point that has to be loaded into the shared memory, while idx represents the index of the shifting point which the thread was assigned to, that is computed as $idx \leftarrow threadIdx.x + blockIdx.x * blockDim.x$.

At the end of the loading phase and the computational phase there are two barriers ($\_\_syncthreads()$), in the first case this assures that all threads have loaded the data into shared memory, while in the second case the barrier guarantees that all the threads have finished using the data in the shared memory (the partial shift has been computed).

Finally during the loading phase a *boundary check* is performed: if a thread is to load a point that is not in the valid index range it loads a dummy point, that will not affect the final output value.

3. Experiments

We carried out several experiments to test the different versions of the implemented mean shift algorithm.

For all our experiments this was the setting:

- We used variable dataset size with 3d points (each point $x_i \in R^d, d = 3$)
- $BANDWIDTH$ fixed to 2
- $N\_ITERATIONS$ set to 10 (enough to ensure the convergence of all the points to the cluster mode)[2]
- Experiments time results are the average over 15 runs of the algorithm (except for the sequential and openMP version with 100,000 points where the average has been computed over 3 runs of the algorithm because of the long execution time (the sequential version of the algorithm takes about six hours)).

All the tests have been performed on a machine equipped with:

- CPU: Intel(R) Core(TM) i7-860 @ 2.80GHz, with 4 cores/ 8 threads
- GPU: NVidia GeForce GTX 980, 4 GB
- CUDA 10.1
- OpenMP 4.5

3.1. Speedup

The speedup metric has been used to compare the differences in the performances of the different implementations, where the **speedup** is defined as the ratio between the completion time of the sequential algorithm t_S and the completion time of the parallel algorithm t_P:

$$S = \frac{t_S}{t_P}$$

3.2. Datasets

The datasets used to evaluate the different implementations were generated with the $datasets.make\_blobs$ function of sklearn [10]. We generated gaussian distributions with 3 centers and stadard deviation equal to 1.5. The datasets we generated to test the algorithm have dimesions equal to: 100 1,000 10,000 100,000 500,000 1,000,000 (the last two values have been used only for CUDA experiments, see Table 5).

4. Results

Below are shown the results obtained for the various tests carried out, let's see them in detail.

4.1. Speedup: sequential over openMP

First of all to evaluate the performances of the OpenMP implementation some tests were performed on each dataset with an increasing number of threads (from two to eight threads).

The results and the speedup obtained compared to the sequential version for different dataset sizes are shown in Tables 1, 2, 3 and 4. Furthermore in Figure 1 we can see

[2] In the mean shift algorithm the *number of iterations* to reach the algorithm convergence depends on the bandwidth value, however to compare the performances more objectively, the value of the number of iterations has been fixed.

Number of points	100		
Algorithm version	**Threads**	**Time**	**Speedup**
Sequential		0.0232 s	
OpenMP	2	0.0118 s	1.97
	3	0.0087 s	2.65
	4	0.0065 s	3.53
	5	0.0063 s	3.68
	6	0.0059 s	3.92
	7	0.0064 s	3.62
	8	0.0085 s	2.71

Table 1. Speedup. Experiments done with 100 points with three dimensions. Mean values over 15 runs of the algorithm.

Number of points	1,000		
Algorithm version	**Threads**	**Time**	**Speedup**
Sequential		2.0954 s	
OpenMP	2	1.0973 s	1.91
	3	0.8075 s	2.60
	4	0.6048 s	3.46
	5	0.5713 s	3.67
	6	0.5752 s	3.64
	7	0.5168 s	4.05
	8	0.5099 s	4.11

Table 2. Speedup. Experiments done with 1000 points with three dimensions. Mean values over 15 runs of the algorithm.

the speedup plot for all the datasets tested as the number of threads increases. Here we can notice that the speedup is sub-linear (almost linear) from two threads to four threads, while from five to eight threads the growth rate of the speedup decreases a lot.

We can also notice that the speedup trend is similar for all the dataset dimensions except for the smallest one: 100 points; for this one the plot shows that after six threads the speedup decreases: this means that for such a little dataset the overhead of the threads takes longer than how much we gain from multithreading.

Seeing the overall results we can say that using the parallelization given by openMP allows us to reach speedup values up to 4, and this is a good result compared to the little difficulty of adding a single openmp directive to the sequential implementation.

4.2. Cuda tiling: tile width

The graph in Figure 2 shows the time (in seconds) for mean shift execution implemented using shared memory in CUDA as the tile width varies. For this test we used the dataset with 100,000 points.

The minimum time value was obtained with tile width set to 128, so this is the setting which ensures the fastest mean shift execution.

We can also notice that a small tile width could undersize active wraps, while an high value for the tile width could limits the overall shared memory available. Values around the one considered (32, 64, 128 and 256) are however good values for the size of the tile.

This value for the tile width size has been used for all the subsequent experiments.

Number of points	10,000		
Algorithm version	**Threads**	**Time**	**Speedup**
Sequential		205.168 s	
OpenMP	2	107.327 s	1.91
	3	77.2562 s	2.65
	4	72.8408 s	2.81
	5	67.1869 s	3.05
	6	57.0997 s	3.59
	7	52.3503 s	3.92
	8	50.2230 s	4.09

Table 3. Speedup. Experiments done with 10,000 points with three dimensions. Mean values over 15 runs of the algorithm.

Number of points	100,000		
Algorithm version	**Threads**	**Time**	**Speedup**
Sequential		21266.10 s	
OpenMP	2	10726.50 s	1.98
	3	7910.51 s	2.69
	4	5945.48 s	3.58
	5	5981.84 s	3.55
	6	5521.53 s	3.85
	7	5068.40 s	4.19
	8	4970.49 s	4.28

Table 4. Sequential version in comparison with parallel openMP version. Speedup. Experiments done with 100,000 points with three dimensions. Mean values over 3 runs of the algorithm.

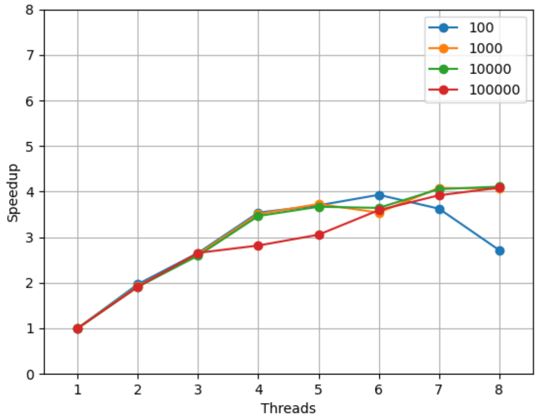

Figure 1. Speedup of the parallel algorithm with openMP over the sequential algorithm for different datasets varying the number of threads.

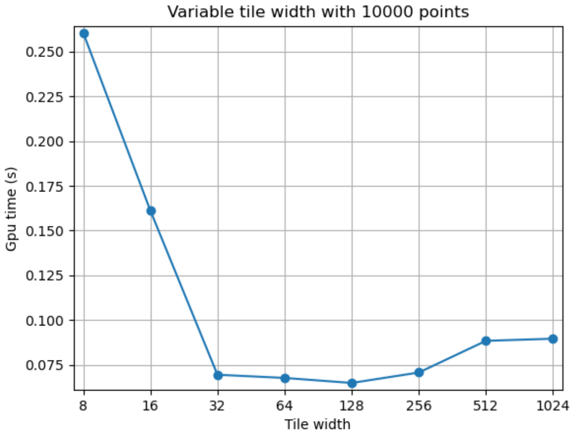

Figure 2. GPU execution time (in seconds) varying the tile width.

4.3. CUDA speedup

Now we present some experiments carried out to compare the different CUDA implentations and to compare these with the sequential one.

Table 5 shows the execution time of the CUDA naive implementation, the execution time of the second version with tiling and the speedup obtained. Both the CUDA naive and the CUDA tiling versions were executed on all the generated datasets.

Table 6 shows the execution time of the sequential version, the CUDA version with tiling and the speedup obtained. As we can note here the speedups are really high and the speedup increases as the number of data increases.

Finally, in Figure 3 are shown the speedup obtained with the CUDA tiling version varying the tile width and so the number of threads in the block, as expected the highest speedup is given by the central values:32, 64 and 128, with a peak at 128.

Moreover if we compare the values of speedup obtained with the CUDA version with tiling to the speedup obtained with openMP we can notice that in the first case the values are much higher. So, as expected, the cuda implementation outperforms both sequential and openMP implementation, at the expense of a more complicated implementation.

5. Conclusions

In this project we analyzed the mean shift algorithm in a sequential version and two parallel version: the first that exploits the CPU parallelism and the other the GPU parallelism. The embarrassingly parallel structure of the mean shift algorithm presented makes it particularly suitable for parallel computing.

Dataset dimension	**CUDA naive**	**CUDA tiling**	**Speedup**
100	0.00059 s	0.00060 s	0.98
1,000	0.00474 s	0.00442 s	1.07
10,000	0.05315 s	0.05131 s	1.04
100,000	5.39592 s	4.84162 s	1.11
500,000	132.575 s	118.647 s	1.12
1,000,000	530.107 s	472.190 s	1.12

Table 5. CUDA naive version in comparison with CUDA tiling version with tile width set to 128. The results reported are the average over 15 runs.

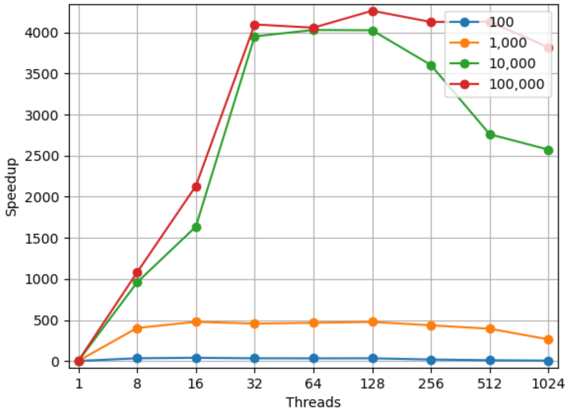

Figure 3. Speedup CUDA tiling over sequential implementation varying the number of threads in the block, for all the datasets.

Dataset dimension	**Sequential**	**CUDA tiling**	**Speedup**
100	0.02322	0.00060 s	38.7
1,000	2.0954 s	0.00442 s	474.072
10,000	205.168 s	0.05131 s	3998.60
100,000	21266.10 s	4.84162 s	4392.35

Table 6. Sequential version in comparison with cuda tiling version with tile width set to 128. The results reported are the average over 15 runs, except for sequential time with 100,000 points that is the mean result over 3 runs.

The parallel implementation with OpenMP allows us to obtain a speedup equal to more than 4, while with the parallel CUDA implementation we reached about 4400 speedup. These results showed how the use of the GPU makes this algorithm applicable to datasets that are intractable with a CPU.

6. Resources

The code is publicly available on GitHub: `https://github.com/francidellungo/Mean_Shift`

References

[1] M. Á. Carreira-Perpiñán. A review of mean-shift algorithms for clustering. *CoRR*, abs/1503.00687, 2015.

[2] D. Demirovic. An implementation of the mean shift algorithm. *Image Processing On Line*, 2019.

[3] K. G. Derpanis. Mean shift clustering, 2005.

[4] K. Fukunaga and L. Hostetler. The estimation of the gradient of a density function, with applications in pattern recognition. *IEEE Transactions on Information Theory*, 21(1):32–40, 1975.

[5] D. B. Kirk and W. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.

[6] J. C. Niebles and F.-F. Li. K-means and mean-shift clustering, 2016. Stanford University Computer Vision course: Foundations and Applications.

[7] Nvidia. *CUDA C++ Programming Guide, Design Guide*. 2021.

[8] NVIDIA, P. Vingelmann, and F. H. Fitzek. Cuda, release: 10.2.89, 2020.

[9] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008.

[10] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[11] S. Weglarczyk. Kernel density estimation and its application. *ITM Web of Conferences*, 23(37), 2018.