Massively-Parallel Best Subset Selection for Ordinary Least-Squares Regression

Fabian Gieseke* Kai Lars Polsterer† Ashish Mahabal‡ Christian Igel* Tom Heskes§
*Department of Computer Science, University of Copenhagen, Copenhagen, Denmark
E-mail: fabian.gieseke@di.ku.dk, igel@diku.dk
†Astroinformatics Group, Heidelberg Institute for Theoretical Studies, Heidelberg, Germany
E-mail: kai.polsterer@h-its.org
‡Center for Data-Driven Discovery, California Institute of Technology, Pasadena, CA
E-mail: aam@astro.caltech.edu
§Institute for Computing and Information Sciences, Radboud University Nijmegen, Nijmegen, The Netherlands
E-mail: t.heskes@cs.ru.nl

***Abstract*—Selecting an optimal subset of k out of d features for linear regression models given n training instances is often considered intractable for feature spaces with hundreds or thousands of dimensions. We propose an efficient massively-parallel implementation for selecting such optimal feature subsets in a brute-force fashion for small k. By exploiting the enormous compute power provided by modern parallel devices such as graphics processing units, it can deal with thousands of input dimensions even using standard commodity hardware only. We evaluate the practical runtime using artificial datasets and sketch the applicability of our framework in the context of astronomy.**

I. Introduction

Feature selection is an important step in data analysis [1]. In contrast to using all available features, choosing an "informative" subset has several advantages: The resulting models are easier to interpret, higher prediction performance may be achieved by discarding noisy features, and execution time is reduced during the application phase. Various selection techniques and models have been proposed in the literature. Two prominent lines of techniques are *wrappers* and *implicit methods*. Here, wrapper-based schemes can be applied to basically arbitrary models and select good features in, e.g., an incremental manner. For implicit methods, feature selection is part of the underlying model fitting process (e.g., *random forests* select good features at each node split) [2].

In this work, we consider the task of selecting an optimal set of k out of d features for linear regression models given n training instances. Especially for high-dimensional learning tasks with $d \gg n$, it is often desirable to obtain sparse models that are based on a small subset of features. The so-called *best subset selection* problem addresses this goal [3]. Unfortunately, the induced optimization problem is NP-hard and computationally very demanding even given moderate problem sizes [4]. However, due to the importance of the problem, various approaches have been developed over the past decades that aim at tackling the combinatorial nature of the problem.

A trivial approach to solving the underlying optimization problem is to test all possible feature subsets of cardinality up to k in a brute-force fashion. While this results in a computationally intractable task for large d and large k, it is worth pointing out that it is actually possible to solve the problem for the special case of relatively small k. While this restricts the general problem definition, it is actually often desired in practice to select only a very small number k of features, in particular for visualization and model understanding.

In this work, we describe an efficient implementation for this brute-force approach. In particular, we show how to gain computational efficiency by precomputing auxiliary matrices and demonstrate the drastic runtime reduction by using modern massively-parallel devices. Our many-core implementation is based on a simple yet very effective way to compute solutions for the intermediate optimization problems per thread. The overall framework allows for the computation of exact solutions in seconds or minutes for the case of large d, large n, and small k. In particular, using our implementation, one can easily solve problem scenarios with $n = 20,000$ instances and $d = 5,000$ or $d = 1,000$ in less than 100 seconds for $k = 3$ and $k = 4$, respectively. We investigate the runtime behavior of the implementation as well as the general applicability of the approach via artificial datasets and real-world data from the field of astronomy.

II. Background

We start by describing the problem more formally. Since our approach also resorts to massively-parallel computing, we will briefly sketch key concepts from that field as well.

A. Best Subset Selection

We consider regression scenarios with training sets of the form $T = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\} \subset \mathbb{R}^d \times \mathbb{R}$. Standard ordinary least-squares solves

$$\underset{\beta \in \mathbb{R}^d}{\text{minimize}} \; \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|_2^2 \tag{1}$$

with $\mathbf{y} \in \mathbb{R}^{n \times 1}$ containing the labels, $\mathbf{X} \in \mathbb{R}^{n \times d}$ containing the patterns as rows, and $\boldsymbol{\beta} \in \mathbb{R}^d$ containing the model coeffi-

978-1-5386-2726-6/17/$31.00 ©2017 IEEE

cients that need to be computed [2].[1] The *best subset selection problem* corresponds to adding an additional constraint that enforces sparse solutions with few non-zero coefficients:

$$\begin{aligned} \underset{\boldsymbol{\beta}\in\mathbb{R}^d}{\text{minimize}} \quad & \|\mathbf{y}-\mathbf{X}\boldsymbol{\beta}\|_2^2 \\ \text{s.t.} \quad & \|\boldsymbol{\beta}\|_0 \leq k \end{aligned} \tag{2}$$

Here, $\|\boldsymbol{\beta}\|_0 = |\{j \mid \beta_j \neq 0\}|$ is the L0 pseudo-norm and the corresponding constraint enforces sparse solutions with at most k non-zero model coefficients [3]. The considered norm is arguably the most intuitive sparsity measure and also exhibits several valuable properties from a statistical perspective. The induced sparse models also permit a direct interpretation of the results.

The underlying optimization problem is of combinatorial nature due to the additional constraint and it has been shown that solving the problem exactly is NP-hard [4]. Prominent alternatives that are usually considered instead are based on replacing the L0 pseudo-norm by norms that yield easier optimization tasks such as the L1 or L2 norm (usually, an additional regularization term is added to the objective). Due to its popularity, the best subset selection problem has gained considerable attention over the past decades. The most prominent implementation might be the `leaps` implementation, provided as an `R` package.[2] However, it does not scale for high dimensions (e.g., $d \gg 30$) [5]. In recent years, various attempts have been made to obtain exact and/or approximate answers in a reasonable amount of time. For a detailed discussion of recent optimization techniques, we refer to recent surveys and work in this field [5], [6], [7].

We address the problem at hand in a simple brute-force fashion. While more sophisticated search schemes have been proposed in the literature, such as the use of mixed-integer programming solvers [5] or branch-and-bound techniques [7], none of the existing approaches is asymptotically faster than this brute-force approach due to the problem being NP-hard. The goal of this work is to provide an efficient way to solve the problem exactly for small k such as $k = 3$ or $k = 4$. Even for such cases, the naïve brute-force approach can quickly become extremely time-consuming given high-dimensional data and/or many training instances. As we will show below, the use of massively-parallel devices can dramatically reduce the practical runtime of the induced exhaustive search. The implementation provided in this work might also pave the way for speeding up more sophisticated search strategies such as branch-and-bound [7].

B. Massively-Parallel Computing

Massively-parallel computing architectures such as modern graphics processing units (GPUs) have become a common tool to speed up general computations. Such devices nowadays contain thousands of "simple" compute units. In contrast, standard multi-core architectures resort to only a small number of more complex compute units (e.g., up to 16). Graphics processing units offer enormous computational resources and various data mining and machine learning approaches have already been adapted according to such platforms (see, e.g., [8], [9], [10], [11], [12]).

While being computationally very appealing, one usually has to adapt the approaches' workflows to make them amenable to a massively-parallel execution: A CPU is generally based on more complex control units and mechanisms that are optimized for a sequential code execution. In contrast, a GPU resorts to simpler control units, which are optimized for a massively-parallel execution [13]: All threads are executed in parallel via the *single instruction multiple data*-paradigm, which describes the fact that all threads of a given *warp* (group of 32 threads) can only execute the same instruction in a single clock cycle, but can access different locations in the main memory of the GPU. From a high-level perspective, such implementations aim at conducting the "inexpensive" part on the host system (i.e., via the CPU), whereas the compute intensive parts are conducted via the GPU.

To derive an efficient parallel implementation for a GPU, two algorithmic concepts are crucial: Firstly, one needs to expose sufficient parallelism to the device meaning that it must be possible to split up the compute intensive parts into a large number of small subtasks that can be processed in parallel. Secondly, it must be possible to adapt the original workflow such that the induced memory accesses fit well to the special memory management of GPU computing (e.g., only little transfer between host and device as well as coalesced access to the device's main memory).

Algorithm 1 NAIVE BRUTE-FORCE SEARCH

Require: Training set $T = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\} \subset \mathbb{R}^d \times \mathbb{R}$ and $k \leq d$.
Ensure: Optimal solution for the best subset selection problem (2).

```
1: G_opt = ∞, S_opt = ∅
2: for each subset S ⊂ {1, ..., d} with |S| ≤ k do
3:     Compute G(S) for task (1) using X_S as input        O(nk^2 + k^3)
4:     if G(S) < G_opt then
5:         G_opt = G(S), S_opt = S
6:     end if
7: end for
8: return (G_opt, S_opt)
```

III. ALGORITHMIC FRAMEWORK

In the following, we will derive a massively-parallel implementation to reduce the practical runtime spent by an exhaustive search. To the best of our knowledge, this is the first time that the corresponding brute-force computations are accelerated via massively-parallel computing. As shown in our experimental evaluation, using such massively-parallel devices render subset selection problems with small k and large d and n possible in seconds or minutes instead of hours or even days.

A. Naive Brute-Force

The task of computing an optimal solution to the problem at hand is known to be NP-hard; nevertheless, for small assignments of k (say, up to $k = 6$), one can actually

[1] The description does not contain an intercept term. Often, such a term is modeled by adding a column of ones to the data matrix $\mathbf{X}$ and by subsequently addressing the $d+1$-dimensional task.

[2] https://cran.r-project.org/web/packages/leaps/leaps.pdf

Algorithm 2 FASTER BRUTE-FORCE SEARCH

Require: Training set $T = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\} \subset \mathbb{R}^d \times \mathbb{R}$ and $k \leq d$.
Ensure: Optimal solution for task (1)
1: Precompute matrix $\mathbf{M} \in \mathbb{R}^{d \times d}$ and vector $\mathbf{F} \in \mathbb{R}^d$ $\quad \mathcal{O}(nd^2)$
2: $\hat{G}_{\text{opt}} = \infty$, $S_{\text{opt}} = \emptyset$
3: **for** each subset $S \subset \{1, \ldots, d\}$ with $|S| \leq k$ **do**
4: Compute $\boldsymbol{\beta}_S = (\mathbf{M}_S)^{-1}\mathbf{F}_S$ $\quad \mathcal{O}(k^3)$
5: Compute $\hat{G}(S) = -2\boldsymbol{\beta}^{\mathrm{T}}\mathbf{F}_S + \boldsymbol{\beta}^{\mathrm{T}}\mathbf{M}_S\boldsymbol{\beta}$ $\quad \mathcal{O}(k^2)$
6: **if** $\hat{G}(S) < \hat{G}_{\text{opt}}$ **then**
7: $\hat{G}_{\text{opt}} = \hat{G}(S)$, $S_{\text{opt}} = S$
8: **end if**
9: **end for**
10: $G_{\text{opt}} = \mathbf{y}^{\mathrm{T}}\mathbf{y} + \hat{G}_{\text{opt}}$ $\quad \mathcal{O}(n)$
11: **return** $(G_{\text{opt}}, S_{\text{opt}})$

conduct an exhaustive search. A naive search procedure is sketched in Algorithm 1: For each possible subset $S = \{i_1, \ldots i_k\} \subset \{1, \ldots, d\}$, one considers the induced data matrix $\mathbf{X}_S \in \mathbb{R}^{n \times k}$ containing the corresponding subset of features as columns. For each such subproblem, one can then compute the associated optimal model parameters $\boldsymbol{\beta}_S \in \mathbb{R}^k$ via

$$\boldsymbol{\beta}_S = \left(\mathbf{X}_S^{\mathrm{T}}\mathbf{X}_S\right)^{-1}\mathbf{X}_S^{\mathrm{T}}\mathbf{y} \tag{3}$$

in case the matrix $\mathbf{X}_S^{\mathrm{T}}\mathbf{X}_S$ is invertible (for simplicity, the non-invertible cases are ignored, which is safe as we assume $k \ll n$). In practice, $\boldsymbol{\beta}_S$ would typically be computed relying on techniques such as QR-decomposition.

To assess the quality of a particular feature subset, we compute the objective value

$$G(S) = \|\mathbf{y} - \mathbf{X}_S\boldsymbol{\beta}_S\|_2^2. \tag{4}$$

Thus, a direct approach to compute the induced objective value $G(S)$ for each possible feature subset S is to first compute $\boldsymbol{\beta}_S$ and to subsequently compute the induced objective. Assuming that matrix inversion takes $\mathcal{O}(k^3)$ operations for a $k \times k$ matrix, dealing with a single subset takes $\mathcal{O}(nk^2 + k^3)$ time, which yields an overall runtime of $\mathcal{O}\left(d^k(nk^2 + k^3)\right)$.

B. Faster Brute-Force

One can reduce the runtime needed for the brute-force approach by precomputing the full data matrix $\mathbf{M} = \mathbf{X}^{\mathrm{T}}\mathbf{X} \in \mathbb{R}^{d \times d}$ and the vector $\mathbf{F} = \mathbf{X}^{\mathrm{T}}\mathbf{y} \in \mathbb{R}^d$: It is well known that one can rewrite the objective $G(S)$ as

$$\begin{aligned} G(S) &= (\mathbf{y} - \mathbf{X}_S\boldsymbol{\beta})^{\mathrm{T}}(\mathbf{y} - \mathbf{X}_S\boldsymbol{\beta}) \\ &= \mathbf{y}^{\mathrm{T}}\mathbf{y} - 2\boldsymbol{\beta}^{\mathrm{T}}\mathbf{F}_S + \boldsymbol{\beta}^{\mathrm{T}}\mathbf{M}_S\boldsymbol{\beta} \\ &= \mathbf{y}^{\mathrm{T}}\mathbf{y} + \hat{G}(S). \end{aligned}$$

Since the term $\mathbf{y}^{\mathrm{T}}\mathbf{y}$ is constant for all subsets, one can focus on the remaining terms $\hat{G}(S)$ (see, e.g., Moghaddam *et al.* [7]). Thus, given a fixed subset S, the corresponding optimal model parameters $\boldsymbol{\beta}_S \in \mathbb{R}^k$ can be obtained via $\boldsymbol{\beta}_S = (\mathbf{M}_S)^{-1}\mathbf{F}_S$, where the corresponding sub-columns/sub-rows are considered.

The overall adapted search is sketched in Algorithm 2: After the computation of the initial data matrix $\mathbf{M}$ and vector $\mathbf{F}$, all possible subsets of features are tested. For each subset S, one needs to invert the $k \times k$ matrix $\mathbf{M}_S$ spending $\mathcal{O}(k^3)$ time. Given the model coefficients $\boldsymbol{\beta}_S$, one can compute the objective $\hat{G}(S)$ spending $\mathcal{O}(k^2)$ time per subset. Note that the runtime per subset is independent of n. The computation of the full data matrix $\mathbf{M}$ and the vector $\mathbf{F}$ in the preprocessing phase takes $\mathcal{O}(nd^2)$ time. Hence, one can obtain an optimal solution for task (1) in $\mathcal{O}(nd^2 + d^k k^3)$ time using $\mathcal{O}(d^2)$ space.

Given small d and $k \geq 2$, the first term will usually dominate the runtime. For such cases, the search scheme shown in Algorithm 2 is about $\frac{d^k nk^2}{nd^2} = d^{k-2}k^2$ faster than the original brute-force search described in Algorithm 1. For high-dimensional data ($d \geq n$) and $k \geq 3$, the second part will dominate—leading to a speed-up of $\frac{d^k nk^2}{d^k k^3} = \frac{n}{k}$ over the naive brute-force implementation.

For the adapted search outlined above, the data matrix $\mathbf{M}$ has to be precomputed and stored in main memory. Modern commodity hardware based systems can easily accommodate data matrices with, say, $d = 50,000$ rows and columns (which would take about 20GB of main memory). In case even higher-dimensional features spaces are given, one can still process all subsets in chunks by considering corresponding subspaces of the whole feature space and by computing the associated data matrices only once for each such subspace.[3]

C. Massively-Parallel Brute-Force

The slightly more sophisticated implementation of the brute-force approach offers significant runtime savings over a naive version and, in addition, is also suited for an efficient massively-parallel execution: Conceptually, the search can be conducted via k nested loops (symmetric candidate solutions can be ignored). The key idea is to parallelize the work over the two outer most loops and to let each GPU thread compute the induced optimal model as well as the associated objective value $G(S)$. More precisely, each thread with global thread id `gid` first computes the two indices `i1 = (counter + gid) / d` and `i2 = (counter + gid) - i1 * d` corresponding to the outer loop (here, `counter` is a counter variable used to process all $\mathcal{O}(d^2)$ pairs in chunks). Given these two indices, each thread then computes the model optimal coefficients $\boldsymbol{\beta}_S$ and $\hat{G}(S)$ for each subset S induced by the remaining $d - 2$ indices. This is done via additional loops over the remaining coefficients (e.g., via loops over `i3` and `i4` in case of $k = 4$).

The main benefit of the slightly more sophisticated brute-force implementation in this context is the fact that only $\mathcal{O}(k^2)$ data items need to be accessed per thread. More precisely, after the auxiliary matrix $\mathbf{M}$ and the vector $\mathbf{F}$ have been precomputed (using efficient massively-parallel matrix libraries), each thread only accesses $\mathbf{M}_S$ and $\mathbf{F}_S$. In particular, each thread takes care of the Steps 4 to 5 in Algorithm 2. The latter one amounts to simple matrix multiplications with matrices and vectors of size $k \times k$ and k, respectively. For the former step, one needs to invert the matrix $\mathbf{M}_S$. The key idea of the efficient many-core implementation is to compute the inverse analytically—which is beneficial for both the standard

[3] By doing so, the runtime needed for constructing the associated data matrices again from time to time will get amortized by the runtime needed for subsequently evaluating all subsets.

CPU implementation and an important ingredient for the GPU implementation. The efficiency of the overall implementation is based on the following algorithmic ingredients:

1) *Coalesced/Cached Memory Access:* Since the threads are instantiated via the two outermost loops, threads with a similar thread id process similar candidate subsets S and, hence, access similar locations in memory. This leads to most of the memory accesses being either coalesced or efficiently supported via caching (in case all of the threads access the same/nearby memory locations).
2) *Analytical Matrix Inverse:* Each thread needs to compute, for many possible subsets S, the associated model parameters $\boldsymbol{\beta}_S$ and the induced objective $\hat{G}(S)$. The computation of the model parameters require the inversion of small matrices. The key ingredient of our massively-parallel implementation is that we compute the involved matrix inverses analytically, i.e., the code executed by the threads essentially only contains a series of multiplications and additions.[4]
3) *Chunks:* The number of possible subsets can be extremely large. Hence, even though the threads are instantiated via the two outermost loops, one still needs to invoke the threads in chunks. This is achieved via the variable `counter` mentioned above.

To sum up, each GPU thread conducts a series of simple matrix operations based on subsets of $\mathbf{M}$ and $\mathbf{F}$. Since these computations are basically the same for all threads in a warp, only little branch divergence happens (except for the `if` statements that are needed to keep track of optimal solutions). Further, nearby threads generally access the same or nearby locations in global memory, which is effectively supported by today's GPUs. As shown in our experimental evaluation, the induced implementation yields promising speed-ups over a corresponding multi-core implementation.

IV. Experiments

The purpose of our experimental evaluation is to analyze the benefits of the massively-parallel brute-force implementation over its multi-core competitor. We do not aim at a detailed comparison with other techniques such as branch-and-bound implementations since these are conceptually very different from ours. As shown below, the massively-parallel implementation can be used to solve optimization at hand for problem instances with $n = 20,000$ and $d = 5,000$ and $k = 3$ as well as $d = 1,000$ and $k = 4$ in less than 100 seconds using standard desktop computers.

A. Experimental Setup

We resort to standard commodity hardware for all experiments. More precisely, we use a desktop computer with an `Intel(R) Core(TM) i7-3770` CPU running at 3.40GHz (4 cores), 16GB RAM, and a `Nvidia Titan Z` GPU having 2880 shader units and 6 GB RAM (only one of its two devices is used). The operating system is `Ubuntu 16.04` (64 Bit) with kernel `4.4.0-83` and `CUDA 8.0` (graphics driver version 375.51).

We consider three brute-force implementations: (1) The naive approach that recomputes the intermediate models from scratch for each subset, (2) the more sophisticated version that resorts to the global matrix $\mathbf{M}$ and the vector $\mathbf{F}$, and (3) the corresponding massively-parallel variant described above. For all implementations, we resort to `Python 2.7` as main programming language. All compute-intensive parts are implemented efficiently via `C`, `CUDA`, and external libraries. In particular, we resort to *Scikit-Learn* (version 0.18.2) [14] for computing least-squares models from scratch for (1), to `C` and *Swig* [15] for (2), and to `PyCUDA` [16] and `scikit-cuda` [17] for (3).

For the runtime comparisons, we make use of artificial dataset instances generated via the `make_regression` function provided by the *Scikit-Learn* package (`n_samples`=n, `n_features`=d, `n_informative`=k, `noise`=10, `coef`=True, `bias`=100). The function generates regression scenarios with a pre-defined number of informative features. Note that the runtimes provided for the brute-force implementations are relatively independent of the particular dataset given, as long as the parameters considered are similar. That is, the computations are not significantly affected by the particular properties of a given dataset such as the distribution of points in the feature space. We also consider a real-world dataset, whose details are provided below.

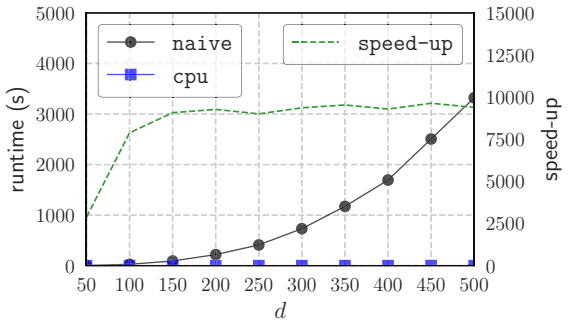
For the sake of illustration, we focus on subsets S with a fixed cardinality k, i.e., we make use of the constraint $\|\boldsymbol{\beta}\|_0 = k$ instead of $\|\boldsymbol{\beta}\|_0 \leq k$ in Equation (2). The latter case can easily be handled by considering the former one k times.

B. Runtime Analysis

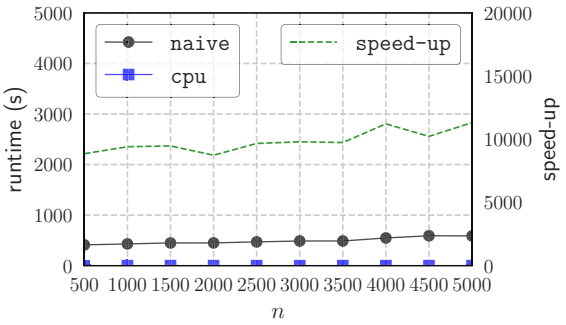
Figure 1 shows a comparison between the naive implementation of the brute-force approach and the more sophisticated one that resorts to the global matrix $\mathbf{M}$ and the vector $\mathbf{F}$. In particular, Figure 1 (a) shows the runtimes of both schemes for a varying number d of input dimensions given a fixed number $n = 1,000$ of training instances and a fixed number $k = 3$ of features to be selected.

The speed-up obtained via the slightly more sophisticated brute-force implementation is sketched via the green dotted line. As it can be seen, the adapted brute-force scheme is about four orders of magnitude faster. Further, as expected, the speed-up stabilizes once the feature space dimensionality d gets sufficiently large. This is due to the second term of runtime bound $\mathcal{O}(nd^2 + d^k k^3)$ for the more sophisticated implementation dominates the overall runtime at some point, which yields a theoretical speed-up of $\frac{n}{k}$ over the naive implementation. This speed-up is independent of d, which is in line with the results showin in Figure 1. A corresponding runtime comparison for a varying number n of training instances given a constant input dimensionality $d = 250$ and constant number $k = 3$ of features to be selected is shown in Figure 1 (b).

[4]Note that invoking libraries for this task is not directly possible since one needs to compute the inverses for many individual matrices.



(a) $k = 3$



(b) $k = 3$

Fig. 1. (a) Comparison of the naive brute-force implementation and the more sophisticated one that makes use of the global matrix $\mathbf{M}$ and the vector $\mathbf{F}$. The naive implementation is about four orders of magnitude slower than the more sophisticated implementation. A corresponding comparison given a varying number n of training instances is shown in Figure (b).

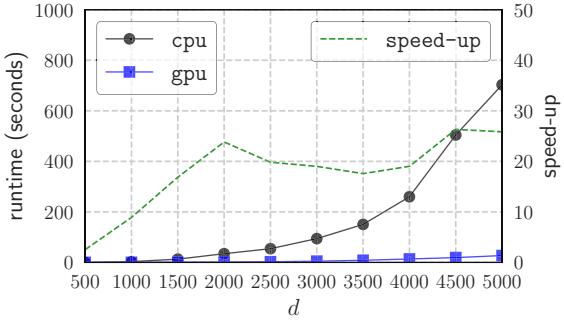
Next, we compare the practical runtime performances of the more advanced brute-force searches on the CPU and GPU. We consider both $k = 3$ and $k = 4$ as assignments for the sizes of the desired feature subsets. For each of the two cases, we consider a fixed number $n = 20,000$ of training instances and vary the number d of input dimensions (varying n does not affect the runtime for the scenarios considered).

The outcome is shown in Figure 2. It can be seen that the GPU implementation achieves valuable speed-ups in both cases. This is especially the case for large d and $k = 4$ due to the underlying task being the most computationally intensive one. Thus, the massively-parallel implementation is up to 30 times faster than an efficient single-core CPU implementation. Since the latter one is about four 10,000 times faster than the naive approach, the final speed-up of the many-core version is about 300,000. This renders problem instances with $k = 3$ or $k = 4$ solvable in seconds or minutes compared to hours or days using a standard brute-force implementation.

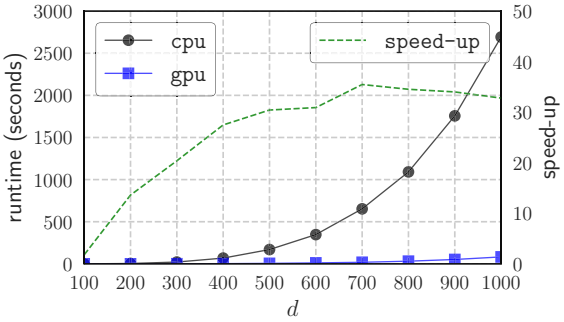
C. Application

The best subset selection problem as well as the optimization framework proposed in this work are, in general, applicable to a wide range of regression problems. An interesting application domain for such feature selection schemes is the field of astronomy due to the fact that one can often extract very expressive features for the objects at hand.

To sketch the applicability of our approach, we consider *photometric redshift estimation* for quasars, which depicts an important problem in astronomy: Two very common types of astronomical data are *photometric* and *spectroscopic data* [18]. Photometric data correspond to images taken at different wavelength regions, whereas spectroscopic data correspond to time-consuming follow-up observations that are made for a relatively small subset of "interesting" objects. Given such spectra, one can obtain precise measurements for, e.g., the redshift of the objects (the redshift depicts a proxy for the distance of the objects to Earth). Photometric redshift estimation aims at obtaining estimates for these redshift measurements given the photometric data only, see Figure 3 for an illustration.



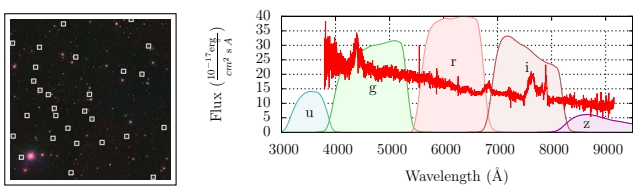
(a) $k = 3$



(b) $k = 4$

Fig. 2. Influence of the dimensionality d on the runtime for both $k = 3$ and $k = 4$. The speed-up of the GPU implementation over its CPU competitor (single core execution) is indicated via the green dashed line. Given the computationally intensive dataset instances, a speed-up of up to 30 is achieved.

This task is among the most challenging and important ones in astronomy. Especially the search for new, distant quasars depicts an extremely difficult problem. From a machine learning point of view, photometric redshift estimation can be modeled as a regression problem [2] and several approaches have been proposed in the past years. Among these approaches are schemes that are based on neural networks [19], Gaussian processes [20], support vector machines [21], nearest neighbor



(a) Photometric Data (b) Specroscopic Data

Fig. 3. For the SDSS [18], photometric data are given as grayscale images that are based on five filters covering different wavelength ranges (called the `u`, `g`, `r`, `i`, and `z` bands). In Figure (a) an RGB image is shown that is based on such images. For a small subset of detected objects (white squares), detailed follow-up observations in terms of spectra are available, see Figure (b) [22]. The task of photometric redshift estimation is to estimate, based on the photometric data, the redshift of a detected object (the true redshift is usually obtained by analzing the associated spectrum in detail).

models [22], template fitting methods [23], and other schemes and analyses [24], [25]. Feature selection schemes have been successfully applied as well in this context [26], [27], [28] and, in particular, using linear models for photometric redshift estimation [29]. We are, however, not aware of any approaches conducting *exact* feature selection for linear regression in this context.

1) Datasets and Models: Given the photometric data for an object, one usually extracts informative features. This typically happens for each wavelength region. We consider data from the *Sloan Digital Sky Survey* (DR12) [18]. In this case, five such wavelength regions are given, called the `u`, `g`, `r`, `i`, and `z` *band*. In this context, the most established feature extraction methods are the *point-spread-function* (psf), the *Model*, and the *Petrosian* magnitudes [18]. In combination with other methods, such methods give rise to a feature vector for each of the detected objects.

We consider two dataset instances. The first dataset is based on all spectroscopically confirmed quasars in the SDSS (DR12) with redshift $z \in [0.0, 5.0]$, which yields $345,622$ instances in total. The second dataset is based on all quasars with a redshift $z \in [4.0, 5.0]$, which contains 2050 instances in total. Again, the "true" redshift values stem from the time-consuming spectroscopic follow-up observations, and the goal is to estimate such values based on the imprecise photometric data. For both datasets, we consider two different feature spaces. The first feature space is based on the so-called *colors* $u-g$, $g-r$, $r-i$, and $i-z$, which are composed of difference values w.r.t. the psf magnitudes (hence, a four-dimensional space). This feature space is usually considered for the task of photometric redshift estimation for quasars (see, e.g., Polsterer *et al.* [22]). The second one is based on various raw features that are available in the SDSS database for each of the objects. Further, various (non-linear) transformations are applied to these raw features such as difference/ratio values based on pairs of raw features, squared and cubed features, etc. This gives rise to $d = 4,520$ features in total.

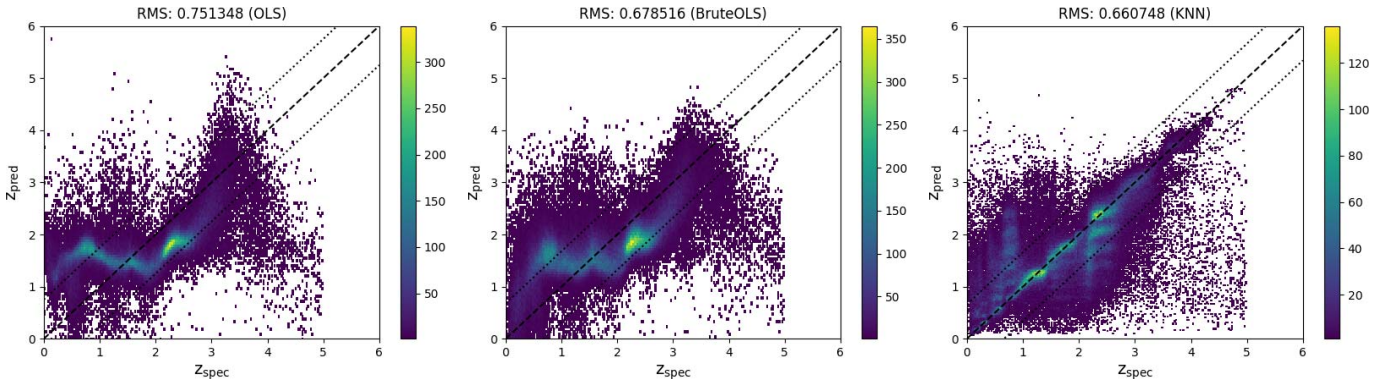
We consider three regressions models: (1) ordinary least squares (`ols`), (2) best subset selected for ordinary least-squares (`bs-ols`), and (3) nearest neighbor regression (`knn`). For both `ols` and `knn`, we resort to the four-dimensional features space (psf colors), which depicts the standard feature space usually considered for this task. For the `bs-ols` scheme, we consider an extended feature space that stems from using various raw features that are given for each object in the database. For `bs-ols`, we make use of our implementation to select $k = 3$ optimal features according to optimization task (2). For all three models, we split up each dataset into two almost equal-sized subsets, where the first half is used for training and the second half for evaluating the final performance via the *root mean square* (RMS) error. For the extended feature space, all features are normalized to $[0, 1]$.

2) Results: A comparison of the performances for the first dataset is shown in Figure 4. It can be seen that the `bs-ols` model outperforms its direct `ols` competitor using only three features that are selected out of the $4,520$ available features (the $k = 3$ optimal features are `norm(psfMag_r-psfMag_i)`, `norm(dered(psfMag_i)/psfMag_u)`, and `norm(psfMag_i/psfMag_r)`, where `norm` is the normalization operator that scales the feature values to the interval $[0, 1]$). Thus, using these three features instead of the standard four color features used for `ols` yields a better overall performance w.r.t. the RMS. The performance of `bs-ols` is even competitive with the more flexible `knn` model. Hence, for the general task of photometric redshift estimation, least-squares regression models in combination with (exact) feature selection depict valuable alternatives. An additional benefit of these models is the fact that they are reproducible without any particular reference set, as it is the case for, e.g., nearest neighbor models.

A similar outcome can be observed for the second dataset, where only quasars with a redshift $z \in [4.0, 5.0]$ are considered. Here, the performance of `bs-ols` is even slightly better than the one of `knn` (which might, in principle, further be improved via a different feature space as well). Especially for such settings, the `bs-ols` model offers a valuable choice of model, which might also be helpful in case one is interested in extrapolation, i.e., in predicting redshift values for $z > 5.0$ in this case. This is not possible with, e.g., nearest neighbor models.

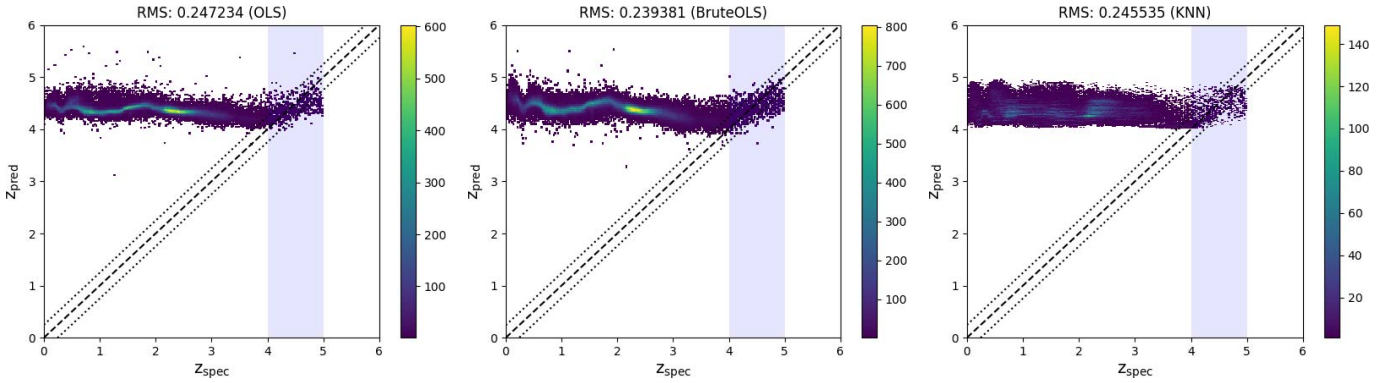
V. Conclusions

We address the best subset selection problem for linear regression scenarios whose underlying optimization task is NP-hard and, hence, difficult to solve. Our approach, which is based on using powerful massively-parallel devices, can efficiently handle scenarios with a high-dimensional feature space and/or a large amount of training instances in case one is interested in feature subsets of small cardinality k. The implementation proposed in this work resorts to a simple brute-force approach that tests all possible subsets. Our key contribution is an efficient massively-parallel implementation for such an approach, which significantly reduces the practical runtime



(a) `ols` (b) `bs-ols` (c) `knn`

Fig. 4. Photometric redshift estimation for quasars. Three models are fitted for quasars with a redshift $z \in [0.0, 5.0]$. As feature space, we resort to the colors $u-g$, $g-r$, $r-i$, and $i-z$ (psf magnitudes) for both the `ols` and the `knn` model, whereas we consider an extended feature space for `bs-ols` and select $k = 3$ optimal features. It can be seen that the `bs-ols` model performs significantly better than the direct `ols` competitor and only slightly worse than the baseline `knn` model. For the `bs-ols` method, the following features were selected: `norm(petroMag_z)`, `norm(dered(expMag_g)/psfMag_u)`, and `norm(dered(devMag_i)/psfMag_i)`. Thus, the `bs-ols` implementation can be used to successfully select a small number of features in the extended feature space such that the resulting model is competitive with state-of-the-art approaches for the task at hand.



(a) `ols` (b) `bs-ols` (c) `knn`

Fig. 5. A similar experiment as shown in Figure 4. This time, only quasars with a redshift of $z \in [4.0, 5.0]$ are used both for training and testing. It can be seen that `bs-ols` slightly outperforms its two competitors, while resorting to three instead of four features only. The `bs-ols` scheme selected different features as before (`norm(psfMag_r-psfMag_i)`, `norm(dered(psfMag_i)/psfMag_u)`, and `norm(psfMag_i/psfMag_r)`).

by several orders of magnitude compared to a naive implementation. We sketch the speed-ups achieved using various artificial dataset instances and demonstrate the applicability of the overall approach in the context of astronomy.

We plan to improve and extend the results presented in this work. An interesting research direction is the combination of the implementation provided in this work with more sophisticated search strategies such as branch-and-bound approaches. We expect the combination of such tools and concepts to further reduce the practical runtime and to render solving the problem for larger feature subsets possible. Further, we expect our framework to be extendable to other learnings tasks as well. In particular, the framework can naturally be extended to linear classification models such as linear discriminant analysis, which could be used to effectively select simple yet expressive classification models.

Achknowledgements

Fabian Gieseke acknowledges support from the Danish Industry Foundation through the *Industrial Data Analysis Service* (IDAS) and Christian Igel acknowledges support from the Innovation Fund Denmark through the *Danish Center for Big Data Analytics Driven Innovation* (DABAI).

REFERENCES

[1] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *Journal of Machine Learning Research*, vol. 3, pp. 1157–1182, 2003.

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

[3] A. Miller, *Subset Selection in Regression*, ser. Monographs on Statistics and Applied Probability. Chapman & Hall/CRC, 2002.

[4] B. K. Natarajan, "Sparse approximate solutions to linear systems," *SIAM Journal on Computing*, vol. 24, no. 2, pp. 227–234, 1995.

[5] D. Bertsimas, R. Mazumder, and A. King, "Best subset selection via a modern optimization lens," *The Annals of Statistics*, vol. 44, no. 2, pp. 813–852, 2016.

[6] M. Hofmann, C. Gatu, and E. J. Kontoghiorghes, "Efficient algorithms for computing the best subset regression models for large-scale problems," *Computational Statistics & Data Analysis*, vol. 52, no. 1, pp. 16–29, 2007.

[7] B. Moghaddam, A. Gruber, Y. Weiss, and S. Avidan, "Sparse regression as a sparse eigenvalue problem," in *Information Theory and Applications Workshop*, 2008, pp. 121–127.

[8] B. Catanzaro, N. Sundaram, and K. Keutzer, "Fast support vector machine training and classification on graphics processors," in *Proceedings of the 25th International Conference on Machine Learning*. New York, NY, USA: ACM, 2008, pp. 104–111.

[9] A. Coates, B. Huval, T. Wang, D. J. Wu, B. C. Catanzaro, and A. Y. Ng, "Deep learning with cots hpc systems," in *Proceedings of the 30th International Conference on Machine Learning*. JMLR.org, 2013, pp. 1337–1345.

[10] F. Gieseke, J. Heinermann, C. Oancea, and C. Igel, "Buffer k-d trees: Processing massive nearest neighbor queries on GPUs," in *Proc. of the 31st International Conf. on Machine Learning*, ser. JMLR W&CP, vol. 32, no. 1. JMLR.org, 2014, pp. 172–180.

[11] F. Gieseke, C. Oancea, and C. Igel, "bufferkdtree: A python library for massive nearest neighbor queries on multi-many-core devices," *Knowledge-Based Systems*, vol. 120, pp. 1–3, 2017.

[12] Z. Wen, R. Zhang, K. Ramamohanarao, J. Qi, and K. Taylor, "Mascot: Fast and highly scalable SVM cross-validation using GPUs and SSDs," in *Proceedings of the 2014 IEEE International Conference on Data Mining*, 2014, pp. 580–589.

[13] J. Cheng, M. Grossman, and T. McKercher, *Professional CUDA C Programming*. John Wiley & Sons, 2014.

[14] 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*, vol. 12, pp. 2825–2830, 2011.

[15] D. M. Beazley, "Swig: An easy to use tool for integrating scripting languages with c and c++," in *Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4*, ser. TCLTK'96. Berkeley, CA, USA: USENIX Association, 1996, pp. 15–15. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267498.1267513

[16] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation," *Parallel Computing*, vol. 38, no. 3, pp. 157–174, 2012.

[17] L. E. Givon, T. Unterthiner, N. B. Erichson, D. W. Chiang, E. Larson, L. Pfister, S. Dieleman, G. R. Lee, S. van der Walt, B. Menn, T. M. Moldovan, F. Bastien, X. Shi, J. Schlüter, B. Thomas, C. Capdevila, A. Rubinsteyn, M. M. Forbes, J. Frelinger, T. Klein, B. Merry, L. Pastewka, S. Taylor, A. Bergeron, F. Wang, and Y. Zhou, "scikit-cuda 0.5.1: a Python interface to GPU-powered libraries," 2015. [Online]. Available: http://dx.doi.org/10.5281/zenodo.40565

[18] S. Alam, F. D. Albareti, C. Allende Prieto, F. Anders, S. F. Anderson, T. Anderton, B. H. Andrews, E. Armengaud, É. Aubourg, S. Bailey, and et al., "The Eleventh and Twelfth Data Releases of the Sloan Digital Sky Survey: Final Data from SDSS-III," *The Astrophysical Journal Supplement Series*, vol. 219, p. 12, Jul. 2015.

[19] O. Laurino, R. D'Abrusco, G. Longo, and G. Riccio, "Astroinformatics of galaxies and quasars: a new general method for photometric redshifts estimation," *Monthly Notices of the Royal Astronomical Society*, vol. 418, no. 4, pp. 2165–2195, 2011.

[20] D. G. Bonfield, Y. Sun, N. Davey, M. J. Jarvis, F. B. Abdalla, M. Banerji, and R. G. Adams, "Photometric redshift estimation using gaussian processes," *Monthly Notices of the Royal Astronomical Society*, vol. 405, no. 2, pp. 987–994, 2010.

[21] Y. Wadadekar, "Estimating photometric redshifts using support vector machines," *The Publications of the Astronomical Society of the Pacific*, vol. 117, no. 827, 2005.

[22] K. L. Polsterer, P. Zinn, and F. Gieseke, "Finding new high-redshift quasars by asking the neighbours," *Monthly Notices of the Royal Astronomical Society (MNRAS)*, vol. 428, no. 1, pp. 226–235, 2013.

[23] M. Bolzonella, J.-M. Miralles, and R. Pelló, "Photometric redshifts based on standard SED fitting procedures," *Astronomy and Astrophysics*, vol. 363, pp. 476–492, 2000.

[24] R. Beck, C. A. Lin, E. E. O. Ishida, F. Gieseke, R. S. de Souza, M. Costa-Duarte, M. W. Hattab, and A. Krone-Martins, "On the realistic validation of photometric redshifts," *Monthly Notices of the Royal Astronomical Society*, vol. 468, no. 4, p. 4323, 2017.

[25] N. Benítez, "Bayesian Photometric Redshift Estimation," *Astrophysical Journal*, vol. 536, pp. 571–583, Jun. 2000.

[26] C. Donalek, A. Arun Kumar, S. G. Djorgovski, A. A. Mahabal, M. J. Graham, T. J. Fuchs, M. J. Turmon, N. Sajeeth Philip, M. T.-C. Yang, and G. Longo, "Feature Selection Strategies for Classifying High Dimensional Astronomical Data Sets," *ArXiv e-prints*, Oct. 2013.

[27] K. 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 (ADASS)*, 2013.

[28] S. Heinis, S. Kumar, S. Gezari, W. S. Burgett, K. C. Chambers, P. W. Draper, H. Flewelling, N. Kaiser, E. A. Magnier, N. Metcalfe, and C. Waters, "Of Genes and Machines: Application of a Combination of Machine Learning Tools to Astronomy Data Sets," *The Astrophysical Journal*, vol. 821, p. 86, Apr. 2016.

[29] A. Krone-Martins, E. E. O. Ishida, and R. S. de Souza, "The first analytical expression to estimate photometric redshifts suggested by a machine," *Monthly Notices of the Royal Astronomical Society*, vol. 443, pp. L34–L38, Sep. 2014.