OPTIMIZATION OF DATA VALUATION FOR K-NEAREST NEIGHBOR ALGORITHMS

Anqi Li, Chunxiao Wu, Yu-Shan Wei, Zhentao Liu

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

Data valuation, the process of distributing values to data points, becomes essential due to data commoditization. This paper proposes various optimizations for two data valuation algorithms which can compute Shapley Values (SV) for K-nearest neighbor (KNN) classifiers. Experiments show that the optimized implementation of both algorithms has achieved significant speed-up.

1. INTRODUCTION

Motivation. Data valuation is the process of assigning values to data points. In this paper, we focus on two algorithms designed to solve the following data valuation problem: given a budget X, a dataset D, and a machine learning model M to train over D, how should we value each data point in D while respecting X? In the current era where data becomes an increasingly valuable commodity, such algorithms are extremely useful [1]. It is crucial to produce fast implementations of them since real-world data valuation problems usually involve large-scale datasets. However, high performance is difficult to achieve during implementation, as these algorithms frequently invoke expensive operations such as sorting and data movement.

Contribution. This paper proposes multiple optimizations to speed up the baseline implementation of two data valuation algorithms that compute the Shapley Values (SV) for K-nearest neighbor (KNN) classifiers. The optimized implementation is proved to outperform the baseline implementation when running on a popular dataset in machine learning research.

Related work. SV is one of the most common ways of revenue sharing. It defines a unique profit allocation scheme that has many desirable properties such as fairness, rationality, and decentralizability [1]. Many researches focus on deriving efficient algorithms to compute SV efficiently in big data setting [2, 3, 4]. In contrast, our work concentrates on writing efficient code that implements some selected algorithms for SV computation.

2. BACKGROUND ON THE ALGORITHM

This section describes two algorithms that compute the SVs for KNN classifiers from [1] and the input dataset used to run the algorithms. We have also performed a cost analysis for each of them, with the cost measure being the total number of floating point operations (flops) in the algorithms. Note that we only count the operations that constitute the mathematical algorithm.

Input data. The algorithms take a training data set and a testing data set as input. The training data set is an $N_{train} * F$ matrix, while the testing data set is an an $N_{test} * F$ matrix. The training and testing data used to run the algorithms come from the CIFAR-10 dataset, where CIFAR stands for Canadian Institute For Advanced Research. It consists of images commonly used in the training of machine learning algorithms. In our project, we focus on training and testing data sets that have a large fixed width F, which equals the number of pixels in one row of the images in CIFAR-10. It is notable that F could be as big as 3072.

Algorithm 1 Exact algorithm for calculating the SV for an unweighted KNN classifier

```
Input: $Train$: training data set, an $N_{train} * F$ matrix;
    $Test$: testing data set, an $N_{test} * F$ matrix.
Output: $SV$: an $N_{train} * F$ matrix.
 1: for i=1 to $N_{test}$ do                                   ▷ Loop1
 2:     for j=1 to $N_{train}$ do                              ▷ Loop2
 3:         $Dis_{i,j}$ = GetDistance($Test_i, Train_j$)
 4:     end for
 5:     $Idx_i$ = Sort($Dis_i$)
 6:     for j=$N_{train} - 1$ to 1 do                          ▷ Loop3
 7:         $TmpSV_{i,j}$ = ComputeTmpSV($Idx_i, Test_i, Train_j$)
 8:     end for
 9: end for
10: for i=1 to $N_{train}$ do                                  ▷ ComputeSV
11:     $SV_i = \frac{1}{N_{test}} * \sum_{N_{test}}^{j=1} TmpSV_{j,i}$
12: end for
```

Algorithm 1. This algorithm computes the exact SV for KNN classifiers. It maintains the KNN by sorting distances.

Its pseudo-code is shown above.

Cost analysis of Algorithm 1.

- GetDistance: the cost is C * N_{train} * N_{test} flops in total, where $C = O(F)$ represents the number of flops for one distance computation.
- ComputeTmpSV: the cost is about T * $N_{test} * (N_{train} - 1)$ flops in total, where the constant T represents the number of flops for one partial SV computation.
- ComputeSV: the cost is $N_{train} * (N_{test} + 1)$ flops in total.
- Computational complexity: $O(FN_{train} \log N_{train} N_{test})$ [1].

Note that T is determined by the function that defines the utility of the KNN classifier. The utility function that we use fixes T to 4, which is much smaller than any realistic C. Thereby, the optimization bottleneck is distance computation.

Algorithm 2. This algorithm computes the approximated SV for KNN classifiers by using permutations. Unlike Algorithm 1, this algorithm uses a max-heap to maintain the KNN. Its pseudo-code is as follows:

Algorithm 2 Improved MC Approach for calculating the SV for an unweighted KNN classifier

```
Input: $Train$: training data set, an $N_{train} * F$ matrix;
    $Test$: testing data set, an $N_{test} * F$ matrix; $N_p$: the
    number of permutations.
Output: $SV$: an $N_{train} * F$ matrix.
 1: for i=1 to $N_p$ do
 2:     $Train$ = permute($Train$)
 3:     for i=1 to $N_{test}$ do heap = CreateHeap()
 4:         for j=1 to $N_{train}$ do
 5:             $Dis_{i,j}$ = GetDistance($Test_i$, $Train_j$)
 6:             HeapInsert(heap, $Dis_{i,j}$) ($Train_i$)
 7:             if heap changed then
 8:                 $\phi_{i,j,t}$ = ComputeDiff($Dis_{i,j}$)
 9:             else
10:                 $\phi_{i,j,t}$ = 0
11:             end if
12:         end for
13:     end for
14:     $Sp_t$ = ComputeSP($\phi_t$)
15: end for
16: $SV$ = ComputeSV($Sp$)
```

Cost analysis of Algorithm 2.

- GetDistance: the cost is at most C * N_{train} * N_{test} * N_p flops in total, where $C = O(F)$ represents the number of flops for one distance computation.
- ComputeSP: the cost is N_p * N_{train} * (N_{test} + 1) flops in total. Here we do the average computation to get the SP values.
- ComputeSV: the cost is $N_{train} * (N_p + 1)$ flops in total. Here we do the average computation to get the SV values.
- Computational complexity: $O(FN_pN_{test}N_{train} \log K)$ [1].

In the input dataset CIFAR-10, F is some large constant, and N_{train} is much bigger than N_{test}. As a result, the optimization bottleneck of Algorithm 2 is also distance computation.

3. THE OPTIMIZATION METHODS

In this section, we demonstrate the optimizations that we have performed for both algorithms from Section 2. We assume that the F is divisible by 4.

3.1. Optimization of Algorithm 1

Baseline implementation. The baseline implementation of Algorithm 1 strictly follows its pseudo-code shown in Section 2. More specifically, matrices are implemented with double pointers, and vectors are implemented with pointers. Moreover, Sort(Dis_i) is implemented with bubble sort, while GetDistance($Test_i$, $Train_j$) is implemented as Subroutine 1.

Subroutine 1 GetDistance(v_1, v_2)

```
Input: Two vectors $v_1$ and $v_2$ of the same length
Output: The distance between $v_1$ and $v_2$
1: sum = 0
2: for i = 1 to $length(v_1)$ do
3:     sum += $(v_1[i] - v_2[i]) * (v_1[i] - v_2[i])$
4: end for
5: dist = $\sqrt{sum}$
6: return dist
```

Optimization method 1: merge sort. Benchmarking of the baseline implementation shows that sorting takes up a large portion of the total runtime. While bubble sort is easy to implement, its time complexity can be up to $O(n^2)$, where n is the size of the input array. In comparison, the time complexity of merge sort is as low as $O(n \log n)$ on all kinds of input. As a result, we replaced bubble sort with merge sort to speed up the implementation.

Optimization method 2: vectorization. Observe that in Subroutine 1, $\forall i, j \in [1, length(v_1)] \wedge i \neq j$, computing $v_1[i]-v_2[i]$ and $v_1[j]-v_2[j]$ requires us to perform the same instruction on different data, which makes it possible for us to exploit data-level parallelism. Therefore, we substituted line 3 of Subroutine 1 with equivalent code written in Single Instruction/Multiple Data (SIMD) fashion using Advanced Vector Extensions 2 (AVX2).

Optimization method 3: loop unrolling. Notice that in Algorithm 1, for every distinct i, the same vector $Test_i$ needs to be loaded for N_{train} times to compute all $Dist_{i,j}$'s where $j \in [1, N_{train}]$. To reduce the reloading of $Test_i$, we unrolled *Loop2* four times. In this way, we can compute the distances between $Test_i$ and four consecutive training data vectors in each iteration of *Loop2*. Consequently, for every distinct i, $Test_i$ is only loaded for $\frac{N_{train}}{4}$ times to compute all $Dist_{i,j}$'s, and the four distances can be stored using a single AVX2 instruction in each iteration of *Loop2*.

Loop unrolling can also be applied to the loop in Subroutine 1. Since addition is associative, we could use separate accumulators to compute partial sums, which allows us to complete more useful work per iteration. After performing some experiments, we decided to unroll this loop with 2 different accumulators.

Note that unrolling *Loop3* is not possible since the utility function used by our implementation causes data dependency in ComputeTmpSV.

Optimization method 4: fused multiply-add. After loop unrolling, it becomes obvious that multiplications and additions are frequently used in the implementation. Thereby, we merged multiplications with additions by using fused multiply-add (FMA) instructions wherever possible.

Optimization method 5: simplified distance comparison. A helpful insight is that we do not need to compute the precise distances between vectors in order to sort them in an ascending order. Assume without loss of generality that the widths F of testing and training data matrices are 3. Given a testing data vector $t_1 = (x, y, z)$ and two training data vectors $t_2 = (a_1, b_1, c_1)$, $t_3 = (a_2, b_2, c_2)$, we have the following assumption:

$$\begin{gathered}\text{GetDistance}(t_1, t_2) > \text{GetDistance}(t_1, t_3) \\ \Leftrightarrow (x - a_1)^2 + (y - b_1)^2 + (z - c_1)^2 > \\ (x - a_2)^2 + (y - b_2)^2 + (z - c_2)^2 \\ \Leftrightarrow a_1^2 + b_1^2 + c_1^2 - 2(a_1 x + b_1 y + c_1 z) > \\ a_2^2 + b_2^2 + c_2^2 - 2(a_2 x + b_2 y + c_2 z).\end{gathered}$$

Hence, comparing $a_1^2 + b_1^2 + c_1^2 - 2(a_1 x + b_1 y + c_1 z)$ with $a_2^2 + b_2^2 + c_2^2 - 2(a_2 x + b_2 y + c_2 z)$ is equivalent to comparing the distances.

Moreover, observe that the terms $a_1^2 + b_1^2 + c_1^2$ and $a_2^2 + b_2^2 + c_2^2$ only involve training data, so they can be computed before entering *Loop1*. As a result, inside *Loop2*, instead of computing the distances, we only compute the terms $2(a_1 x + b_1 y + c_1 z)$ and $2(a_2 x + b_2 y + c_2 z)$. This reduces the total number of arithmetic operations of the algorithm, thus reducing the runtime.

3.2. Optimization of Algorithm 2

Baseline implementation. The baseline implementation of Algorithm 2 strictly follows its pseudo-code shown in Section 2. Matrices, vectors, and the max-heap are implemented using double pointers, pointers and structs respectively.

Optimization method 1: GetDistance vectorization. The cost analysis in Section 2 indicates that distance computation is the optimization bottleneck, so we started the optimization by vectorizing such computation. Moreover, loop unrolling is adopted to improve spatial locality of the code. The number of unrolling was obtained via experiments. Furthermore, FMA instructions were applied wherever possible. Additionally, we performed scalar replacement and rewrote the code in ”read-compute-store” style.

Optimization method 2: ComputeSP vectorization. The optimization performed on ComputeSP is similar to the optimization method above. As shown by the cost analysis, ComputeSP has a much lower cost than GetDistance when the input dataset has a large F. Nevertheless, optimizing ComputeSP is still necessary, as this will become very useful when F is small. We will discuss this in more details in Section 4.

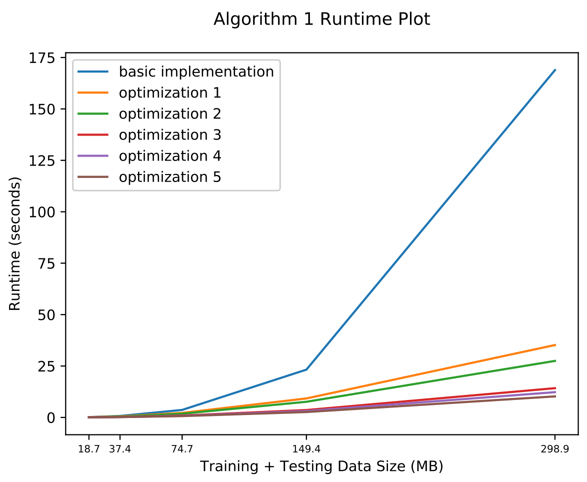
Optimization method 3: distance precomputation. An important observation is that the distance between a training data vector and a testing data vector is not affected by permutations. Instead of performing distance computations after each permutation, we could compute and store the distances between each training data vector and each testing data vector in advance. In this way, after each permutation, the desired distances can be retrieved from the precomputed distances using the permuted indices.

Optimization method 4: data structure, function inlining, and strength reduction. Several optimization techniques are combined to form optimization method 4. Firstly, we replaced the struct with an index array and a value array to implement the max-heap. Secondly, function inlining was applied, as the original code invoked many functions calls associated with the heap. This allows us to remove some unnecessary operations. For instance, now we no longer need to update a flag to represent whether the heap changes or not. Lastly, divisions were substituted with multiplications wherever possible.

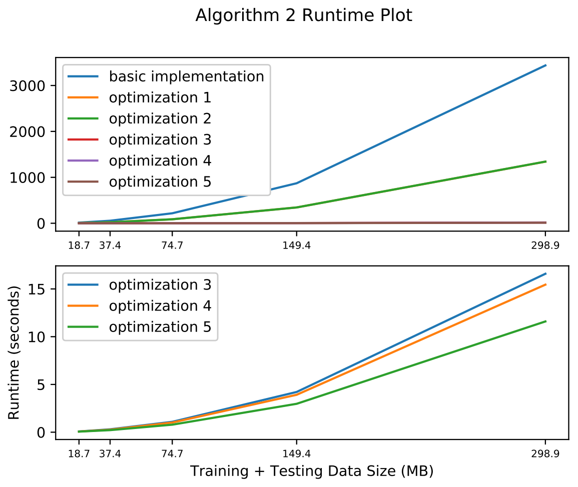
Optimization method 5: simplified distance comparison. Notice that in Algorithm 2, distance comparison is still required to maintain the max-heap. Hence, the optimization method 5 used in Algorithm 1 can be applied to Algorithm 2 as well.

4. EXPERIMENTAL RESULTS

We have conducted a number of experiments to measure the effect of the proposed optimization methods. In the follow-



(a) Runtime Comparison for Algorithm 1 with K=1.



(b) Runtime Comparison for Algorithm 2 with K=5 and N_p=100. To get a clear view of the runtime plot for Optimizations 3-5, please refer to the diagram at the bottom.

Fig. 1. Runtime Comparison for Algorithms 1 & 2.

ing paragraphs, we illustrate experiment procedure, experiment setup, input data used and the optimization results for Algorithms 1 and 2.

Procedures of the experiments. For each algorithm, we have developed implementations of different optimization levels in C. We use the term *Optimization i* to refer to the implementation incorporating optimization methods 1 to i from Section 3. In the experiments, the baseline implementation is run first, followed by all optimized implementations. While we run each implementation for at least 10 iterations, the Time Stamp Counter (TSC) is invoked to record the total runtime. Between each iteration, we do not manually clean up the cache, but as the input data used in the experiments always has a bigger size than the cache, our experiments are equivalent to those that are done with a cold cache. Next, we divide the total runtime by the number of iterations to compute the average runtime of each implementation. Finally, a comparison is made on the average runtime of all implementations to derive the optimization results.

Experimental setup. We use gcc 9.3.0 to compile all code with flags $-mavx2, -march = native, -lm$. Afterwards, the compiled code is executed on a computer with the following specification with Turbo Boost disabled:

- Processor: Intel Core i7-10710U
- Number of cores: 6
- Microarchitecture: Comet Lake
- Processor base frequency: 1.10 GHz
- L1 cache: 32KB per core, 8-way set associative
- L2 cache: 256KB per core, 4-way set associative
- L3 cache: 12MB shared, 16-way set associative
- Maximum memory bandwidth: 45.8 GB/s

Input data. Input data of different sizes is used in the experiments. We change the input data size by changing N_{test}, i.e. the amount of rows taken from the testing data batch in CIFAR-10. Since we fix the ratio of testing data size to training data size to 1:189, N_{train}, i.e. the amount of rows taken from the training data batch changes as N_{test} changes.

Implementation	Runtime (Cycles)	Speed-up
Baseline implementation	$1.86 * 10^{11}$	1
Optimization 1	$3.87 * 10^{10}$	4.80
Optimization 2	$3.02 * 10^{10}$	6.14
Optimization 3	$1.57 * 10^{10}$	11.86
Optimization 4	$1.35 * 10^{10}$	13.74
Optimization 5	$1.13 * 10^{10}$	16.50

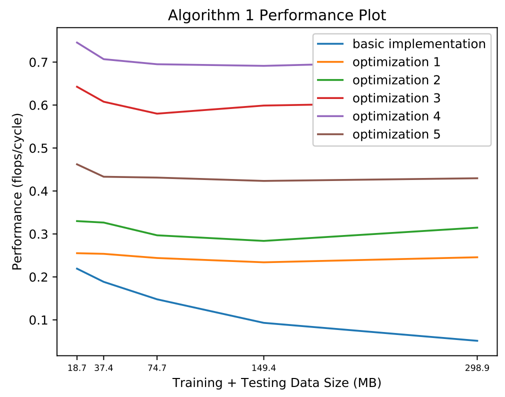
Table 1. Runtime Speed-Up of Algorithm 1 with K=1.

Implementation	Runtime (Cycles)	Speed-up
Baseline implementation	$3.78 * 10^{12}$	1
Optimization 1	$1.48 * 10^{12}$	2.58
Optimization 2	$1.48 * 10^{12}$	2.58
Optimization 3	$1.82 * 10^{10}$	207.39
Optimization 4	$1.70 * 10^{10}$	220.42
Optimization 5	$1.28 * 10^{10}$	297.17

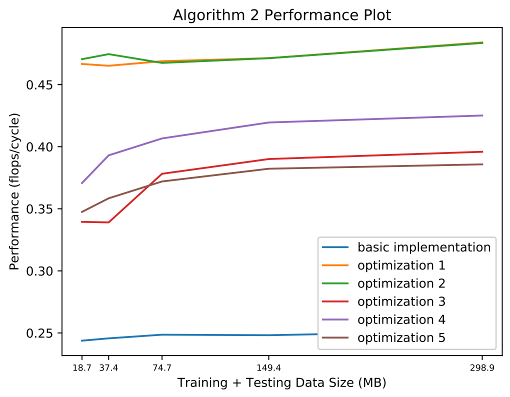
Table 2. Runtime Speed-Up of Algorithm 2 with K=5 and N_p=100.

Result 1 of optimizations: Runtime. Figures 1(a) and 1(b) show the runtime differences among all implementations of Algorithms 1 and 2 respectively. We can easily observe that the more implementation methods used, the lower the runtime gets. For both algorithms, Optimization 5, i.e. the implementation incorporating all proposed optimizations, has the lowest runtime on all input sizes. Moreover, when the input size is large, its runtime is substantially lower than the runtime of the baseline implementation, which indicates the effectiveness of the proposed optimization methods.

Tables 1 and 2 summarize the runtime speed-up after optimization for both algorithms on some fixed input. The input data size is around 300 MB, composed of 298.2 MB of training data and 15.7 MB of testing data. The data in the tables suggest that both algorithms, especially Algorithm 2, are much faster after optimization.



(a) Performance Plot of Algorithm 1 with K=1.

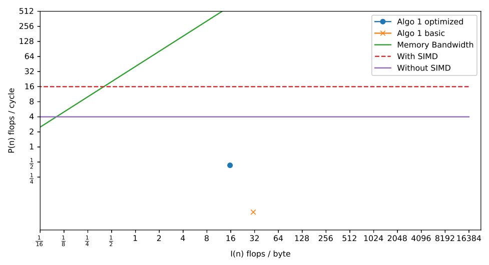


(b) Performance Plot of Algorithm 2 with K=5 and N_p=100.

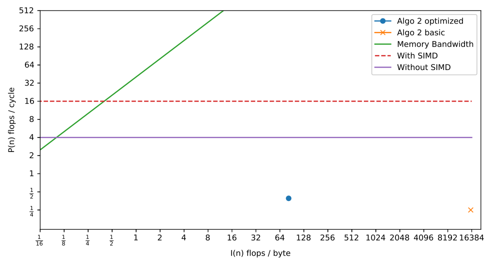
Fig. 2. Performance Plots of Algorithms 1 & 2.

Result 2 of optimizations: performance plot. Figures 2(a) and 2(b) are the performance plots for all implementations of Algorithms 1 and 2 respectively. For Optimizations 1-4 of Algorithm 1, the performance increases as the optimization level increases. Notwithstanding, Optimization 5, the implementation that has the lowest runtime, does not have the highest performance. The reason is that by using the simplified distance comparison algorithm, the number of flops in Optimization 5 is lower than that in Optimizations 1 to 4. The decrease in the number of flops is greater than the decrease in the runtime, which makes the performance of Optimization 5 lower than some other optimizations. Due to the same reason, for Algorithm 2, Optimizations 3 and 4 have lower performance than Optimizations 1 and 2, and Optimization 5 has lower performance than Optimization 4.

Result 3 of optimizations: Roofline plot. Figures 3(a) and 3(b) are the Roofline plots for Algorithms 1 and 2 respectively. For both algorithms, only the baseline implementation and the final optimized implementation are marked on the diagram. We can easily conclude that all 4 implementations are compute bound. Notice that after optimization, the dot representing the final optimized implementation moves in upper-left direction in both diagrams, meaning that the optimized code is closer to the theoretical performance upper-bound, which is consistent with our expectation.



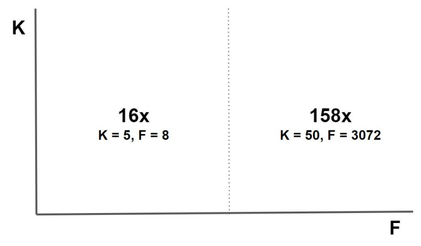
(a) Roofline Plot of Algorithm 1 with K=1.



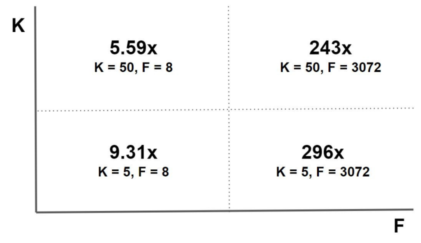
(b) Roofline Plot of Algorithm 2 with K=5 and N_p=100.

Fig. 3. Roofline Plots of Algorithm 1 & 2.

Further discussion. In the experiments above, the width F of the input data matrices is fixed to some big constant, while K in KNN is fixed to some small constant. What happens if we change the values of F and K? We summarized the influence of different values of K and F on our imple-



(a) Speed-Ups of Optimization 5 given Different F and K for Algorithm 1.



(b) Speed-Ups of Optimization 5 given Different F and K for Algorithm 2 with $N_p = 100$.

Fig. 4. Speed-Ups of Optimization 5 given Different F and K for Algorithms 1 & 2.

mentation speed-up in Figure 4.

When F changes from a big value to a small one, the amount of improvement caused by each optimization becomes different. For example, when $F = 3072, K = 5$, Optimizations 1, 3, 4, 5 of Algorithm 1 and Optimizations 1, 3, 5 of Algorithm 2 result in significant runtime reduction. When $F = 8, K = 5$, Optimization 2 of Algorithm 1 and Optimizations 2, 4 of Algorithm 2 result in significant runtime reduction. This is because when F gets smaller, the optimization bottleneck shifts from distance computation to KNN maintenance.

Similarly, we analyze the impact of different values of K as follows. When F is fixed to 8 and K changes from 5 to 50, the optimization that causes significant runtime reduction changes from Optimization 2 to Optimizations 2, 4 for Algorithm 2. This is because when K is large, the heap needs to maintain more elements and thus more heap operations are invoked. Unlike Algorithm 2, no significant difference could be observed for Algorithm 1 as K increases, because K does not affect the run-time of sorting all elements.

5. CONCLUSIONS AND FUTURE WORK

This paper has proposed multiple optimization methods to speed up the implementation of two algorithms that compute SV for KNN classifiers. The proposed methods include SIMD, standard C optimization, sorting algorithms and so on. Significant reduction in runtime has been achieved for both algorithms.

In the future, we may run more comprehensive experiments to improve our work. Currently, even the smallest input data used in our experiments is larger than 10MB. Feeding smaller input to our implementation might help us understand the relation between cache size and the performance of our code, and therefore brings us new optimization ideas. Another possible improvement is to experiment with datasets of different data distribution. For example, with sparse arrays, is it possible to implement distance computation differently? In this case, many computations might be simplified.

6. CONTRIBUTIONS OF TEAM MEMBERS

Anqi Li. Implemented baseline implementation of Algorithm 1, vectorization and loop unrolling for Algorithm 1. Created performance and runtime plots.

Chunxiao Wu. Implemented baseline implementation of Algorithm 1, 2 and all optimization methods for Algorithm 2. Performed cost analysis and bottleneck analysis on Algorithms 1 and 2.

Yu-shan Wei. Implemented baseline implementation of Algorithm 1, vectorization, loop unrolling, FMA, simplified distance comparison for Algorithm 1. Created Roofline plots.

Zhentao Liu. Implemented baseline implementation of Algorithm 1, 2 and all optimization methods for Algorithm 2. Implemented benchmark that measures the runtime and performance of Algorithms 1 and 2.

7. REFERENCES

[1] Ruoxi Jia, David Dao, Boxin Wang, Frances Ann Hubis, Nezihe Merve Gurel, Bo Li, Ce Zhang, Costas J. Spanos, and Dawn Song, "Efficient task-specific data valuation for nearest neighbor algorithms," 2019.

[2] Ruoxi Jia, David Dao, Boxin Wang, Frances Ann Hubis, Nick Hynes, Nezihe Merve Gürel, Bo Li, Ce Zhang, Dawn Song, and Costas J. Spanos, "Towards efficient data valuation based on the shapley value," in *Proceedings of the Twenty-Second International Conference on Artificial Intelligence and Statistics*, Kamalika Chaudhuri and Masashi Sugiyama, Eds. 16–18 Apr 2019, vol. 89 of *Proceedings of Machine Learning Research*, pp. 1167–1176, PMLR.

[3] Jörg Bremer and Michael Sonnenschein, "Estimating

shapley values for fair profit distribution in power planning smart grid coalitions," pp. 208–221, 2013.

[4] Prasang Upadhyaya, Magdalena Balazinska, and Dan Suciu, "How to price shared optimizations in the cloud," 2012.