ESANN 2014 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning. Bruges (Belgium), 23-25 April 2014, i6doc.com publ., ISBN 978-287419095-7.
Available from http://www.i6doc.com/fr/livre/?GCOI=28001100432440.

Speedy Greedy Feature Selection: Better Redshift Estimation via Massive Parallelism

Fabian Gieseke[1], Kai Lars Polsterer[2], Cosmin Eugen Oancea[1], and Christian Igel[1]

1- University of Copenhagen - Department of Computer Science
Universitetsparken 5, 2100 Copenhagen - Denmark

2- Heidelberg Institute for Theoretical Studies gGmbH - Astroinformatics
Schloß-Wolfsbrunnenweg 35, 69118 Heidelberg - Germany

Abstract. Nearest neighbor models are among the most basic tools in machine learning, and recent work has demonstrated their effectiveness in the field of astronomy. The performance of these models crucially depends on the underlying metric, and in particular on the selection of a meaningful subset of informative features. The feature selection is task-dependent and usually very time-consuming. In this work, we propose an efficient parallel implementation of incremental feature selection for nearest neighbor models utilizing nowadays graphics processing units. Our framework provides significant computational speed-ups over its sequential single-core competitor of up to two orders of magnitude. We demonstrate the applicability of the overall scheme on one of the most challenging tasks in astronomy: redshift estimation for distant galaxies.

1 Motivation

Astronomy is nowadays a data-rich science. Current projects such as the *Sloan Digital Sky Survey* [1] contain terabytes of data about hundreds of millions astronomical objects. Upcoming projects to be launched within the next few years will gather such data volumes *per night* [2] yielding total data volumes in the peta- and exabyte range. Naturally, such large-scale settings render a manual data analysis impossible and machine learning techniques will become crucial tools in this field [3]. *Nearest neighbor models* are one of the simplest yet often effective tools in machine learning [4]. Learning scenarios well-suited for this type of models are usually given if the input space dimension is low and a large amount of training patterns is available. This is precisely the situation given for astronomical surveys with millions of training objects.

In recent studies, nearest neighbor methods have successfully been applied to various astronomical learning tasks including the detection of distant galaxies [5] and the estimation of specific physical properties [6]. These models can already yield a superior performance over other learning schemes using standard features given in the astrophysical catalogs. In addition, as shown by Polsterer *et al.* [7], one can even further improve the models' quality by performing a feature selection step in the training phase. The involved computations can, however, be very time-consuming and have to be repeated for each new task.

In this work, we provide a parallel implementation for incremental feature selection given nearest neighbor models, which is tailored towards the specific

ESANN 2014 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning. Bruges (Belgium), 23-25 April 2014, i6doc.com publ., ISBN 978-287419095-7.
Available from http://www.i6doc.com/fr/livre/?GCOI=28001100432440.

needs of nowadays *graphics processing units* (GPUs). Our framework can effectively take advantage of the massive computational resources provided by these devices yielding speed-ups of up to two orders of magnitude over its single-core variant. We demonstrate the practical merits of our scheme in the context of redshift estimation models for distant galaxies given photometric data.

2 Nearest Neighbors and Feature Selection

We focus on regression with training sets $T = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\} \subset \mathbb{R}^d \times \mathbb{R}$. A nearest neighbor model is based on the labels given in the training set, which are averaged for the k nearest neighbors of a given query object $\mathbf{q} \in \mathbb{R}^d$ via $h(\mathbf{q}) = \sum_{\mathbf{x}_i \in N_k(\mathbf{q})} y_i$. Here, N_k denotes the set of indices of the k patterns that are closest to $\mathbf{q}$, where "closeness" is defined via an arbitrary metric (we use the Euclidean one in this work). The features (i.e., the components of $\mathbf{q}$) usually vary w.r.t. their "expressiveness", and using only a subset of features can improve the quality of the model h. Therefore, *feature selection* [4] is employed for choosing informative features. Selecting an optimal subset of fixed cardinality $f < d$ can, for instance, be accomplished in a brute-force manner by evaluating all possible subsets using the cross-validation (CV) error [4]. However, such an approach scales badly and quickly becomes computationally intractable. Standard alternatives are *forward* and *backward feature selection*, which try to select the best-performing features in a greedy manner [4]. Still, even given training sets of moderate sizes, these greedy methods can be very time-consuming.

One way to accelerate such a feature selection step is to speed up the involved nearest neighbor computations. A variety of schemes has been proposed over the last decades for this task. Typical techniques are *k-d trees* [8] or *locality-sensitive hashing* [9]. However, such tools either perform poorly in higher dimensions or only yield approximated answers. A recent trend is to make use of (exact) parallel implementations for many-core devices. For instance, Garcia *et al.* [10] utilize highly-tuned GPU matrix multiplication libraries for nearest neighbor search. Other schemes are based on, e.g., adapted spatial search structures [11, 12, 13], see, e.g., Cayton [11] and the references therein. However, to our knowledge, no parallel implementation that addresses incremental feature selection and nearest neighbor models *simultaneously* is available for GPUs.

3 Algorithmic Framework

Our framework aims at an efficient parallel implementation of feature selection for nearest neighbor models (using cross-validation errors to select "informative" dimensions). We will concentrate on the case of forward selection; all derivations provided can, however, easily be adapted to the case of backward selection.

3.1 General Workflow

The workflow of our approach is outlined in Algorithm 1: First, the matrix $\mathbf{M} \in \mathbb{R}^{n \times n}$ containing the squared distances, the array `selected_dimensions` indicat-

ESANN 2014 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning. Bruges (Belgium), 23-25 April 2014, i6doc.com publ., ISBN 978-287419095-7.
Available from http://www.i6doc.com/fr/livre/?GCOI=28001100432440.

Algorithm 1 IncrementalFeatureSelection(T, f)

Require: A training set $T = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\} \subset \mathbb{R}^d \times \mathbb{R}$ and a number $f < d$.
Ensure: An array `selected_dimensions` containing the selected features.
1: Initialize empty (squared) distance matrix $\mathbf{M} \in \mathbb{R}^{n \times n}$; {GPU, in parallel}
2: `int selected_dimensions`$[d] = \{0, \ldots, 0\}$; `float val_errors`$[d]$;
3: **for** $i = 1, \ldots, f$ **do**
4: `val_errors` = GetValidationErrors($\mathbf{M}$); {GPU, in parallel}
5: $i_{\min}$ = GetMinDim(`val_errors`);
6: `selected_dimensions`$[i_{\min}] = 1$;
7: $\mathbf{M} = \mathbf{M} + \mathbf{M}^{i_{\min}}$; {GPU, in parallel}
8: **end for**
9: **return** `selected_dimensions`

ing the selected features, and the array `val_errors` are initialized. The iterative feature selection starts in Step 3: GetValidationErrors computes, for each dimension j that has not yet been selected (i.e., `selected_dimensions`$[j]$=0), the CV error for the case of dimension j being "added" to the current set of features (see below). These values are stored in the array `val_errors`, and GetMinDim returns the index of the smallest error contained in it (thus, $i_{\min}$ corresponds to the dimension whose addition leads to the smallest CV error). Finally, both `selected_dimensions` and $\mathbf{M}$ are updated accordingly, where $\mathbf{M}^{i_{\min}}$ denotes the all-pairs (squared) distance matrix based on dimension $i_{\min}$ only.

3.2 Parallel Implementation for Many-Core Systems

The procedure GetValidationErrors consumes most of the runtime. We now describe this component as well as details of our GPU implementation.

Computing Validation Errors. The layout of GetValidationErrors is shown in Algorithm 2: The procedure computes, for each dimension j not yet selected, the matrix $\widehat{\mathbf{M}} = \mathbf{M} + \mathbf{M}^j$ containing all pairwise distances with dimension j being "added". The intermediate matrix is then used to obtain the desired CV error via ComputeCVError.[1] Finally, the array `val_errors` is returned.

Many-Core Implementation. The most significant part of the overall work of Algorithm 1 takes place on the GPU (Steps 1, 4, and 7), while the CPU is only used for synchronizations and for updating `selected_dimensions` (Steps 2, 5, and 6). The initialization and the update of $\mathbf{M}$ are trivially parallelizable on the GPU and exhibit a negligible runtime. As such, we focus on the call GetValidationErrors($\mathbf{M}$). We make use of the following three techniques:

(1) *Aggressive fusion* between the kernels that produce and consume the same arrays. For example, $\widehat{\mathbf{M}}$ and $\mathbf{M}^j$ are not materialized in memory. Instead,

[1] Our implementation can efficiently compute the cross-validation errors for a whole range $k \in \{k_1, \ldots, k_{\max}\}$ of model parameters, basically at the same cost as for $k_{\max}$ only (by initially computing the k_{max} nearest neighbors and by reusing them for each $k_i < k_{max}$).

ESANN 2014 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning. Bruges (Belgium), 23-25 April 2014, i6doc.com publ., ISBN 978-287419095-7.
Available from http://www.i6doc.com/fr/livre/?GCOI=28001100432440.

Algorithm 2 GetValidationErrors($\mathbf{M}$)

Require: A distance matrix $\mathbf{M} \in \mathbb{R}^{n \times n}$.
Ensure: An array `val_errors` containing the cross-validation errors.
1: **for** $j = 1, \ldots, d$ **do**
2: **if** `selected_dimensions`$[j] = 0$ **then**
3: $\widehat{\mathbf{M}} = \mathbf{M} + \mathbf{M}^j$; {GPU, in parallel}
4: `val_errors`$[j]$ = ComputeCVError($\widehat{\mathbf{M}}$); {GPU, in parallel}
5: **end if**
6: **end for**
7: **return** `val_errors`

each thread computes the corresponding element of $\widehat{\mathbf{M}}$ "on the fly" by resorting to both $\mathbf{M}$ and T that are kept (and updated) in global memory.

(2) *Enhancing the locality of reference* by ensuring that the access to the matrix $\mathbf{M}$ and to the training patterns given in T are effectively supported by *coalescing* and by *caching* (i.e., all threads in a local work group access the *same* memory element per instruction), respectively.

(3) *Increasing the degree of parallelism* by a factor of about d, which is achieved by parallelizing the outer loop of Algorithm 2. This yields an efficient execution of the previous steps, which would otherwise fail to hide latency.

The overall implementation for Algorithm 2 invokes $\bar{d}n$ threads, where $\bar{d}$ denotes the number of dimensions not yet selected (`selected_dimensions`$[j] = 0$). Each thread, being associated with a training pattern and a dimension j, computes the $k_{\max}$ nearest neighbors of the pattern given in the *remaining* folds (while "adding" dimension j on the fly; for instance, using 10-fold CV, neighbors in 9 folds have to be computed). Note that this enables the effective cached access to all training patterns, see technique (2). Given the nearest neighbors (stored in global memory), the induced predictions are obtained using another kernel (for each pattern and each k value). This yields the desired CV errors.

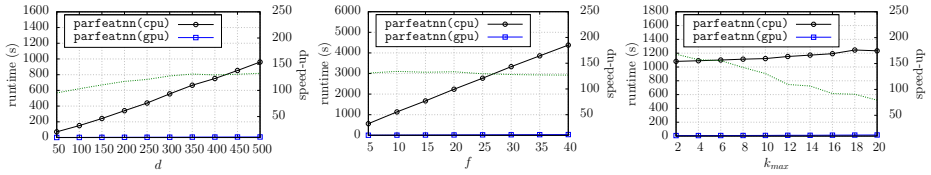
4 Experiments

4.1 Experimental Setup

The experiments were conducted on a standard computer with an `Intel(R) Core(TM) i7-3770` CPU at 3.40GHz (4 cores, one is used), 16GB RAM, and a `GeForce GTX 770` GPU with 1536 shader units and 4GB RAM. The operating system was `Ubuntu 12.04` (64 Bit). We focused on a comparison between the sequential (`parfeatnn(cpu)`) and the many-core (`parfeatnn(gpu)`) implementation, which were implemented in `C` and `OpenCL` (using `gcc-4.6` and `Swig`).

We used a challenging astronomical regression task for the runtime experiments, namely photometric redshift estimation for distant galaxies. The data stem from the Sloan Digital Sky Survey [1]. Instead of resorting to the standard

ESANN 2014 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning. Bruges (Belgium), 23-25 April 2014, i6doc.com publ., ISBN 978-287419095-7.
Available from http://www.i6doc.com/fr/livre/?GCOI=28001100432440.



(a) (b) (c)

Figure 1: Runtime behavior w.r.t. (a) the dimension d, (b) the number of selected features f, and (c) the maximum number $k_{\max}$ of nearest neighbors. Speed-ups of `parfeatnn(gpu)` over `parfeatnn(cpu)` are given in each plot (dotted line).

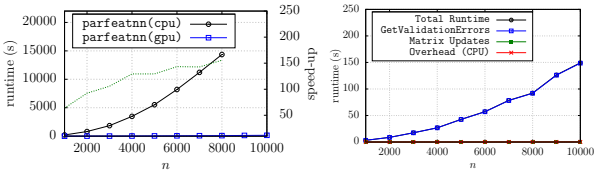
features commonly considered in this setting [5], we used an augmented set of features induced by all difference combinations of selected raw features (similar to Polsterer *et al.* [7]). The final training set consisted of $n = 10,000$ patterns each having $d = 585$ features.[2] We would like to point out that similar speed-ups have been observed on several other data sets as well.

4.2 Runtime Results

If not stated otherwise, we considered $k \in \{2, 5, 10\}$ as model grid, conducted a 10-fold cross-validation, and fixed the number of selected features to $f = 10$.

Parameters: d, f, and k. We analyzed the influence of several problem-dependent parameters using $n = 5,000$ patterns. In Fig. 1, the runtime dependencies on d, f, and k are shown for varying assignments (we used k_{max} as single model parameter). While the general runtime behavior was the same for both schemes, the GPU implementation needed significantly less time. Further, the obtained speed-ups were roughly independent on the values of d and f, while the runtime gain slightly decreased for larger assignments of k (keeping track of more nearest neighbors leads to an increase of branch divergence, which is, in turn, expensive on many-core devices).

Speed-Ups & Overhead. We subsequently investigated the performance gains for various training set sizes n, see Fig. 2 (a) for the case of $f = 50$ selected features. It can be seen that the GPU implementation yielded a significant speed-up (dotted line) over its single-core competitor (>150). The runtimes of the different phases of `parfeatnn(gpu)` are shown in Fig. 2 (b). Clearly, the most significant part of the runtime was spent on computing the CV errors (Algorithm 2); the other phases contributed less than 1% to the runtime.



(a) (b)

Figure 2: (a) Speed-Ups and (b) Overhead

[2] An in-depth description of this learning task will be provided in a follow-up work.

ESANN 2014 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning. Bruges (Belgium), 23-25 April 2014, i6doc.com publ., ISBN 978-287419095-7.
Available from http://www.i6doc.com/fr/livre/?GCOI=28001100432440.

5 Applications: Redshift Estimation and Beyond

We applied the proposed algorithm to the problem of redshift estimation for galaxies [5, 7]. The feature selection significantly reduced the CV error, see Fig. 3: The plot shows the $\mathrm{RMS}(\Delta_{\mathrm{norm}})$ [5] for an increasing number f of selected features. Our GPU implementation computed these results in less than 3 minutes (instead of more than 6 hours). This demonstrates its potential for such tasks, especially for (but not restricted to) astronomy.

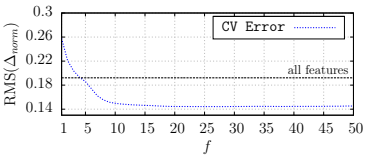


Figure 3: Redshift Estimation

Acknowledgements. FG acknowledges support from the *German Academic Exchange Service* and CI from *The Danish Council for Independent Research* through the project *Surveying the sky using machine learning* (SkyML).

References

[1] D. G. York et al. The Sloan digital sky survey: Technical summary. *The Astronomical Journal*, 120(3):1579–1587, 2000.

[2] Z. Ivezic, J. A. Tyson, E. Acosta, R. Allsman, et al. LSST: From science drivers to reference design and anticipated data products. *arXiv/0805.2366v2*, 2011.

[3] K. Borne. Scientific data mining in astronomy. In *Next Generation of Data Mining*, pages 91–114. Chapman and Hall/CRC, 2009.

[4] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer, 2 edition, 2009.

[5] K. L Polsterer, P. Zinn, and F. Gieseke. Finding new high-redshift quasars by asking the neighbours. *Monthly Notices of the Royal Astronomical Society*, 428(1):226–235, 2013.

[6] K. Stensbo-Smidt, C. Igel, A. Zirm, and K. Steenstrup Pedersen. Nearest neighbour regression outperforms model-based prediction of specific star formation rate. In *Proc. of the 2013 IEEE International Conference on Big Data*, pages 141–144. IEEE, 2013.

[7] K. L. Polsterer, F. Gieseke, C. Igel, and T. Goto. Improving the performance of photometric regression models via massive parallel feature selection. In *Proceedings of the 23rd Annual Astronomical Data Analysis Software & Systems Conference*, 2013. In press.

[8] J. L Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[9] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 604–613. ACM, 1998.

[10] V. Garcia, E. Debreuve, F. Nielsen, and M. Barlaud. K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching. In *Proceedings of the 17th IEEE International Conference on Image Processing*, pages 3757–3760. IEEE, 2010.

[11] L. Cayton. Accelerating nearest neighbor search on manycore systems. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 402–413. IEEE, 2012.

[12] J. Heinermann, O. Kramer, K. L. Polsterer, and F. Gieseke. On GPU-based nearest neighbor queries for large-scale photometric catalogs in astronomy. In *KI 2013: Advances in Artificial Intelligence*, pages 86–97. Springer, 2013.

[13] N. Nakasato. Implementation of a parallel tree method on a GPU. *Journal of Computational Science*, 3(3):132–141, 2012.