High-Performance Geometric Algorithms for Sparse Computation in Big Data Analytics

Philipp Baumann
Department of Business Administration
University of Bern
Switzerland
philipp.baumann@pqm.unibe.ch

Dorit S. Hochbaum
IEOR Department
University of California, Berkeley
USA
dhochbaum@berkeley.edu

Quico Spaen
IEOR Department
University of California, Berkeley
USA
qspaen@berkeley.edu

***Abstract*—Several leading supervised and unsupervised machine learning algorithms require as input similarities between objects in a data set. Since the number of pairwise similarities grows quadratically with the size of the data set, it is computationally prohibitive to compute all pairwise similarities for large-scale data sets. The recently introduced methodology of "sparse computation" resolves this issue by computing only the relevant similarities instead of all pairwise similarities. To identify the relevant similarities, sparse computation efficiently projects the data onto a low-dimensional space where a similarity is considered relevant if the corresponding objects are close in this space. The relevant similarities are then computed in the original space. Sparse computation identifies close pairs by partitioning the low-dimensional space into grid blocks, and considering objects close if they fall in the same or adjacent grid blocks. This guarantees that all pairs of objects that are within a specified L_∞ distance are identified as well as some pairs that are within twice this distance. For very large data sets, sparse computation can have high runtime due to the enumeration of pairs of adjacent blocks. We propose here new geometric algorithms that eliminate the need to enumerate adjacent blocks. Our empirical results on data sets with up to 10 million objects show that the new algorithms achieve a significant reduction in runtime. The algorithms have applications in large-scale computational geometry and (approximate) nearest neighbor search. Python implementations of the proposed algorithms are publicly available.**

***Keywords*-Big data; similarity-based machine learning; sparsification; sparse computation; computational geometry**

I. Introduction

Machine learning algorithms that use pairwise similarities such as the k-nearest neighbor algorithm, the supervised normalized cut algorithm [1], and kernel-based support vector machines [2] are often superior in performance compared to non-similarity-based algorithms (see e.g. [3]). Pairwise similarities are able to incorporate non-linear and transitive relationships in the data set. This transitivity property makes pairwise similarity particularly robust, because similarities are captured between pairs of objects that are not directly compared via a transitive chain of similarities [4].

While the use of pairwise similarities is beneficial, it is computationally expensive to compute all pairwise similarities. This is because the number of pairwise similarities grows quadratically with the number of objects in the data set. For instance, the complete similarity matrix for a data set with 500,000 objects has 125 billion entries. If it takes a microsecond to compute one entry, then it would take more than a day to compute all entries. In addition, it would require considerable memory to store all entries.

Nevertheless, to achieve good classification or clustering accuracy, it is often unnecessary to compute all entries of the similarity matrix. Recently, a method called sparse computation [5] was introduced to efficiently identify highly-similar pairs in large-scale data sets. It was shown that the accuracy obtained with a sparse similarity matrix is almost identical to the accuracy obtained with the complete similarity matrix. See Figure 1 for an illustration.

Most approaches for sparsifying a matrix remove some entries while preserving specific matrix properties, see e.g., [6], [7], [8]. These approaches are not suitable for large-scale data sets because they require as input the complete similarity matrix. To enable similarity-based data mining in large-scale data sets, it is necessary to identify similar pairs without computing a large number of dissimilar pairs.

In sparse computation the data is first projected onto a low-dimensional space. This projection is typically done with a probabilistic version of principal component analysis. Sparse computation relies on closeness in the low-dimensional space as a proxy for similarity in the original space. The low-dimensional space is then subdivided into grid blocks and only pairs of objects that fall into the same block or in neighboring blocks are deemed to be similar. The accepted degree of similarity can be controlled by varying the dimension of the low-dimensional space or by varying the grid resolution. Sparse computation has been used to drastically reduce the runtime of classification [5] and clustering algorithms [9] with minor and often no loss in accuracy (see Figure 1). It has also been used to efficiently compress nearly-identical objects into a single object to generate a compact representation of the data set with minor loss of information [10].

For large-scale data sets, the computational bottleneck of sparse computation is the identification of pairs of adjacent

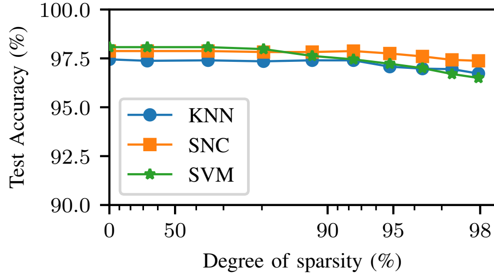

Figure 1. Accuracy at selected sparsity levels for the letter recognition (LER) dataset obtained with three similarity-based classifiers (k-nearest neighbors, SNC [1], and SVM [2]) on a sparse similarity matrix generated with sparse computation. Sparsity, reported on a log scale, is measured as the percentage of matrix entries that are not computed. The runtime of the algorithms is inversely proportional to the sparsity of the similarity matrix.

blocks in the grid structure. For each non-empty block, it identifies adjacent blocks and checks whether these blocks are non-empty. This process is referred to as *block enumeration*. For large-scale data sets to which sparse computation is applied with a high grid resolution, the vast majority of adjacent blocks are empty, but are still checked. Hence, a large fraction of the computational workload is unnecessary.

In this paper, we introduce a computational geometry concept called data shifting, which is used to identify pairs of similar objects in a low-dimensional space much faster than with state-of-the-art techniques. More precisely, data shifting enables us to identify, for a set of objects in $\mathbb{R}^d$, the set of close neighbors for each of the objects simultaneously. The set of "close" neighbors of an object consists of all objects that are within a specified value of L_∞ distance and possibly some objects that are within twice this distance. This can be interpreted as a 2-approximation for the problem of finding all neighbors within a given L_∞ distance.

We devise two algorithms for identifying close neighbors that utilize data shifting to relieve the computational bottleneck of identifying objects in non-empty adjacent blocks. Data shifting may also have potential applications in other contexts, such as in approximate nearest neighbor search.

The first algorithm, called *object shifting*, shifts the objects multiple times along different directions within the grid structure. For each shift, the pairs of objects that fall in the same block are deemed to be similar. Object shifting avoids having to explicitly compute adjacent blocks, but the same pairs of objects may be selected for different shifts. The second algorithm, called *block shifting*, partially addresses this drawback of duplicate pairs by identifying all pairs of non-empty adjacent blocks by shifting representatives for non-empty blocks instead of the individual objects. Note that these two algorithms generate the same pairs as sparse computation with block enumeration.

We compare the algorithms with block enumeration on real-world data sets with up to 10 million objects. The results indicate that sparse computation with block shifting is the overall fastest approach. Sparse computation with block shifting is consistently faster than the implementation with block enumeration irrespective of the data set or grid resolution. The speedup factor increases with increasing grid resolution and can be up to a factor of five. For very high grid resolutions, object shifting is sometimes slightly faster than block shifting. The Python source code for all three algorithms is publicly available at https://github.com/hochbaumGroup/sparsecomputation.

Other strategies that determine for each object its most similar neighbors, without computing the full similarity matrix first, include (approximate) nearest neighbor search techniques such as locality sensitive hashing (LSH) [11], [12] and kd-trees [13]. LSH uses hash functions to map objects to buckets. Any pair of objects that map to the same bucket for one of the hash functions is evaluated. Limited empirical experience [5] with LSH indicates that a large number of hash functions are needed. This results in evaluating many pairs and consequently high runtime.

The paper is structured as follows. In Section II, we discuss existing methods for efficiently finding similar objects in a data set. In Section III, we present the concept of data shifting and two geometric algorithms for sparse computation. In Section IV, we compare the runtime of sparse computation with block enumeration, object shifting, and block shifting. In Section V, we conclude the paper and provide some directions for future research. Throughout the paper, we use the notation given in Table I.

Table I
NOTATION

Symbol	Description
n	Number of objects
d	Number of features
$\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \mathbb{R}^d$	Objects
p	Number of dimensions in low-dimensional space
k and k'	Grid resolution
ω	Pre-specified L_∞ distance

II. Existing methods for finding similar objects

The problem of finding highly-similar objects in a data set is often tackled by performing a series of k-nearest neighbor searches. The k-nearest neighbor search problem consists of finding the k objects that are most similar to a given object. In Sections II-A and II-B, we present two well-known methods for exact and approximate nearest neighbor search, respectively. In Section II-C, we discuss the method of sparse computation that does not rely on nearest neighbor search and instead directly identifies and returns pairs of similar objects in a data set.

A. Kd-trees

A kd-tree represents a data set of n objects with d features as a binary tree with n nodes, [13]. Each node corresponds to an object and is associated with one of the d dimensions of the feature space. A non-leaf node can be interpreted as a hyperplane that is perpendicular to the associated dimension and divides the d-dimensional space into two half-spaces. Objects one the left side of the hyperplane are represented by the left subtree and objects on the right side of the hyperplane are represented by the right subtree. The dimensions are associated to the nodes in a cyclical manner as one moves down the tree. To construct a balanced kd-tree, one usually chooses the non-leaf nodes to be the object that is the median of the objects in the subtree with respect to the associated dimension. The kd-tree can be used to efficiently find the k-nearest objects of a given object by eliminating large parts of the search space. However, in high-dimensional spaces, nearest neighbor search based on kd-trees is only slightly faster than a linear search of all objects because the number of nodes visited increases considerably with increasing dimensions. Another disadvantage is that to find all pairs of similar objects, n nearest neighbor queries are necessary, which results in redundant work when pairs of objects are mutual neighbors. Retrieving all nearest neighbors of a data set is referred to as the all-nearest-neighbors problem in the literature [14].

B. Locality sensitive hashing

Locality sensitive hashing (LSH) is a popular technique for approximate nearest neighbor search that is much faster than kd-trees for high-dimensional data sets, [11], [12]. However, it is a randomized algorithm that does not guarantee to find the actual nearest neighbors in every case. The idea of LSH is to use random projections to map high-dimensional objects to buckets. Since random projections are distance preserving, similar objects are more likely to be mapped to the same bucket than dissimilar objects. Given a query object, LSH uses the random projections to also map the query object to buckets. Each object from the data set that mapped to one of these buckets is considered as a candidate nearest neighbor. An advantage of LSH is that it provides a probability guarantee that it will return the actual nearest neighbors. However, to reach a sufficiently high probability level it is often necessary to use a large number of random projections which increases the number of candidates that need to be considered [5]. Moreover, LSH shares the disadvantage with kd-trees that n nearest neighbor queries are required to find all pairs of similar objects.

C. Sparse computation

Sparse computation [5] takes as input a data set with n objects $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \mathbb{R}^d$ and d features. The method consists of the steps: dimension reduction, grid construction and selection of pairs, and similarity computation.

In the dimension reduction step, the input data is projected from a d-dimensional space onto a p-dimensional space, where $p \ll d$. The projection is done with a probabilistic variant of PCA called approximate PCA [5]. Let the data in the p-dimensional space be normalized, i.e., the values of each dimension are scaled to the range $[0, 1]$.

In the grid construction and selection of pairs step, the goal is to select all pairs of objects that have an L_∞ distance smaller or equal to ω in the p-dimensional space. This is achieved as follows. First, the range of values along each dimension is subdivided into $k = \frac{1}{\omega}$ equally long intervals. This partitions the p-dimensional space into k^p grid blocks. Parameter k denotes the grid resolution. Each object is then assigned to a single block based on its p coordinates. Objects which lie exactly on a grid line (horizontally and/or vertically) are assigned to the upper and/or right grid block. If the upper and/or right grid block is outside the grid, then the object is assigned to the lower and/or left block.

Since the largest L_∞ distance within a block is equal to $\omega = \frac{1}{k}$, all pairs of objects that belong to the same block are selected. In addition, some pairs of objects are within a distance of ω but fall in different blocks. To select those pairs as well, horizontally, vertically, and diagonally adjacent blocks need to be considered. Each block has up to $3^p - 1$ neighbors. We refer to this process as *block enumeration*. By selecting all pairs of objects that are assigned to adjacent blocks, it can be guaranteed that all pairs of objects whose distance is less than or equal to ω are selected. It is possible that pairs of objects whose L_∞ distance is more than ω but less than 2ω are selected as well, whereas objects whose distance is more than 2ω will not be selected.

The total number of selected pairs depends on the grid resolution k and the dimension of the low-dimensional space p. A higher grid resolution results in smaller blocks and thus reduces the set of pairs that fall in a block or its adjacent blocks. Similarly, when the number of dimensions of the low-dimensional space is increasing, the blocks contain fewer objects and this reduces the number of pairs selected.

In the similarity computation step, a similarity function is used to quantify the similarity for each of the pairs selected in the previous step. The similarity value is computed with respect to the original d-dimensional space.

The selection of pairs in sparse computation relies on enumerating $(3^p-1)/2$ adjacent blocks for each non-empty block to determine pairs of adjacent non-empty blocks[1]. This computation becomes the bottleneck of the method when a) the number of non-empty blocks is large, and b) most of the adjacent blocks are empty. Checking empty blocks is unnecessary and only contributes to the runtime. Conditions a) and b) are often met when sparse computation is applied with high grid resolution to a large-scale data set with

[1]For each non-empty block, only half of the adjacent blocks need to be checked since the adjacency relation is symmetric.

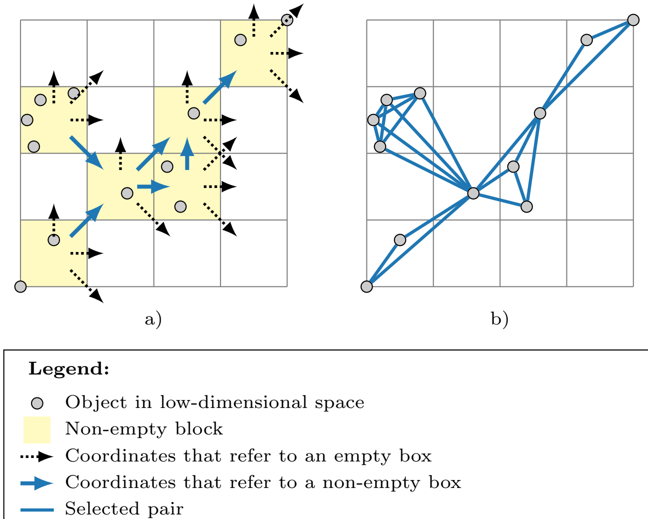

Figure 2. Visualization of the block enumeration strategy with $k = 4$ and $p = 2$: a) for each non-empty block, four adjacent blocks must be considered to identify all pairs of adjacent non-empty blocks. The majority of considered blocks are empty (dotted arrows). b) all pairs of objects that fall into the same or in neighboring blocks are selected (blue lines)

millions of objects. To illustrate this issue, Figure 2 shows a two-dimensional projection of a data set that contains 12 objects. With a grid resolution of $k = 4$, the two-dimensional space was partitioned into 16 blocks, six of which are non-empty. To find the adjacent blocks of the six non-empty blocks, 24 other blocks (visualized by arrows) are considered. Only 6 out of these 24 blocks are non-empty (blue arrows). The plot on the right hand side of Figure 2 highlights all selected pairs by blue lines that connect the corresponding objects.

III. Geometric algorithms for sparse computation

To address the bottleneck of identifying pairs in adjacent blocks, we introduce a computational geometry concept called data shifting. We show how data shifting can be used to devise two geometric algorithms for sparse computation: object shifting and block shifting. These algorithms replace the block enumeration process in the second step of sparse computation. In Section III-A we introduce the general concept of data shifting. In Sections III-B and III-C, we present the object shifting and the block shifting algorithm, respectively. Finally, in Section III-D, we discuss the runtime performance of the algorithms.

A. *The concept of data shifting*

Sparse computation relies on a grid to identify close pairs of objects in the low-dimensional space. The identified pairs are all within an L_∞ distance of ω and potentially some within an L_∞ distance of 2ω. In contrast to sparse computation with block enumeration, the low-dimensional space is partitioned into k'^p grid blocks for $k' = \frac{1}{2\omega}$. Each grid block is thus twice as large in each dimension. All objects within a grid block are now within an L_∞ distance of 2ω. The grid, however, might still arbitrarily separate objects that are close by a grid line. Two objects that are within an L_∞ distance of ω, but separated by a grid line, are denoted as border pair. In a two-dimensional grid, there are three types of border pairs: horizontal border pairs, vertical border pairs, and diagonal border pairs. In a p-dimensional grid, there are $2^p - 1$ types of border pairs. Data shifting addresses the issue of identifying border pairs by shifting the data from its initial position along a single or multiple axes such that all border pairs of one type will no longer be separated by a grid line after the shift. To capture all types of border pairs, $2^p - 1$ shifts are required. In each shift, the data is shifted by ω along the respective axes. This is sufficient to identify all close neighbors of an object, since for each neighbor that is within a distance of ω from the object there exist at least one grid such that they belong to the same grid block. By efficiently identifying all close neighbors, data shifting provides a 2-approximation for the problem of identifying, for each object, the set of neighbors that are within an L_∞ distance of ω.

B. *Object shifting algorithm*

To get exactly the same selection of similar pairs as the block enumeration strategy with a grid resolution of k, the object shifting algorithm first partitions the low-dimensional space into k'^p grid blocks for $k' = \frac{1}{2}k$. Based on the concept of data shifting, $2^p - 1$ directions are determined. Given p, the directions can be determined by generating binary representations of width p of the integers $[1, \ldots, 2^p - 1]$. Each bit corresponds to a dimension and a one means that the data is shifted along this dimension. For example, if $p = 2$, the binary representations are 01, 10, and 11. The algorithm is called object shifting because all objects are shifted by $\frac{1}{k} = \frac{1}{2k'}$ in that direction. Based on the new coordinates, each object is assigned to a single grid block and all pairs of objects that are assigned to the same block are selected. Since the same pairs of objects might be assigned to the same block in different shifts, one needs to identify and remove duplicate pairs. If the grid resolution is high and hence the total number of selected pairs is low, the runtime for identifying and removing redundant pairs is negligible. However, if the grid resolution is low and the total number of selected pairs is large, then identifying and removing redundant pairs can become computationally expensive. Figure 3 illustrates object shifting algorithm for our two-dimensional example. The original position of the data is shown in the top left plot.

C. *Block shifting*

The disadvantage of object shifting is that certain pairs are selected for multiple shifts. In particular, objects that fall in

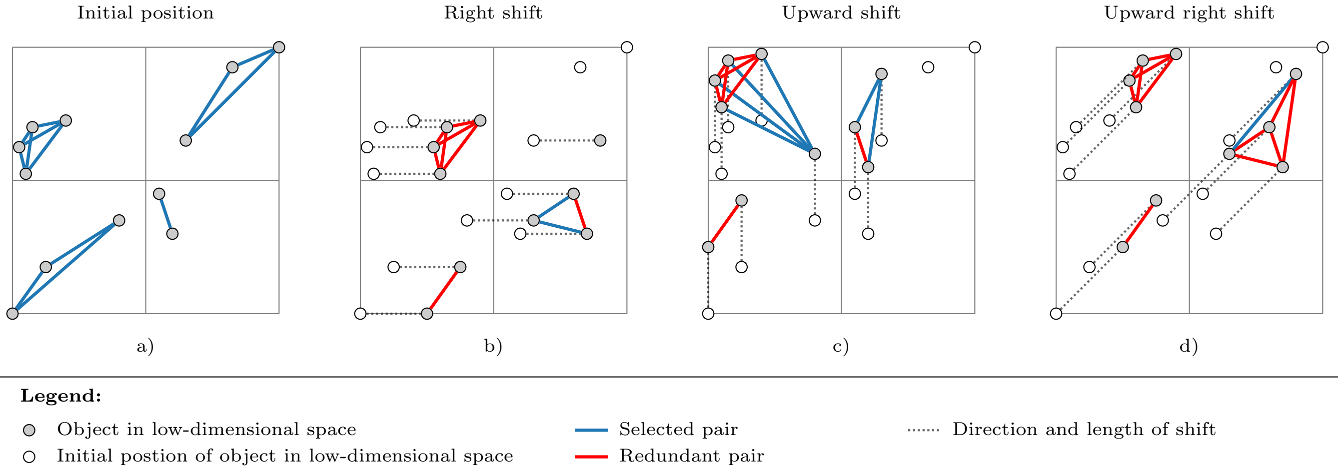

Figure 3. Visualization of the object shifting algorithm with grid resolution $k' = 2$ for a data set projected to a $p = 2$ dimensional space: a) selection of pairs of objects that fall in the same grid block, b) objects are horizontally shifted by $1/k$ to capture horizontal border pairs. c) objects are vertically shifted by $1/k$ to capture vertical border pairs, d) objects are diagonally shifted to capture diagonal border pairs. Redundant pairs are highlighted in red.

the same sub-block, defined by splitting the blocks in half in each dimension, will fall in the same block for each shift and are thus repeated $2^p - 1$ times. Block shifting addresses this by replacing them with a single object.

To generate the same pairs as sparse computation with block enumeration with grid resolution k, the p-dimensional space is first partitioned with a grid resolution of k into k^p grid blocks and each object is assigned to the corresponding block. The corresponding pairs for each block are selected. Instead of identifying the border pairs directly with data shifting each non-empty block is first replaced with a representative object at its center. Note that this new data set consists of objects that are located at multiples of $\frac{1}{2k}$ along the dimensions. All coordinates are within the range $[\frac{1}{2k}, 1 - \frac{1}{2k}]$ and the L_∞ distance between pairs of objects that represent adjacent blocks is exactly $\frac{1}{k}$. Hence, we can apply the object shifting algorithm described in Section III-B to the new data set with a grid resolution of $k' = \frac{1}{2}k$ to find all pairs of adjacent blocks. Finally, all pairs of objects that consists of objects in adjacent blocks are selected. Figure 4 illustrates block shifting for our two-dimensional example.

D. Runtime analysis

The runtime of block enumeration, object shifting, and block shifting depends on the parameters k and p and on the distribution of the objects within the low-dimensional space. Since all algorithms have the same output, they only differ with respect to the overhead (unnecessary work performed): The object/block shifting algorithms may identify pairs of objects/representatives more than once, and the block enumeration algorithm may enumerate empty adjacent blocks.

The overhead of the block enumeration is large when objects fall into many isolated blocks. This typically occurs for large k and p. The overhead of object shifting is large when many objects fall into few adjacent blocks. This typically occurs for small k and p. The overhead of block shifting is only large when the number of adjacent representatives is large. This does not occur for small or for large values of k and p. To obtain a large number of adjacent representatives, few objects must fall in a large number of adjacent blocks. Hence, the number of adjacent representatives tends to initially increase with increasing p and k, but decreases again once the blocks become isolated. In practice, we observe that block shifting identifies a pairwise similarity no more than twice. For more details, see the experimental results in section IV-C and Figure 6.

IV. Computational Analysis

We compared the runtime of block enumeration, object shifting, and block shifting for different grid resolutions on nine datasets. We evaluated only runtime because all algorithms generate the same output when the grid resolutions are chosen accordingly. We implemented the algorithms and the computational analysis in Python 3.5. The source code is available at https://github.com/hochbaumGroup/sparsecomputation. The source code of the computational analysis is available at https://github.com/quic0/sparse-experiments. In Section IV-A, we describe the data sets in detail. In Section IV-B, we present the experimental design, and in Section IV-C, we analyze the runtimes. Extensive empirical results on how sparse computation and thus the new algorithms affect the accuracy of (un)supervised learning algorithms are presented in [5], [9].

A. Data sets

The data sets represent various domains including life sciences, engineering, social sciences and business and have

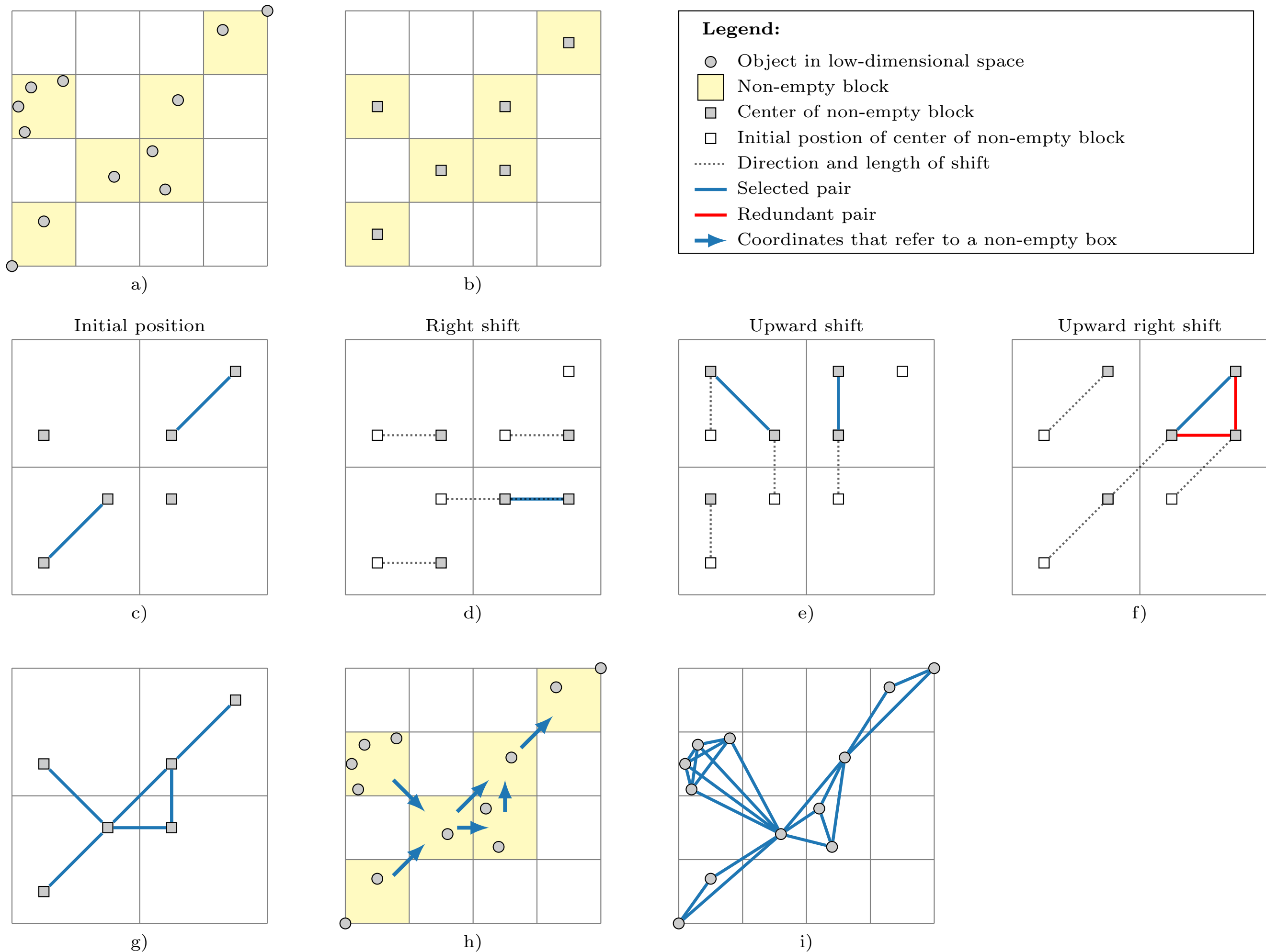

Figure 4. Visualization of the block shifting algorithm with grid resolution $k = 4$ for a data set projected to a $p = 2$ dimensional space: a) low-dimensional space is partitioned with grid resolution k, b) each non-empty block is replaced by a representative, c)–f) the object shifting algorithm is applied with grid resolution $k' = 2$ to find pairs of representatives that correspond to neighboring blocks (redundant pairs of representatives are highlighted in red), g)–i) the identified pairs of representatives are used to select pairs of objects that fall within the same or within neighboring blocks.

Table II
STATISTICS OF THE DATA SETS (AFTER MODIFICATION)

Abbreviation	# of objects	# of unique objects	# of features
NEU	961	961	800
LER	20,000	18,668	16
BOW1	41,361	37,993	33,781
ADU	45,222	45,170	88
COV	581,012	581,012	54
CKR	1,000,000	999,910	2
RLC	5,749,132	5,749,133	16
BOW2	8,544,543	3,087,117	100
RNG	10,000,000	10,000,000	20

sizes ranging from hundreds to millions of objects. Six real-world data sets were taken from the UC Irvine Machine Learning Repository [15], two datasets were taken from previous studies [16], [17], [18], [19], and one real-world data set was taken from the Neurofinder public benchmark for calcium imaging [20]. In all datasets, we substituted categorical features by a binary feature for each category and removed any duplicate objects. We briefly describe each data set and mention further modifications that we made. Statistics about each of the data sets are provided in Table II.

The data set *Adult* (ADU) [21] stems from the census bureau database and each object represents a person. In the original data set there is a categorical and a continuous feature to capture the educational level of the persons. To avoid double use, we removed the categorical feature. In addition, we removed all features that contain missing values.

In the *Bag of Words* data set (BOW), the objects are

documents from five different sources. The sources are Enron emails, KOS blog entries, New York Times articles, NIPS full papers, and PubMed abstracts. A document in this data set is represented as a bag of words, i.e., a set of vocabulary words. This representation disregards word order but keeps word multiplicity. We generated two data sets from the bag of words data set: BOW1 contains the NIPS full papers and the Enron emails. For each document in BOW1, we divide the number of occurrences of each word by the total number of words in the document to obtain the relative frequencies. BOW2 contains the New York Times articles and the PubMed abstracts. For BOW2, we limit the feature space to the 100 words with the largest absolute difference in mean relative frequency between the New York Times articles and the PubMed abstracts. A binary vector indicates which of those words occur in the respective document.

The data set *Checkerboard* (CKR) is an artificial data set that has been used for evaluating large-scale SVM implementations [18], [19]. Each object represents a point on a 4×4 checkerboard that is characterized by two features which represent the x- and the y-coordinates. Following [19], we create a checkerboard data set with 1 million objects by randomly choosing the x- and y-coordinates of objects between -2 and 2 according to a uniform distribution.

The data set *Covertype* (COV) [22] contains cartographic characteristics of forest cells in northern Colorado. There are seven different cover types which are labeled 1 to 7.

The data set *Letter Recognition* (LER) [23] comprises 20,000 objects. Each object corresponds to an image of a capital letter from the English alphabet.

The data set *Neuron* (NEU) [20] is a calcium imaging recording of a neuron, a brain cell. The objects are the pixels in the recording and the features are the intensity of a pixel for each of the frames. Sparse computation is used as a subroutine in a leading algorithm for cell identification in calcium imaging recordings [24].

In the data set *Record Linkage Comparison Patterns* (RLC) [25], the objects are comparison patterns of pairs of patient records. We substitute the missing values with value zero and introduce a binary feature to indicate missing values. The objects can be classified as match or no-match.

The data set *Ringnorm* (RNG) is an artificial data set that has been used in [16] and [17]. The objects are points in a 20-dimensional space and belong to one of two Gaussian distributions. Following the procedure of [17], we generate a Ringnorm data set instance with 10 million objects.

B. Experimental design

We compared the runtime of sparse computation with block enumeration, object shifting, and block shifting on the nine data sets for different grid resolutions k. For large data sets, it was impossible to construct the complete similarity matrix due to memory limitations of our machine. The grid resolutions used for each of the data sets are described in Table III. We chose the number of dimensions p of the low-dimensional space to be 3 for all data sets except for CKR for which we chose 2 because it only has two features. For the dimension reduction step, we used approximate PCA [5] which is much faster than exact PCA as it reduces the size of the original data set by sampling a small fraction of objects and features. Here we set the sampling fraction for objects to one percent and the sampling fraction for features to five percent unless the number of features is less than 150 in which case all features were used. Only the NEU and BOW1 datasets have more than 150 features.

Table III
GRID RESOLUTIONS REPORTED FOR EACH DATA SET

Abbreviation	Grid resolution k
NEU	4, 10, 20, 50, 100, 200
LER	4, 10, 20, 50, 100, 200
BOW1	4, 10, 20, 50, 100, 200, 500, 1000, 2000
ADU	4, 10, 20, 50, 100, 200, 500, 1000, 2000
COV	50, 100, 200, 500, 1000, 2000
CKR	100, 200, 500, 1000, 2000
RLC	200, 500, 1000, 2000, 4000, 10000
BOW2	200, 500, 1000, 2000, 4000, 10000
RNG	200, 500, 1000, 2000, 4000, 10000

We ran each combination of algorithm, data set, and grid resolution five times and recorded the mean runtime and the standard deviation across all five runs. In addition, we recorded various statistics that affect the runtime of the algorithms. The computational analysis was performed on a workstation with Intel Xeon CPUs (model E5-2667 v2) with clock speed 3.30 GHz and 256 GB of RAM.

C. Experimental results

The mean and standard deviations of the runtimes of the different algorithms are reported in Figure 5 and for the largest three data sets in Table IV. Sparse computation with block enumeration performs well at low grid resolutions, but gets slow when applied with high grid resolutions. In contrast, sparse computation with object shifting performs poorly at low grid resolutions because there is a large number of pairs consisting of objects that fall into the same sub-block and are selected for each shift. Sparse computation with object shifting performs well when the grid resolution is high. At these grid resolutions, few objects fall in the same sub-block and fewer duplicate pairs are selected.

Sparse computation with block shifting combines the properties of sparse computation with block enumeration and object shifting and provides (near-) best runtime across data sets and grid resolutions. Although outperformed by object shifting at extremely high grid resolutions, it does not suffer from the pitfalls that affect block enumeration and object shifting. By replacing each sub-block by a single representative, it significantly reduces the number of duplicate pairs. This results in faster runtimes for low grid resolutions when each block typically contains multiple

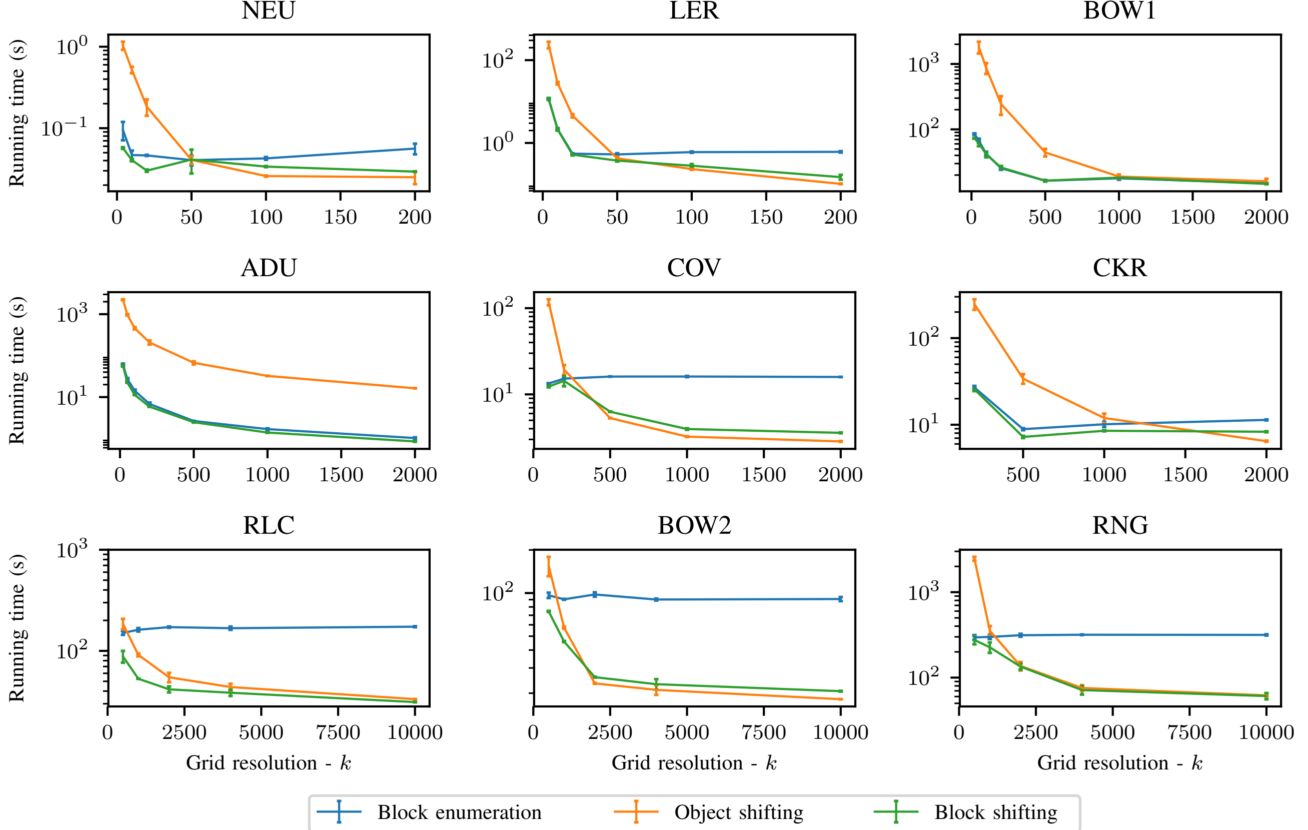

Figure 5. Mean runtimes, measured in log scale, across data sets for sparse computation with block enumeration, object shifting, and block shifting. The error bars indicate the standard deviation of the runtimes. Grid resolution is measured with respect to sparse computation with block enumeration.

Table IV
MEAN RUNTIME FOR RLC, BOW2, AND RNG FOR SELECTED GRID RESOLUTIONS

Algorithm	**Dataset**								
	RLC			BOW2			RNG		
	$k = 500$	$k = 2000$	$k = 10000$	$k = 500$	$k = 2000$	$k = 10000$	$k = 500$	$k = 2000$	$k = 10000$
Block enumeration	150 ± 6	172 ± 2	173 ± 1	96 ± 4	98 ± 3	91 ± 2	296 ± 18	313 ± 12	316 ± 5
Object shifting	185 ± 22	55 ± 5	33 ± 0.07	155 ± 23	**23** ± 0.2	**18** ± 0.04	2466 ± 122	137 ± 15	62 ± 2
Block shifting	**88** ± 11	**42** ± 2	**31** ± 0.2	**74** ± 0.6	26 ± 0.2	21 ± 0.1	**276** ± 30	**135** ± 10	**61** ± 5

objects. Figure 6 shows that block shifting identifies only few duplicate pairs for low grid resolutions. For all datasets block shifting identifies, on average, a pair no more than twice independent of the grid resolution.

An interesting insight can be gained from the CKR data set: although object shifting generates more duplicate pairs than block shifting, it is still slightly faster for very high grid resolutions. This is most likely due to the overhead resulting from generating the representatives and mapping representatives to objects. In the CKR data set, the objects are uniformly spread, which means that at a grid resolution of $k = 1,000$, there are 1 million blocks ($p = 2$), that are mostly non-empty. Since there are only 1 million objects, each block contains a single object in expectation. Therefore, block shifting gains little by replacing the objects in each block by a representative but still incurs the overhead.

The computational effort required by sparse computation also depends on the number of dimensions p of the low-dimensional space. For example, the number of (adjacent) blocks, and the number of required shifts for object shifting and block shifting grow exponentially in p. To explore the impact of the dimension of the low-dimensional space on runtime, we applied the different algorithms to the COV data set with $p = 2$, 3, and 4. As shown in Figure 7, the runtime of sparse computation with block enumeration increases steadily as p is increased even though the number

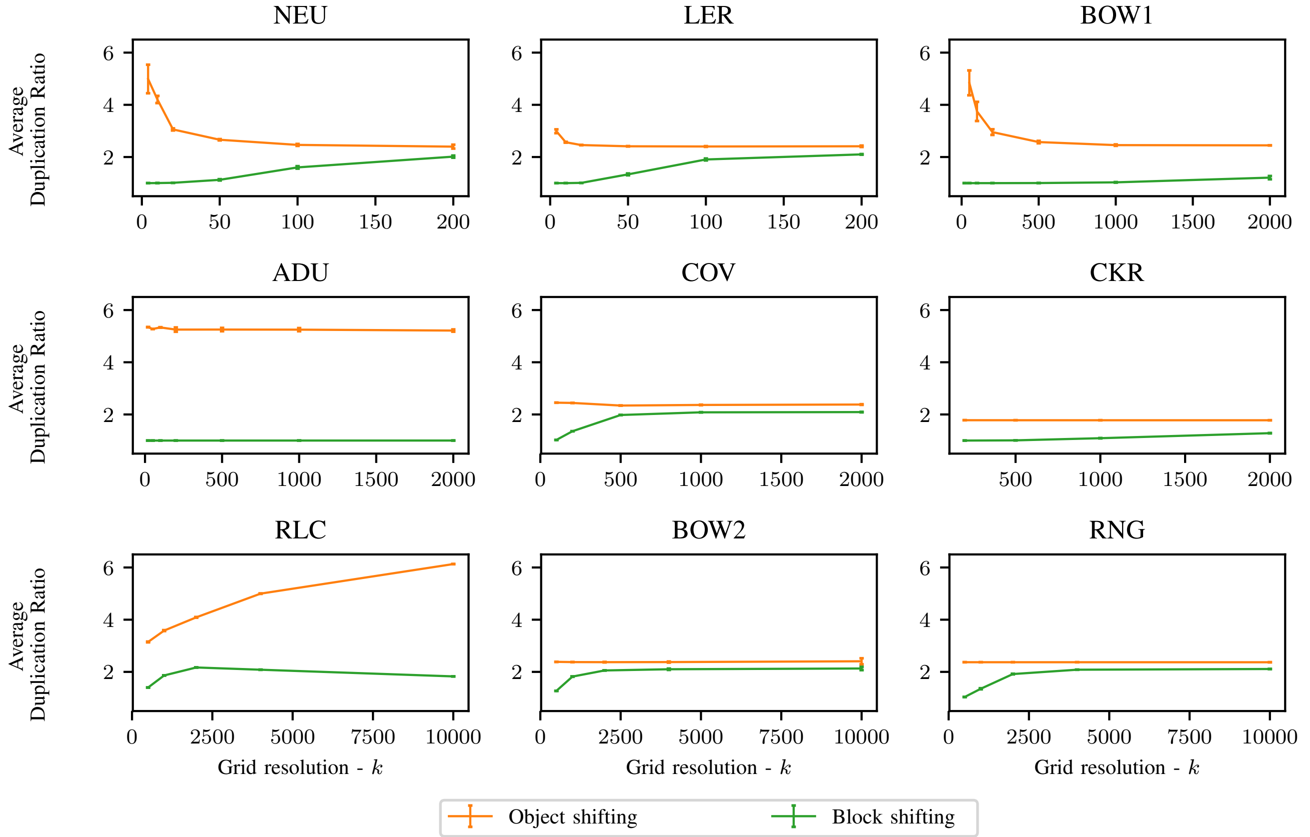

Figure 6. Average duplication ratio across data sets for sparse computation with block and object shifting. The duplication ratio is computed as `(# unique object pairs + # duplicate object pairs) / # unique object pairs` for object shifting and as `(# unique object pairs + # duplicate representative pairs) / # unique object pairs` for block shifting. The error bars indicate the standard deviation in the number of duplicate pairs. Grid resolution is measured with respect to sparse computation with block enumeration.

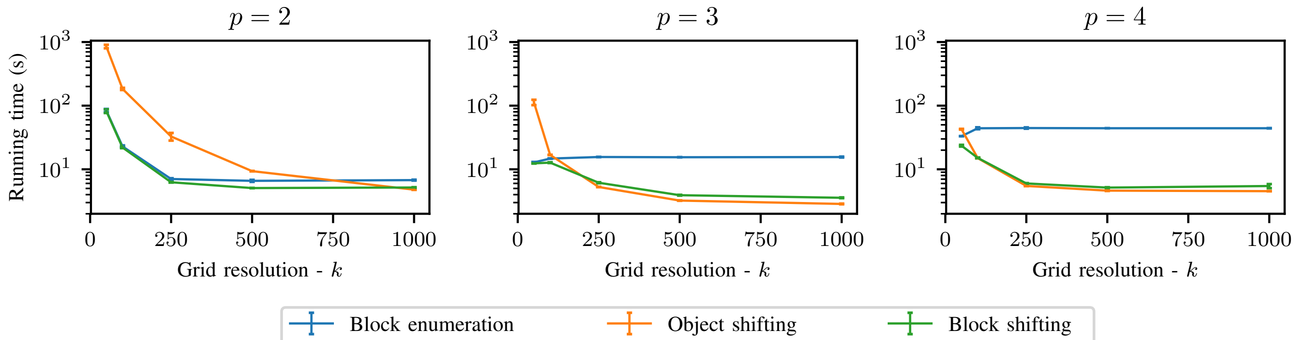

Figure 7. Effect of the number of dimensions in the low-dimensional space, p, on the mean runtimes of sparse computation with block enumeration, object shifting, and block shifting for the COV data set. The error bars indicate the standard deviation of the runtimes. Grid resolution is measured with respect to sparse computation with block enumeration.

of selected pairs decreases with increasing grid resolution. Sparse computation with object shifting and block shifting scales much better with increasing values of p.

V. Conclusions

We introduced a new computational geometry concept called data shifting that allows to efficiently find pairs of similar objects in a low-dimensional space. Proximity of objects is recognized by performing multiple shifts of the data. Two new algorithms are devised that use data shifting to speed up sparse computation which is a recently introduced technique for finding similar objects also in high-dimensional space. Based on real-world data sets with up to 10 million objects, we demonstrate that sparse computation with block shifting provides overall the best runtime performance across grid resolutions for data sets of up to 10 million objects. Python implementations of all algorithms are publicly available at https://github.com/hochbaumGroup/sparsecomputation. Promising directions for future research include the application of the proposed data shifting concept in other contexts such as approximate nearest neighbor search and grid-based clustering.

References

[1] D. S. Hochbaum, "Polynomial time algorithms for ratio regions and a variant of normalized cut," *IEEE transactions on pattern analysis and machine intelligence*, vol. 32, no. 5, pp. 889–898, 2010.

[2] C. Cortes and V. Vapnik, "Support vector machine," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[3] P. Baumann, D. S. Hochbaum, and Y. T. Yang, "A comparative study of the leading machine learning techniques and two new optimization algorithms," *UC Berkeley manuscript*, 2017.

[4] H. Kawaji, Y. Takenaka, and H. Matsuda, "Graph-based clustering for finding distant relationships in a large set of protein sequences," *Bioinformatics*, vol. 20, no. 2, pp. 243–252, 2004.

[5] D. S. Hochbaum and P. Baumann, "Sparse computation for large-scale data mining," *IEEE Transactions on Big Data*, vol. 2, no. 2, pp. 151–174, 2016.

[6] S. Arora, E. Hazan, and S. Kale, "A fast random sampling algorithm for sparsifying matrices," in *Approximation, Randomization and Combinatorial Optimization: Algorithms and Techniques*. Springer, Berlin, 2006, pp. 272–279.

[7] C. Jhurani, "Subspace-preserving sparsification of matrices with minimal perturbation to the near null-space. part i: Basics," *arXiv preprint arXiv:1304.7049*, 2013.

[8] D. A. Spielman and S.-H. Teng, "Spectral sparsification of graphs," *SIAM Journal on Computing*, vol. 40, no. 4, pp. 981–1025, 2011.

[9] P. Baumann, "Sparse-reduced computation for large-scale spectral clustering," in *IEEE International Conference on Industrial Engineering and Engineering Management*. IEEE, 2016, pp. 1284–1288.

[10] P. Baumann, D. S. Hochbaum, and Q. Spaen, "Sparse-reduced computation," in *Proceedings of the 5th International Conference on Pattern Recognition Applications and Methods*, 2016, pp. 224–231.

[11] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the twentieth annual symposium on Computational geometry*. ACM, 2004, pp. 253–262.

[12] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," in *47th Annual IEEE Symposium on Foundations of Computer Science*. IEEE, 2006, pp. 459–468.

[13] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[14] H.-L. Chen and Y.-I. Chang, "All-nearest-neighbors finding based on the hilbert curve," *Expert Systems with Applications*, vol. 38, no. 6, pp. 7462–7475, 2011.

[15] M. Lichman, "UCI machine learning repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml

[16] L. Breiman, "Bias, variance, and arcing classifiers," 1996.

[17] J.-x. Dong, A. Krzyzak, and C. Y. Suen, "Fast svm training algorithm with decomposition on very large data sets," *IEEE transactions on pattern analysis and machine intelligence*, vol. 27, no. 4, pp. 603–618, 2005.

[18] Y.-J. Lee and O. L. Mangasarian, "Rsvm: Reduced support vector machines," in *Proceedings of the 2001 SIAM International Conference on Data Mining*. SIAM, 2001, pp. 1–17.

[19] I. W. Tsang, J. T. Kwok, and P.-M. Cheung, "Core vector machines: Fast svm training on very large data sets," *Journal of Machine Learning Research*, vol. 6, no. Apr, pp. 363–392, 2005.

[20] "The neurofinder challenge," 2017, accessed: August 5, 2017. [Online]. Available: http://neurofinder.codeneuro.org

[21] R. Kohavi, "Scaling up the accuracy of naive-bayes classifiers: A decision-tree hybrid." in *KDD*, vol. 96, 1996, pp. 202–207.

[22] J. A. Blackard and D. J. Dean, "Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables," *Computers and electronics in agriculture*, vol. 24, no. 3, pp. 131–151, 1999.

[23] P. W. Frey and D. J. Slate, "Letter recognition using holland-style adaptive classifiers," *Machine learning*, vol. 6, no. 2, pp. 161–182, 1991.

[24] Q. Spaen, D. S. Hochbaum, and R. Asín-Achá, "Hnccorr: A novel combinatorial approach for cell identification in calcium-imaging movies," *arXiv preprint arXiv:1703.01999*, 2017.

[25] I. Schmidtmann, G. Hammer, M. Sariyar, A. Gerhold-Ay, and K. des öffentlichen Rechts, "Evaluation des krebsregisters nrw schwerpunkt record linkage," *Abschlußbericht vom*, vol. 11, 2009.