Improved Greedy Algorithms for Sparse Approximation of a Matrix in terms of Another Matrix

Crystal Maung
Department of Computer Science
The University of Texas at Dallas

Haim Schweitzer
Department of Computer Science
The University of Texas at Dallas

Abstract

We consider simultaneously approximating all the columns of a data matrix in terms of few selected columns of another matrix that is sometimes called “the dictionary”. The challenge is to determine a small subset of the dictionary columns that can be used to obtain an accurate prediction of the entire data matrix. Previously proposed greedy algorithms for this task compare each data column with all dictionary columns, resulting in algorithms that may be too slow when both the data matrix and the dictionary matrix are large. A previously proposed approach for accelerating the run time requires large amounts of memory to keep temporary values during the run of the algorithm. We propose two new algorithms that can be used even when both the data matrix and the dictionary matrix are large. The first algorithm is exact, with output identical to some previously proposed greedy algorithms. It takes significantly less memory when compared to the current state-of-the-art, and runs much faster when the dictionary matrix is sparse. The second algorithm uses a low rank approximation to the data matrix to further improve the run time. The algorithms are based on new recursive formulas for computing the greedy selection criterion. The formulas enable decoupling most of the computations related to the data matrix from the computations related to the dictionary matrix.

1 Introduction

Representing large amounts of data in terms of a small number of atoms taken from a given dictionary is computationally challenging. Greedy algorithms that solve this problem are routinely applied in areas such as signal processing and machine learning (e.g., [1, 2, 3, 4, 5]), where the amount of data and the size of the dictionary are manageable. But applying previously proposed algorithms to large databases may be impractical.

Consider data consisting of $N$ items and a dictionary consisting of $n$ atoms. Previously proposed greedy algorithms (e.g., [6, 7]) compare each data item with all dictionary atoms, leading to run time that is at least proportional to $nN$. In fact, taking into account the comparison time the run time of a naive implementation is $O(kmnN)$, where $k$ is the selection size and $m$ is the size of a single data item. While there are many studies that analyze the accuracy of the greedy scheme (e.g., [6, 7, 8, 9, 10, 11]) there is a relatively small amount of published work aimed at reducing the run time of the greedy algorithms. We are aware of the studies in [7, 12]. The study in [12] addresses the number of passes over the data, but it is applied only to the case where the data matrix has a single column. As discussed in Section 2, the algorithm established in [7] by Civril and Magdon-Ismail, that we call “The CM Algorithm”, reduces the run time to $O(mnN)$ by keeping in memory $nN$ intermediate values. Additional savings can be obtained by taking advantage of the dictionary sparsity. The CM can take advantage of sparse dictionaries when the average number of non-zeros
in each atom is reduced from \( m \) to no less than \( k \). As explained in Section 2.7 sparser dictionaries do not yield an improved asymptotic run time for the CM.

We propose two new greedy algorithms. They require significantly less memory than previously proposed algorithms, retaining in memory only two values for each column of the dictionary. This compares favorably with previous algorithms that typically require partial orthogonalization of the dictionary matrix. When the dictionary matrix is sparse, the partial orthogonalization is non-sparse, so that \( m \) values may have to be retained for each column of the dictionary. Furthermore, unlike the CM our algorithms can take advantage of higher levels of dictionary sparsity to reduce their asymptotic complexity. Their asymptotic run time is reduced with higher levels of sparsity as long as the number of non-zeros is above \( km \). This should be compared with the CM that cannot benefit from sparsity levels where the number of non-zeros falls below \( kn \), and with the naive method that cannot take advantage of sparse dictionaries.

The first proposed algorithm is exact, producing the same output as other greedy algorithms. The second algorithm uses a low rank approximation of the data matrix to further improve the run time. The result is no longer identical to exact greedy algorithms, but it is very similar and allows for much faster run time.

The proposed algorithms make use of the (surprising) observation that the computations related to the data matrix can “almost” be decoupled from the computations related to the dictionary matrix. Specifically, we derive recursive formulas that show how to determine the selection of the \( j+1 \)th atom using mostly computations that were already performed when the \( j \)th atom was selected. Since what remains to be calculated is independent of \( N \), the computations involving \( N \) are decoupled from the computations involving \( n \). Using these recursive formulas, which to the best of our knowledge are new, the only computations that involve both \( n \) and \( N \) are those performed while selecting the first atom.

The first algorithm that we propose, the SOLS-exact, computes the values necessary to select the first atom, and then applies the recursive formulas to select the additional atoms. When applied to dense dictionaries it has the same asymptotic complexity as the CM, but it uses significantly less memory. As discussed earlier it performs better than the CM with sparse dictionaries. For example, if the average number of nonzero entries in each dictionary column is a constant independent of the column size, the run time of the SOLS-exact (under the assumption that \( km \) is not too big) would be \( O(nN) \). This is significantly better than the \( O(knN) \) of the CM and the \( O(kmnN) \) of the naive approach under the same assumptions.

The second algorithm that we propose, the SOLS-lowrank, improves over the first algorithm by replacing the data matrix with a low rank approximation. Recent studies (e.g., [13, 14, 15, 16]) have shown that low rank approximations can be computed very rapidly with randomized algorithms, so that the run time of the proposed algorithm is typically not affected by the complexity of computing the low rank approximation. When the data matrix is approximated by a rank \( d \) matrix, the complexity of running the second algorithm is \( O(dmn) \) for dense dictionaries, and \( O(dn) \) for very sparse dictionaries (assuming again that \( km \) is not too big compared to \( n \)).

1.1 Problem statement and related work

Let \( Y \) be a data matrix of \( m \) rows and \( N \) columns. Its columns are \( y_1, \ldots, y_N \), where a column \( y_i \) is an \( m \) dimensional vector. Similarly, let \( X \) be a “dictionary” matrix of \( m \) rows and \( n \) columns. Its columns, also called “atoms”, are \( x_1, \ldots, x_n \), where a column \( x_i \) is an \( m \) dimensional vector. We consider the problem of selecting a subset of \( k \) columns from \( X \) that can be used to approximate all the columns of \( Y \). Specifically, suppose the columns \( x_{s_1}, \ldots, x_{s_k} \) are selected from \( X \), then each column \( y_i \) of \( Y \) can be approximated by the linear combination:

\[
y_i \approx a_{i,1}x_{s_1} + \ldots + a_{i,k}x_{s_k}
\]

where \( a_{i,1}, \ldots, a_{i,k} \) are coefficients associated with \( y_i \) and chosen to minimize the approximation error. In matrix form this can be written as:

\[
Y \approx SA
\]  

(1)

where \( S = (x_{s_1}, \ldots, x_{s_k}) \) is the \( m \times k \) selection matrix, and the coefficients matrix \( A \) is \( k \times N \).

The quality of the selection \( S \) is determined by the following error:

\[
E = \min_A \|Y - SA\|_F^2.
\]  

(2)
Figure 1: The relevant complexity parameters. The data matrix $Y$ is $m \times N$, the dictionary matrix $X$ is $m \times n$, and the selection matrix $S$ is $m \times k$. The matrix $X$ has $z_x$ nonzero elements. The matrix $Y$ may be approximated by a $d$-rank matrix.

<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0. Global pre-calculations (done once).</td>
</tr>
<tr>
<td>1. Run iteration $j = 1, \ldots, k$:</td>
</tr>
<tr>
<td>1.0 Pre-calculations for Iteration-$j$.</td>
</tr>
<tr>
<td>1.1 for $i = 1, \ldots, n$: evaluate $f(x_i)$.</td>
</tr>
<tr>
<td>1.2 $x_{s_j} = \text{arg max } f(x_i)$. Add $x_{s_j}$ to $S$.</td>
</tr>
<tr>
<td>1.3 Post-calculations for Iteration-$j$.</td>
</tr>
</tbody>
</table>

Figure 2: Algorithm GREEDY. The greedy framework of algorithms discussed in the paper.

where the matrix norm is the Frobenius norm. We are interested in the case where $n$ (the dictionary size) is big and $k$ (the selection size) is small, as illustrated in Fig. 1. This approximation and closely related variants have attracted a lot of attention. Optimal solutions, as well as approximations within a constant are known to be NP-hard even for the $N=1$ case [17, 18]. There are two common approaches for computing an approximate solution. The first approach recasts the approximation as an $l_1$ convex optimization (e.g. [19, 20]). The second approach is greedy, selecting the $k$ columns from $X$ one after the other (e.g. [6, 7, 8, 10, 11, 17, 21, 22]). The $l_1$ algorithms typically use linear programming, and there is ample evidence (e.g. [12]) that they are much slower than the greedy approach. In this paper we consider only the greedy approach.

Referring to the matrix $X$ as a dictionary is common in the signal processing literature (e.g. [1]), where in the heavily analyzed case of $N=1$ the matrix $Y$ is called “the signal”. The case $N=1$ is also common in machine learning and related studies, where it is known as feature selection in linear regression (e.g. [2, 3, 5]).

Another heavily researched special case has $X=Y$, taking the dictionary to be the same as the data matrix. Here the challenge is to approximate the data matrix $Y$ by a small subset of its own columns. This is sometimes called “unsupervised feature selection” in machine learning (e.g. [2]) and “column subset selection” in computational linear algebra (e.g. [23]). Computing the optimal solution for this special case is believed to be NP-hard [24], although there are efficient algorithms with a guaranteed good approximation. See, e.g. [25, 26].

1.2 Paper organization

The paper is organized as follows. Section 2 reviews previously proposed greedy algorithms for the sparse matrix approximation. Their run time and memory requirements are summarized in a table shown in Fig. 3. Our main result is based on recursive formulas, derived in Section 3, for solving the SOLS variant. The improved algorithms are presented in sections 4 and 5. Experimental results are discussed in Section 6.

2 Greedy Algorithms

In this section we describe the main variants of previously proposed greedy algorithms for computing a sparse approximation in terms of dictionary atoms. Algorithm GREEDY in Fig. 2 is the basic greedy approach for selecting a subset of $k$ columns from the matrix $X$ to estimate the matrix $Y$. It includes as special cases all the algorithms discussed in this paper. For clarity, our presentation of
The Matching Pursuit algorithm (MP) was proposed in [21]. It is applied with

\[ f(x) \text{ (the selection criterion) } kn \text{ times.} \]

in estimating the required amount of memory we assume that the storage of

the input matrices \( X \) and \( Y \) is “free”, but that this storage cannot be reused. This assumption is

justified in situations where the matrices are sparse, or when they are stored in a secondary memory

that cannot be (cheaply) overwritten.

### 2.1 Orthogonal projections

The following definitions and notation are useful in describing the various algorithms. We write \( \text{Orthogonal}(x, G) \) for the error (vector) of estimating the vector \( x \) in the column space of the matrix \( G \). It is defined as follows:

\[
\text{Orthogonal}(x, G) = x - Ga
\]

where \( a \) minimizes \( |x - Ga|^2 \)

The vector \( Ga \) is the best \( l_2 \) approximation of the vector \( x \) in the column space of \( G \). A common technique for computing orthogonal projections (see [23]) is to compute a matrix \( Q \) with columns that form an orthonormal basis to the column space of \( G \). This can be done, for example, by the QR factorization of \( G \). Given such \( Q \) we have:

\[
\text{Orthogonal}(x, G) = \text{Orthogonal}(x, Q) = x - QQ^T x = x - \sum q_j q_j^T x
\]

where the summation is over all the columns of \( Q \). In particular we have the following recursion:

\[
\text{Orthogonal}(x, Q_j) = \text{Orthogonal}(x, Q_{j-1}) - q_j q_j^T x
\]

where \( Q_j = (Q_{j-1}, q_j) \). We write \( \text{Orthonormal}(x, G) \) for the direction of \( \text{Orthogonal}(x, G) \). It is defined as follows:

\[
\text{Orthonormal}(x, G) = \frac{r}{|r|}, \quad r = \text{Orthogonal}(x, G)
\]

### 2.2 The MP algorithm

The Matching Pursuit algorithm (MP) was proposed in [21]. It is applied with \( N=1 \), where the data

matrix \( Y \) consists of a single vector \( y \). The changes in Algorithm GREEDY are:

\[
0. \quad \text{Normalize } X: x_i \leftarrow x_i/|x_i|. \quad \text{Set } r_0 = y.
\]

\[
1.1 \quad \text{for } i = 1, \ldots, n: \quad f(x_i) = |r_{j-1}^T x_i|.
\]

\[
1.3 \quad a = x_j^T r_{j-1}, \quad r_j = r_{j-1} - ax_j.
\]

The algorithm run time is typically dominated by Step 1.1, which computes \( f(x) \) (the selection criterion) \( kn \) times. The algorithm run time is justified in situations where the matrices are sparse, or when they are stored in a secondary memory that cannot be (cheaply) overwritten.
Observe that the value of $f(x_i)$ is inversely related to the angle between $x_i$ and the residual $r_{j-1}$. Thus, the selection of $x_{s_j}$ in Step 1.2 is the dictionary column “most similar” to (having the smallest angle with) the residual vector $r_{j-1}$. Step 1.3 updates the residual by removing the projection on the selected vector $x_{s_j}$ from the current residual vector.

Using all coordinates of $x_i$ for computing the dot product in 1.1 gives $O(kmn)$ run time. When $X$ is sparsely represented by $x_i$ nonzero values the dot products can be computed more efficiently and the run time reduces to $O(knz_j)$. Since only the vectors $y, r_j, x_j$ need to be kept in memory the amount of memory needed by the MP algorithm is $O(m)$.

### 2.3 The OLS algorithm

The Orthogonal Matching pursuit (OMP) is only slightly more expensive than the MP but it is expected to select a better set of dictionary columns. For theoretical analysis see, e.g., [6, 10, 11]. For applications in signal processing see, e.g., [1].

The OMP is applied with $N=1$, so that the data matrix $Y$ consists of a single vector $y$. It uses the same selection criterion as the MP, but calculates the residual vector as the orthogonal projection. As discussed in Section 2.1 this can be done efficiently by maintaining an orthogonal matrix $Q$ whose columns form an orthonormal basis to the selected vectors. With this implementation the changes to Algorithm GREEDY are:

0. Normalize $X$: $x_i \leftarrow x_i/|x_i|$. Set $r_0 = y$.
1.1 for $i = 1, \ldots, n$: $f(x_i) = |r_{j-1}^T x_i|$.
1.3 $q_j = \text{Orthonormal}(x_{s_j}, Q_{j-1})$
   $Q_j = (Q_{j-1}, q_j)$
   $a = q_j^T r_{j-1}$, $r_j = r_{j-1} - a q_j$

The calculations in Step 1.1 dominate the run time of the algorithm so that it is essentially the same as the run time of the MP, namely $O(kmn)$. The OMP requires more memory to keep and maintain the $m \times k$ matrix $Q$, leading to $O(km)$ memory requirement.

### 2.4 The OLS algorithm

The Orthogonal Least Squares (OLS) algorithm is only slightly more expensive than the OMP but it is expected to select a better set of dictionary columns. As the MP and the OMP it is implemented for $N=1$ so that the data matrix $Y$ consists of a single vector $y$. We use the name OLS following [10, 28]. The algorithm has sometimes been called OMP (e.g. [29]), OOMP (e.g. [22]), and Forward Selection (e.g. [3]) among others. See [28] for additional details about the naming of the algorithm. For a theoretical analysis of the OLS see, e.g. [10, 17]. For applications to feature selection see, e.g. [2, 3]. For applications related to signal processing see, e.g. [4]. For applications related to computer vision see, e.g. [30]. For applications related to numerical linear algebra see, e.g. [17].

The main idea behind the OLS is to select at each iteration the vector that reduces the prediction error the most. Let $e$ be the squared error of predicting $y$ from the selection $S$, and let $Q$ be an orthonormal basis for the column space of $S$. We have:

$$e = \min_a |y - Sa|^2 = |\text{Orthogonal}(y, S)|^2$$
$$= |y - QQ^T y|^2 = |y|^2 - |Q^T y|^2$$

This shows that minimizing $e$ can be achieved by finding $Q$ that maximizes $|Q^T y|$. The greedy search can be simplified by noticing the following recursion. Let $Q_j$ denote the submatrix with the first $j$ columns of $Q$. Then:

$$|Q_j^T y|^2 = |Q_{j-1}^T y|^2 + |q_j^T y|^2$$

where $q_j$ is the last column of $Q_j$. This shows that the best greedy selection of $x_{s_j}$ is the vector $x_{s_j}$ satisfying the following criterion:

$$x_{s_j} = \arg\max_{x_i} |q_j^T y|, \quad q_j = \text{Orthonormal}(x_i, Q_{j-1})$$

(3)
When \( q_i \) is orthogonal to \( Q_{j-1} \) we have: \( q_i^T y = q_i^T r_{j-1} \), where \( r_{j-1} = \text{Orthogonal}(y, Q_{j-1}) \).

Therefore the local optimality criterion (3) can also be expressed as:

\[
x_{s_j} = \arg \max_{x_i} |q_i^T r_{j-1}|
\]

\[\begin{align*}
q_i &= \text{Orthonormal}(x_i, Q_{j-1}) \\
r_{j-1} &= \text{Orthogonal}(y, Q_{j-1})
\end{align*}\]  

(4)

Using either (3) or (4) requires partial orthogonalization of the entire matrix \( X \). It is common to use the recursion specified in Section 2.1 and replace the vector \( x_i \) with its partial orthogonalization. A typical implementation of the OLS using the local optimality criterion (4) (e.g. [17]) can be achieved with the following changes to Algorithm GREEDY:

\[
0. \text{ Set } r_0 = y.
\]

1.1 for \( i = 1, \ldots, n \):

\[
x_i \leftarrow x_i - q_j q_j^T x_i, \quad f(x_i) = |r_{j-1}^T x_i|/|x_i|.
\]

1.3 \( q_j = \text{Orthonormal}(x_{s_j}, Q_{j-1}) \)

\[
Q_j = (Q_{j-1}, q_j) \\
a = q_j^T r_{j-1}, \quad r_j = r_{j-1} - a q_j
\]

The algorithm run time is dominated by Step 1.1, that takes about twice the time of the corresponding Step 1.1 of the OMP. What makes the OLS significantly more costly is the fact that the vectors \( x_i \) in Step 1.1 are not the original dictionary columns, since they have been partially orthogonalized with respect to the previously selected vectors. The partial orthogonalization results in loss of sparsity, and the need for additional memory to store the partially orthogonalized dictionary. This requires additional \( mn \) memory, with no run time savings for sparse dictionaries.

### 2.5 The SOMP algorithm

The Simultaneous Orthogonal Matching Pursuit algorithm (SOMP) is a direct generalization of the OMP for handling a data matrix of \( N \) columns [8]. The changes to Algorithm GREEDY are:

\[
0. \text{ Normalize } X: \quad x_i \leftarrow x_i/|x_i|. \text{ Set } R_0 = Y.
\]

1.1 Let \( r_{j-1}, \ldots, r_{j,N} \) be \( R_j \) columns,

for \( i = 1, \ldots, n \):

\[
f(x_i) = \sum_{t=1}^N |r_{j-1,t}^T x_i|.
\]

1.3 \( q_j = \text{Orthonormal}(x_{s_j}, Q_{j-1}) \)

\[
Q_j = (Q_{j-1}, q_j) \\
for t = 1, \ldots, N : \\
R_j = R_{j-1} - q_j q_j^T R_{j-1}
\]

As in the previous cases the calculations in Step 1.1 dominate the run time. They require \( k n N \) dot products of \( m \)-vectors that take \( O(k m n N) \). This can be reduced to \( O(k z N) \) when the dictionary is sparse. The amount of memory used by the algorithm to keep the matrices \( Q \) and \( R \) is \( k m + m N \).

### 2.6 The SOLS algorithm

The Simultaneous Orthogonal Least Squares algorithm (SOLS) is a direct generalization of the OLS for handling a data matrix of \( N \) columns. The algorithm is analyzed in [7, 9, 27].

As in the case of the OLS the column selected from the dictionary is the one that reduces the prediction error the most. The following discussion mirrors the discussion in 2.4. Let \( e_i \) be the squared error of predicting \( y_i \) from the selection \( S \), and let \( Q \) be an orthonormal basis for the column space of \( S \). The global error \( E \) is defined by:

\[
E = \sum_{i=1}^N e_i,
\]

so that \( E = \sum_i |y_i|^2 - \sum_i |Q^T y_i|^2 \). Therefore the minimum of \( E \) can be achieved by maximizing \( \sum_{i=1}^N |Q^T y_i|^2 \). Let \( Q_j \) be the submatrix of the first \( j \) columns in \( Q \), and let \( q_j \) be the last column of
The implementation of the SOLS using the local optimality criterion (6) can be achieved with the where the matrix norms are the Frobenius norms. This shows that the best greedy selection of the local optimality criterion (5) can also be expressed as:

\[ x_{s_j} = \arg \max_{x_i} |Y^T q_i|, \quad q_i = \text{Orthonormal}(x_i, Q_{j-1}) \]  

Since \( q_i \) is orthogonal to \( Q_{j-1} \) we have: \( Y^T q_i = R_{j-1}^T q_i \) where \( R_{j-1} = Y - Q_{j-1} Q_{j-1}^T Y \). Therefore the local optimality criterion (5) can also be expressed as:

\[ x_{s_j} = \arg \max_{x_i} |R_{j-1}^T q_i| \]

\[ q_i = \text{Orthonormal}(x_i, Q_{j-1}) \]

\[ R_{j-1} = Y - Q_{j-1} Q_{j-1}^T Y \]

The implementation of the SOLS using the local optimality criterion (6) can be achieved with the following changes to Algorithm GREEDY:

0. Set \( R_0 = Y \).

1. for \( i = 1, \ldots, n \):
   
   \( x_i \leftarrow x_i - q_i q_i^T x_i, \ f(x_i) = |R_{j-1}^T x_i|/|x_i|. \)

1.3 \( q_j = \text{Orthonormal}(x_{s_j}, Q_{j-1}) \)

\[ Q_j = (Q_{j-1}, q_j) \]

\[ R_j = R_{j-1} - q_j q_j^T R_{j-1} \]

As in previous cases the calculations in Step 1.1 dominate the run time. This step is visited \( kn \) times, and the matrix-vector product costs \( O(mN) \). Therefore the run time is \( O(kmnN) \). It cannot be reduced when the dictionary \( X \) is sparse since the partial orthogonalization of \( X \) typically destroys its sparsity.

2.7 The CM algorithm

Civril and Magdon-Ismail describe in [7] a fast implementation of the SOLS algorithm that we call The CM Algorithm. Their motivation was column subset selection, which they achieved as follows. Suppose the goal is to select \( k \) columns from a matrix \( A \). Set \( A \) to be the dictionary matrix \( X \), and construct the data matrix \( Y \) consisting of the \( k \) dominant singular vectors of \( A \), scaled by the corresponding singular values. It is shown in [7] that with this setting running the SOLS gives a good solution to column subset selection, as discussed in Section 1.1. To obtain a fast algorithm the paper establishes the CM, an optimized SOLS algorithm.

The proposed optimization retains in memory the values of the dot products computed in Step 1.1 of the SOLS, so that they need only be updated after each iteration. There are \( nN \) dot products to be maintained, and computing their initial values has an associated cost of \( O(mnN) \). The paper shows that these updates can be done fast, with an associated cost of \( O(nN) \) per iteration. Thus, the updates cost a total of \( O(knN) \). Adding the initialization cost and the updates cost gives an algorithm with run time of \( O((k + m)nN) \). The sparsity of \( X \) can be used during the initialization, but not during the updates where \( X \) is partially orthogonalized. This reduces the run time to \( O(knN + z_xN) \) for a sparse \( X \).

Examining Fig. 3 we see that the CM improves the run time by a factor of \( k \) over the naive SOLS. The improvement for a sparse \( X \) is even more impressive. When \( z_x \leq kn \) the run time of the CM is \( O(knN) \), better than the naive SOLS by a factor of \( m \).
2.8 Our results

As mentioned in Section 1 the algorithms we describe are based on formulas that allow recursive efficient calculations of the selection criterion. The algorithms differ by how they evaluate the initial values that are needed to select the first dictionary column. The exact version of the algorithm that we call “ISOLS-exact” produces the same output as the SOLS and the CM. It runs in time $O(mnN)$ on dense dictionaries, and in $O(kmN + z_N N)$ on sparse dictionaries. The algorithm keeps two values for each column of the dictionary in addition to the matrix $Q$, that is of size $m \times k$.

The second version of the algorithm that we call “ISOLS-lowrank” approximates the data matrix and produces results that may not be identical to the “ISOLS-exact”. But it has a much faster run time.

Comparing these algorithms to the CM we observe the following.

- The memory requirements of the CM are heavier than the memory requirements of our algorithms.
- Assuming the same low rank approximation for the data matrix by the CM, that can reduce the term $N$ to $d$, we consider the case where $z_N \approx n$. The complexity of the CM is roughly $O(dkn)$, which is worse than the $O(dn)$ of the ISOLS-lowrank by a factor of $k$.

3 Recursive SOLS criterion

Our main result is described in this section. We analyze the selection criterion in Line 1.1 of the SOLS algorithm and show that it can be calculated recursively.

The matrix $Q_{j-1} = (q_1, \ldots, q_{j-1})$ is known at Line 1.1 of the SOLS. The vector $q_i^j$ is computed for a candidate vector $x_i$ at Iteration-$j$ by:

$$r^j_i = x_i - Q_{j-1}^T q_{j-1} x_i, \quad q^j_i = \frac{r^j_i}{|r^j_i|}$$  \hspace{1cm} (7)

Applying the criterion in (5) the selection is computed by:

$$x_{s_j} = \arg \max_{x_i} |Y^T q^j_i|^2$$

$$= \arg \max_{x_i} \frac{|Y^T r^j_i|^2}{|r^j_i|^2} = \arg \max_{x_i} \frac{u^j_i}{v^j_i}$$  \hspace{1cm} (8)

where $u^j_i = |Y^T r^j_i|^2$, $v^j_i = |r^j_i|^2$. We proceed to show that both $u^j_i$ and $v^j_i$ can be computed recursively. The recursion for $v^j_i$ is well known, but to the best of our knowledge the recursion for $u^j_i$ is new. We state the result as a theorem.

**Theorem.** Let $Y$ be $m \times N$, let $Q_j$ be $m \times j$ with orthonormal columns, and let $q_j$ be the last column of $Q_j$, so that $Q_j = (Q_{j-1}, q_j)$. For any vector $x$ define:

- $r_j = x - Q_{j-1} Q_{j-1}^T x$
- $u_j = |Y r_j|^2, \quad v_j = |r_j|^2, \quad f_j = \frac{u_j}{v_j}$

If $u_j$ and $v_j$ are known then $f_{j+1}$ can be computed by steps 1-5 described below, where Step 1 is independent of $x$, and steps 2-5 are independent of $N$.

1. $c_j = Y Y^T q_j, \quad d_j = c_j - Q_{j-1} Q_{j-1}^T c_j, \quad \beta_j = q_j^T c_j$
2. $\alpha_j = q_j^T x, \quad \gamma_j = d_j^T x$
3. $u_{j+1} = u_j + \alpha_j^2 \beta_j - 2 \alpha_j \gamma_j$
4. $v_{j+1} = v_j - \alpha_j^2$
5. $f_{j+1} = \frac{u_{j+1}}{v_{j+1}}$
Proof. The value of \( r_j \) as defined in (7) satisfies:

\[
 r_{j+1} = x - Q_j Q_j^T x = r_j - q_j q_j^T x
\]

This gives the following fundamental recursion:

\[
 r_{j+1} = r_j - \alpha_j q_j
\]  

(9)

Observe that \( q_j \) and \( r_{j+1} \) are orthogonal. Writing (9) as

\[
 r_{j+1} + \alpha_j q_j = r_j
\]

and taking squared norms of both sides gives: \( |r_{j+1}|^2 + \alpha_j^2 = |r_j|^2 \). This proves the recursion on line 4. above. This recursion is well known. It was previously used, for example, in \([7, 31]\).

We proceed to derive the recursion for \( u_j \), given by the formula in line 3., that to the best of our knowledge is new. From the fundamental recursion (9):

\[
 Y^T r_{j+1} = Y^T r_j - \alpha_j Y^T q_j
\]

Taking norms:

\[
 u_{j+1} = |Y^T r_{j+1}|^2 = |Y^T r_j - \alpha_j Y^T q_j|^2
\]

\[
 = u_j + \alpha_j^2 \beta_j - 2\alpha_j c_j^T r_j
\]

It remains to show that \( c_j^T r_j = d_j^T x \), and this follows by direct observation. □

3.1 Complexity

Step 1. The computation of \( c_j \) as specified in Step 1 in the theorem can be done in one pass as follows:

\[
 c_j = \sum_{i=1}^{N} y_i (y_i^T q_j)
\]

The cost is roughly \( 4mN \) flops. Similarly, the cost of calculating \( d_j \) is roughly \( 5(j-1)m \), and the cost of calculating \( \beta_j \) is \( 2m \). We observe that summing it up for \( j = 1, \ldots, k \) gives roughly \( 4km(N+1) \) flops.

Step 2. The cost of the two dot products is roughly \( 4m \) flops.

Steps 3,4,5. These steps take roughly 8 flops.

4 The ISOLS-exact algorithm

In this section we describe the ISOLS-exact, a straightforward implementation of the theorem formulas. In order to apply the recursion in the theorem we need the values of \( u_0, v_0 \) for all columns of \( X \). The ISOLS-exact computes these values, then selects the first column, and then applies the recursion to select the rest of the columns. The detailed algorithm is shown in Fig.4 The values needed for selecting the first column of \( X \) are calculated in Step 0.1, and the initial best selection is identified in Step 0.2. The remaining \( k-1 \) selections are determined by the \( k-1 \) iterations of Step 1.

4.1 Complexity

There are three distinct components that affect the run time: the global initialization in Step 0.1, the initial calculation in each iteration, performed in Step 1.0, and the two dot products in Step 1.1.

Dense dictionary. The expensive computation in Step 0.1 is the calculation of \( u_0^i \) that takes roughly \( 2mnN \) flops. The cost of Step 1.0 is dominated by the evaluation of \( c_j \). Taking into account the evaluation of both \( c_j \) and \( d_j \) gives roughly \( 4(k-1)m(N+1) \) flops for the run time of Step 0.1. The cost of Step 1.1 is dominated by the calculation of \( \alpha_j, \gamma_j \) which is roughly \( 4mn \) flops. This gives an asymptotic run time of \( O(mnN) \), dominated by the global initialization.

Sparse dictionary. The number of flops needed to perform a dot product between a non-sparse and a sparse vector is the number of nonzero entries in the sparse vector. Therefore, when \( X \) is sparse
an approximate algorithm where \( Y \) of the ISOLS-exact, and that in both cases the computation depends only on \( YY^T \). The analysis in Section 4 shows that the asymptotic run time of the ISOLS-exact is proportional to \( kmN \) since \( km \) may be larger than \( z_xN \). The asymptotic run time including both Step 0.1 and Step 1.0 is \( O(z_xN + kmN) \).

**Memory.** The algorithm requires a temporary storage of \( km \) for the matrix \( Q \), and a temporary storage of \( 2n \) for keeping the values \( u_i, v_i \) for each column of the dictionary \( X \).

## 5 The ISOLS-lowrank algorithm

The analysis in Section 4 shows that the asymptotic run time of the ISOLS-exact is proportional to \( N \), the number of columns of the data matrix \( Y \). Observe that \( Y \) appears only in steps 0.1 and 1.0 of the ISOLS-exact, and that in both cases the computation depends only on \( YY^T \). This suggests an approximate algorithm where \( Y \) is replaced by the matrix \( H \) that has fewer columns, but can still approximate \( Y \) in the sense that \( H H^T x \approx YY^T x \) for an arbitrary vector \( x \). More specifically we need \( H H^T \) to approximate \( YY^T \) in the spectral norm. The resulting algorithm is described below as a modification of the ISOLS-exact. The changes needed in Algorithm ISOLS-exact are specified blow.

```

0. Initial steps
0.1 for i = 1, ..., n:
   \( u_i^0 = |Y^T x_i|^2, \quad v_i^0 = |x_i|^2 \), \( f_i^0 = u_i^0/v_i^0 \)
0.2 \( i^* = \arg \max_i f_i^0, \quad x_{s_1} = x_{i^*} \).
0.3 S = \{x_{s_1}\}, \quad q_1 = x_{s_1}/|x_{s_1}|, \quad Q_0 = ( ), \quad Q_1 = (q_1).

1. Run iterations for \( j = 1, \ldots, k - 1 \):
1.0 \( c_j = YY^T q_j, \quad d_j = c_j - Q_{j-1} Q_{j-1}^T c_j, \quad \beta_j = q_j^T c_j \).

1.1 for i = 1, ..., n:
   \( \alpha_j^i = q_j^T x_i, \quad \gamma_j^i = d_j^T x_i \),
   \( u_{j+1}^i = u_j^i + (\alpha_j^i)^2 \gamma_j^i - 2 \alpha_j^i \gamma_j^i, \quad v_{j+1}^i = v_j^i - (\alpha_j^i)^2 \gamma_j^i \),
   \( f_{j+1}^i = u_{j+1}^i/v_{j+1}^i \).

1.2 \( i^* = \arg \max_i f_{j+1}^i, \quad x_{s_{j+1}} = x_{i^*} \).

1.3 Add \( x_{s_{j+1}} \) to \( S \)
   \( q_{j+1} = \text{Orthonormal}(x_{s_{j+1}}, Q_j) \)
   \( Q_{j+1} = (Q_j, q_{j+1}) \).
```

Figure 4: The ISOLS-exact algorithm

the complexity of Step 0.1 is reduced to \( O(z_xN) \), where \( z_x \) is the number of non-zeros in the matrix \( X \). Similarly, the complexity of Step 1.1 is reduced to \( 4z_x \). But there is no change in the complexity of Step 1.0 since it is independent of \( X \). This shows that the run time of Step 1.1 cannot be ignored since \( km \) may be larger than \( z_x \). The asymptotic run time including both Step 0.1 and Step 1.0 is \( O(z_xN + kmN) \).

```
Input: X, Y, k, d. Output: S.

0.0 Compute an \( m \times d \) matrix \( H \) such that
   \( H H^T \approx YY^T \).
0.1 Compute \( u_0^i \) by: \( u_0^i = |H^T x_i|^2 \)
0.0 Compute \( c_j \) by: \( c_j = H H^T q_j \).
```

Modifications to the ISOLS-exact algorithm that give the ISOLS-lowrank algorithm
5.1 Complexity

There are four distinct parts that affect the run time of the ISOLS-lowrank algorithm: Step 0.0, where the low rank approximation is computed, and steps 0.1,1.0,1.1, that were discussed in Section 4.1. We write $T_{LR}(Y)$ for the run time of computing the low rank approximation of $Y$. It is discussed in detail in Section 5.2.

Dense dictionary. It is clear from the analysis in Section 4.1 that Step 0.1 dominates steps 1.0 and 1.1 with asymptotic complexity of $O(dmn)$. This gives asymptotic complexity of $O(dmn + T_{LR}(Y))$ for running the ISOLS-lowrank with a dense dictionary.

Sparse dictionary. Using same arguments as in Section 4.1 the asymptotic run time with sparse dictionaries is $O(d(km + z_x) + T_{LR}(Y))$.

Memory. The algorithm requires storage of $km$ for the matrix $Q$, and $2n$ for keeping the values $u_i, v_i$ for each column of the dictionary $X$.

5.2 Low rank approximations to the data matrix

The ISOLS-lowrank replaces the $m \times N$ data matrix $Y$ with the $m \times d$ matrix $H$. The matrix $H$ approximates the matrix $Y$ in the sense that $\|HH^T - YY^T\|_2$ is small. It is known that the best low rank approximation can be obtained from the truncated SVD of $Y$. Suppose the SVD of $Y$ is given by: $Y = U\Sigma V^T$, then $YY^T = U\Sigma^2 U^T$, and the best possible $HH^T$ is $\sum_{j=1}^d \sigma_j^2 u_j u_j^T$, where the vectors $u_j$ are the columns of the orthogonal matrix $U$. This is clearly satisfied if the matrix $H$ is chosen to be:

$$H = (h_1, \ldots, h_d), \quad h_j = \sigma_j u_j \quad \text{for } j = 1, \ldots, d$$

This shows that the matrix $H$ can be calculated from the truncated SVD of the matrix $Y$. Classical iterative approaches for calculating the truncated SVD have run time of $O(dmN)$ (e.g., [23]).

Recently proposed algorithms that use randomization can compute the SVD much faster. See, e.g., [13, 14, 15], and the experimental evaluation in [16]. Since our goal is computing the matrix $H$ and not the SVD, we observe that in some of these algorithms it is not necessary to pass through the SVD, as the desired matrix $H$ is computed in early or intermediate stages of the algorithm. We illustrate this with the algorithm of [13].

To apply the algorithm of [13] for computing $H$ we need the squared norms of all the columns of $Y$. This is sometimes known in advance, or can be calculated in one pass at the cost of roughly $2mN$. Given these norms, the algorithm selects $d$ columns at random, where the probability of selecting the column $y_i$ is proportional to $|y_i|^2$. Without loss of generality suppose the selection is $y_1, \ldots, y_d$. The desired matrix $H$ is formed by:

$$H = (h_1, \ldots, h_d), \quad h_j = y_j / |y_j|^2 \quad \text{for } j = 1, \ldots, d$$

To obtain the same accuracy as with the SVD method, one would need a larger value of $d$. Still, it is shown in [13] that with high probability the increase in the value of $d$ is only by a factor of $O(\log N)$.

6 Experimental results

In this section we describe experimental results with the proposed algorithms. Our goal was to evaluate the run time, stability, and accuracy of the algorithms. The relevant parameters for run time are the amount of speedup for sparse dictionaries, and the amount of speedup gained by reducing the rank of the data matrix. By stability we refer to a known problem with algorithms that require orthogonalization of large matrices. The loss of orthogonality among the vectors may lead to the selection of redundant columns not contributing to the reduction of the error (2). The accuracy is related to the performance of the ISOLS-lowrank as compared with that of the ISOLS-exact.
6.1 Some details about the implementation

In our implementation the low rank approximation of the data matrix $Y$ was computed using the technique of [15]. The idea is to compute a low rank orthonormal matrix $Q$ of dimensions $m \times d$ that approximately spans the column space of $Y$. Using $Q$ one can compute the $d \times d$ matrix $W=Q^TYY^TQ$ in one pass over the data. Let $W=SS^T$ be the Cholesky decomposition of $W$, and set $H^T=S^TQY$, where $H$ is $m \times d$. It is easy to show (see [15]) that if $Q$ is a good in the sense of $QQ^TY \approx Y$ then $H$ is good in the sense of $HH^T \approx YY^T$.

Instead of keeping the matrix $Q$ explicitly we use the well known technique of Householder matrices [23] to indirectly represent it. Observe that the matrix $Q$ is used in steps 1.0 and 1.3 of the algorithm. Both steps can be implemented with the Householder representation, so that only the calculations involving the vector $q_j$ need to be performed outside of the Householder representation.

6.2 The experiments

We conducted two experiments with each dataset. Each experiment involved 7 runs, as explained below.

**Experiment 1.** In this experiment we took both the data matrix $Y$ and the dictionary matrix $X$ to be the same as the given dataset. This is the classical column subset selection problem, as discussed in Section 1. We ran this experiment with $k$ in the range of $1 - 100$, and for various values of $d$, the reduced rank of the data matrix. In particular the following values of $d$ were used: $\{1, 5, 10, 20, 50, 75, 100\}$.

**Experiment 2.** The data matrix $Y$ was taken as identical to the dataset, and the value of $d=50$ was used as the rank of $H$ in ISOLS-lowrank. The dictionary matrix $X$ was created by randomly zeroing out entries in the dataset so that the sparsity of the dictionary was increased. In particular, for $p$ in the range of $\{1, 5, 10, 20, 50, 75, 100\}$ we zeroed out entries with probability of $p/100$. Thus, $p$ is the percentage of retained values. Observe that $p = 100$ is a special case of Experiment 1 with $d=50$. But for smaller values of $p$ the data matrix and the dictionary matrix are distinct.

**Reported measurements.** In each experiment we measured the run time and the approximation error as a function of $d$ and $p$. In the case of ISOLS-lowrank the reported run time does not include the time taken to compute the low rank approximation to $Y$. Clearly, the measurements of run time depend on the hardware, and may be influenced by background processes. But we believe that the rate of change that was observed is properly reflecting the run time dependency on the above parameters.

The approximation error was calculated according to Equation (2), and normalized so that the highest possible value (the squared norm of $Y$) is 100. Thus, the reported approximation error values are the percentage of the error as given by Equation (2).

6.2.1 Experiments with the TechTC01 dataset

This dataset is part of the Technion repository of text categorization datasets. It has $m=163$ rows and $N=29,261$ columns. The results of the experiments are summarized in Fig.5.

Running the ISOLS-exact for selecting 100 dictionary columns took 774 seconds. Using ISOLS-lowrank with $d=100$ took 12 seconds, an improvement by a factor of 64. In the asymptotic limit the run time is expected to be linearly related to $d$. The reported results show a reduction in time for smaller $d$, but not linear dependency. (For example, the run time for $d = 100$ is not 10 times the run time for $d = 10$.) This was observed with other datasets as well, and indicates that the computation overhead is quite significant.

The dependency on $p$, the dictionary sparsity, is exactly as expected, with the run time going down as the value of $p$ is decreased.

The table and plots show the error of predicting the entire data matrix from selected dictionary columns. Because of the rapid decrease in the error we describe the results of each experiment with two plots. The first plot (on the left) shows the results for all tested values of $k$, while the second
Figure 5: Experiments with the TechTC01 dataset. The top part shows time dependency on $d$, the degree of $H$ in ISOLS-lowrank, and on $p$, the sparsity percentage. The table and the plots show how the various parameters affect the accuracy. The error is specified as percentage of the initial error.

The experiments with very sparse dictionary show a decrease in the prediction accuracy, but the decrease is much less than what we expected. We find it surprising that a small fraction of the data can produce such accurate prediction of the entire data matrix. These observations are specific to the TechTC01 dataset. They did not show up with other datasets.
6.2.2 Experiments with the Day1 dataset

This dataset is part of the “URL_reputation” collection at the UCI Repository. It has \( m = 20,000 \) rows and \( N = 3,231,957 \) columns. The data is very sparse. It occupies about 40 megabytes of memory, while a non-sparse representation would require about 200 gigabytes. IOLS-exact ran too slow on this dataset and all results were obtained with the IOLS-lowrank.

We ran the same set of experiments with this dataset as those applied to the TechTC01 dataset. The results are very similar. They are summarized in Fig.6. The main difference is that the prediction error did not go down as rapidly as in the TechTC01 case.
6.2.3 Experiments with the gisette dataset

This dataset was used in the NIPS 2003 selection challenge. It has $m=6,000$ rows and $N=5,000$ columns. We ran the same set of experiments as with the previous two datasets, and the results are summarized in Fig. 7. IOLS-exact ran too slow on this dataset and all results were obtained with the IOLS-lowrank. The results are similar to what was observed with the other datasets.

6.2.4 Summary

The experimental results mostly agree with theoretical analysis. The run time is improved with the reduction in $d$ and with the increase in sparsity. We observed that the dependency on $d$ is
monotone but not linear. Another unexpected observation is the level of prediction accuracy that can be achieved when the ISOLS-lowrank uses small values of $d$.

We did not observe any stability issues. The prediction error kept improving with additionally selected features, indicating that no redundant features were selected.

7 Concluding remarks

The problem that was discussed in this paper, approximating one matrix in terms some columns of another matrix, is well known and appears to have many practical applications. The majority of previous studies address the accuracy of the proposed scheme, but not the resources that are needed to apply it to large matrices. We consider the CM algorithm of Civril and Magdon-Ismail, established in [7], to be the current state-of-the-art. It reduces the $O(kmnN)$ run time of the naive approach to $O(mnN)$, and further to $O(knN)$ in the sparse case. Using a low rank approximation to the data matrix the run time of the CM can be further reduced to $O(dkn)$. On the Other hand, the CM does not improve on the memory requirements of the naive approach. Both require $O(mn)$ memory to store a partially orthogonalized dictionary matrix, and the CM requires an additional storage of $nN$.

The algorithms presented in this paper use an entirely different approach than the CM. They have the same asymptotic complexity for the non-sparse case, and a better asymptotic complexity of $O(nN)$ for the sparse case. Using a low rank approximation to the data matrix the run time is further reduces to $O(dn)$. In addition to the run time our algorithms also require a significantly smaller amount of memory, roughly $km + 2n$ floats.

To illustrate the difference consider the specific example of the Day1 dataset of Section 6.2.2. In our experiments it took less than 4 minutes to select 100 columns, and the algorithm used less than 150 megabytes of memory. In this example $k = 100$, $d = 100$, $m=20,000$, $N=3,231,957$, $N=3,231,957$. The naive method would be roughly $mN \approx 610^{10}$ time slower, taking more than 27 thousand years. It would also require roughly 200 gigabytes of memory. Using the CM the run time would be only 100 times slower than in our approach, taking roughly 7 hours, but it would still require around 200 gigabytes of memory.

References


