# Computational Statistics - Accelerating kNN

Andrew Struthers, Nathan Chapman, Nick Haviland

March 2023

***Honor code: We pledge that we have neither given nor received help from anyone other than the instructor or the TAs for all work components included here. – Andrew, Nate, Nick***

## Introduction

The purpose of this project is to accelerate a k-Nearest-Neighbor (kNN) approximated probability density function. We will be focusing on the $k^{th}$ nearest neighbor estimator for the density function generated from the machine learning training data provided. Because we have discrete points in the dataset, we don't have a clear density function. What we will be doing, instead of trying to generate a continuous density function, is creating a *probability density estimation*. We can build an estimation of the density function from the observed data by calculating the Information Energy for each unique point in the provided dataset, which relies on using the kNN algorithm.

We were given code that accomplished this task in R, but due to some of the limits of R as a high-performance programming language, in addition to the computationally expensive nature of the kNN algorithm, the code provided has lengthy execution time. We were tasked with figuring out where the R code execution time could be improved via profiling, then creating faster subroutines in C++. We will be profiling the R code, then creating two functions in C++, with the function names $kNN()$ and $IE\_xy()$, to replace the two provided R functions. We will use the Rcpp package inside of R, which allows us to use a specific wrapper in C++ that easily integrates into R. The Rcpp package will allow us to compile and dynamically link our C++ code to the R code at runtime, reducing the complexity of the bridge between both languages. For simplicity, we will only be exposing the $IE\_xy()$ function to R, so that the usability from the R side is very simple and straightforward.

## R Code Analysis

Upon analysis of the R code provided, one of the first things we noticed is between the $IEx\_y()$ function and the $kNN()$ function, the two for loops guarantee at least $O(n^2)$ runtime due to nested $for$-loops. The sort algorithm in the $kNN()$ function, and the $unique()$ function in the $IEx\_y()$ function also undoubtedly add to the runtime of the algorithm, since R is inherently slower at sorting and parsing large datasets compared to optimized low-level language paridigms. Our first thoughts on how to optimize this brought us to vectorization. With vectorization, we can improve the runtime of any operation on the provided data vectors, including time improvements when sorting and selecting the unique subset of data. We also noticed that there were some memory inefficiencies in the R code, such as storing the individual Information Energy and weight values into an array, just to sum them up at the end. There is no need to store these values in an array, and because operations on arrays are much more computationally expensive than a single operation on a float value, we could see performance gains by removing the unnecessary operations.

## C Functions

Our code includes two functions written in C++ to decrease the time required for time intensive routines. Both functions rely on the Rcpp package which provides integration of R and C++, while making it so that the C++ functions look and work similarly to their R counterparts. For the functions, all calculations are vectorized due to the performance increase vectorization offers. The speed performance as a result of using vectors, as well as managing our memory by cutting down on the number of arrays relative to the provided code, we were able to substantially decrease the execution time of this calculation heavy estimation from 174.59 seconds to 32.57 seconds while maintaining a sum of percent errors of only $1.2923\text{E}^{-4}$ from the results generated from the provided R code, which could easily be explained by floating point arithmetic rounding errors.

One of the C++ functions is named $\text{IE}_{xy}$ and is the only function exposed to R at runtime. This functions allows us to pass in two vectors of floats, one for the x-values and one for the y-values, named $\text{data}_x$ and $\text{data}_y$ respectively. Additionally an integer value k is passed in to be used in the $\text{k}^{th}$ nearest neighbor calculations. The two vectors are cast from cast from arrays in R to vectors in C++ to allows for substantially faster operations on large data sets. This function first finds a vector of all unique elements in the y vector and iterates through this subset. Each iteration, we grab the subset of the x vector where every x element in the subset corresponds to the unique y value we are using in this iteration. We then calculate the mean conditional information energy using the kNN approximation function and we calculate a weight to this specific subset of data, then add the product of those two values to our overall result.

The second function, kNN takes in a vector of floats, data, and an integer k. The data vector is then sorted with std::sort to sort the elements by their absolute distance from the $\text{i}^{th}$ element in the data. A vector is then returned which contains the kNN weights for each element of the vector data.

## Results

In Figure 1 below, we can see the results of plotting the resulting vector of the $IE\_xy$ using the Mathematics Self-Efficacy as x and Mathematics Intentions as y plotted against values of k from 5,000 to 20,000 is 2,500 increments. From the plot we can see that at the lowest values, 5,000 and 7,500, the conditional information energy is much higher than at the other values of k, at 0.2917 and 0.1934 respectively. This value remains the same along the other data points at 0.1368. This indicates that lower values of k may lead to a more accurate model.

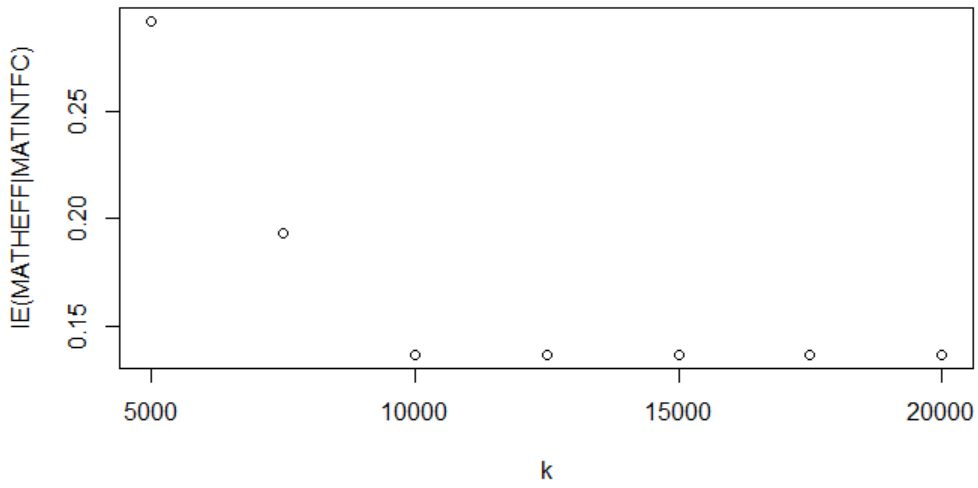


Figure 1: $IE\_xy$ function with Mathematics Self-Efficacy as x and Mathematics Intentions as y plotted against various values of k.

Figure 2 below contains a table which details the data obtained to form Figure 1. From the table below, we can see that the highest resulting error comes from the outputs when k is between 10,000 and 20,000 at 2.0771E-05 each. We feel that this is small enough to be attributed to a round-off error.

| | Provided Code Output | Our Code Output | Difference | Percent Error |
|---|---|---|---|---|
| a | 0.291676582839643 | 0.291676610708237 | 2.7868594E-08 | 9.5546217E-06 |
| b | 0.193382410660318 | 0.193382382392883 | 2.8267435E-08 | 1.4617376E-05 |
| c | 0.136791749398995 | 0.136791720986366 | 2.8412629E-08 | 2.0770718E-05 |
| d | 0.136791749398995 | 0.136791720986366 | 2.8412629E-08 | 2.0770718E-05 |
| e | 0.136791749398995 | 0.136791720986366 | 2.8412629E-08 | 2.0770718E-05 |
| f | 0.136791749398995 | 0.136791720986366 | 2.8412629E-08 | 2.0770718E-05 |
| g | 0.136791749398995 | 0.136791720986366 | 2.8412629E-08 | 2.0770718E-05 |

Figure 2: Table showing difference in outputs for Figure 1 between provided code and C++ implemented code.

In Figure 3 below, we can see a table with the resulting time required for the calculations between the provided code and the code with the C++ functions. The elapsed time for the provided code is 174.59 seconds, while our code with C++ implementation has a run time of 32.57 seconds. Our code runs in 18.655% of the time required for R to generate the same results.

| Provided | User | System | Elapsed |
|---|---|---|---|
| (seconds) | 171.19 | 3.42 | 174.59 |
| Ours | User | System | Elapsed |
| (seconds) | 31.48 | 1.09 | 32.57 |

Figure 3: Table showing the difference in time-cost between provided code and C++ implemented code.

## Conclusion

The fundamental quantity of interest in this investigation is Information Energy (IE). But before we can calculate the IE, we must first find the underlying probability density function (PDF). This PDF can be computationally expensive to calculate, but thankfully can be well approximated by a $k$-nearest-neighbors (kNN) clustering algorithm.

If the kNN is implemented in the R programming language, the performance of the algorithm will be impaired. This is because the straight-forward implementation of the kNN will use nested for-loops (resulting in polynomially increasing runtime), native sorting algorithms, and native uniqueness algorithms. For small datasets, these are fine and there is no need for premature optimization. When it comes to the datasets considered here, there are too many points for the procedure to run in a reasonable amount time (let alone if there were more points). Therefore we move to implementing the "workhorse subroutines" in a high-performance language like C++, and then call those functions from within the R runtime.

If the kNN is implemented in C++, we can take advantage of several things to increase the runtime performance of the algorithm. The most notable of these advantages are vectorization and overall "quickness" of C++. Vectorization in C++ gives the ability to do calculations on whole vectors using hardware accelerated functionality instead of looping of each element in a vector; even further performance gains could be obtained using the relatively new tensor cores on GPUS. The second advantage of using a language like C++ is that the language itself is overall just quicker when compared to R and Python.

If the latter is done, the performance benefits of C++ can be utilized while still having the "friendliness" of R by calling the routines using a package such as Rcpp. This package in particular allows the source code to be written very similar to how one would write native R code. Similar functionality can be attained by using the native R functions to call external C/C++ functions.

We find the information energy has a generally inverse-exponential dependency on the distance of the nearest neighbor quantity $k$; decreasing quickly for low $k$ and asymptotically approaching zero for greater $k$. Quantitatively, the IE decreases by about 47% between $k = 5 \times 10^3$ and $k = 10^4$. The error between the R results and our custom results is centered around $10^-5$ for low $k$ and saturates at around $2 \times 10^5$ for larger $k$. On these magnitudes, we conclude our results are well within the range of reasonable accuracy and the errors are attributed to round-off.

Overall, **the implementation of the "work horse" algorithms in C++ decreased the run time by a factor of five** as compared to the time needed using only R. This performance gain could be even further increased by implementing and running the calculations in parallel on a multi-threaded system or, for even greater still, by using GPU computing like CUDA for a massive increase in performance. Other possibilities for performance increase might come from moving away from using R at all.

## References


[1] Ben Ogorek. Three ways to call c/c++ from r: R-bloggers. *R*, Feb 2014.

[2] Hadley Wickham, Chapman, and Hall. Advanced r. *25 Rewriting R code in C++ — Advanced R*, 2019.

[3] Dirk Eddelbuettel. Rcpp package. *RDocumentation*, 2023.

[4] Xuejun Liang, Ali A. Humos, and Tzusheng Pei. Vectorization and parallelization of loops in c/c++ code. *Department of Computer Science, Jackson State University, Jackson, MS, USA*, 2017.

[5] Raymond Cheng. Vectorization: Must-know technique to speed up operations 100x faster. *Medium*, May 2022.

[6] Angel Caţaron, Răzvan Andonie, and Yvonne Chueh. knn estimation of the unilateral dependency measure between random variables. In *2014 IEEE Symposium on Computational Intelligence and Data Mining (CIDM)*, pages 471–478. IEEE, 2014.