Three Iteratively Reweighted Least Squares Algorithms for L1-Norm Principal Component Analysis

Young Woong Park [*1] and Diego Klabjan [†2]

[1]Cox School of Business, Southern Methodist University, Dallas, TX, USA
[2]Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, IL, USA

May 1, 2017

Abstract

Principal component analysis (PCA) is often used to reduce the dimension of data by selecting a few orthonormal vectors that explain most of the variance structure of the data. L1 PCA uses the L1 norm to measure error, whereas the conventional PCA uses the L2 norm. For the L1 PCA problem minimizing the fitting error of the reconstructed data, we propose three algorithms based on iteratively reweighted least squares (IRLS). We first develop an exact reweighted algorithm. Next, an approximate version is developed based on eigenpair approximation when the algorithm is near convergent. Finally, the approximate version is extended based on stochastic singular value decomposition. We provide convergence analyses, and compare their performance against benchmark algorithms in the literature. The computational experiment shows that the proposed algorithms consistently perform best and the scalability is improved as we use eigenpair approximation and stochastic singular value decomposition.

Keywords: L1 PCA · Iteratively Reweighted Least Squares

1 Introduction

Principal component analysis (PCA) is a technique to find orthonormal vectors, which are a linear combination of the attributes of the data, that explain the variance structure of the data [13]. Since a few orthonormal vectors usually explain most of the variance, PCA is often used to reduce dimension of the data by keeping only a few of the orthonormal vectors. These orthonormal vectors are called *principal components* (PCs).

For dimensionality reduction, we are given target dimension p, the number of PCs. To measure accuracy, given p principal components, first, the original data is projected into the lower dimension using the PCs. Next, the projected data in the lower dimension is lifted to the original dimension using the PCs. Observe that this procedure causes loss of some information if p is smaller than the dimension of the original attribute space. The reconstruction error is defined by the difference between the projected-and-lifted data and the original data. To select the best p PCs, the following two objective functions are usually used:

[P1] minimization of the reconstruction error,
[P2] maximization of the variance of the projected data.

The traditional measure to capture the errors and variances in P1 and P2 is the L_2 norm. For each observation, we have the vector of the reconstruction error and variance for P1 and P2, respectively. Then, the squared L_2 norm of the vectors are added over all observations to define the total reconstruction error

[*] ywpark@smu.edu
[†] d-klabjan@northwestern.edu

and variance for P1 and P2, respectively. In fact, in terms of a matrix norm, we optimize the Frobenius norm of the reconstruction error and projected data matrices for P1 and P2, respectively. With the L_2 norm as the objective function, P1 and P2 are actually equivalent. Further, P2 can be efficiently solved by singular value decomposition (SVD) of the data matrix or the eigenvalue decomposition (EVD) of the covariance matrix of the data. However, the L_2 norm is usually sensitive to outliers. As an alternative, PCA based on L_1 norm has been used to find more robust PCs.

In fact, due to the robust-to-outliers characteristic of the L_1 norm, the research interest and activities have been increasing recently. In Figure 1, we present search results for keywords "L1 PCA" and "L1 Norm" by Google Scholar over the past 12 years. The number of search results for L1 PCA and L1 Norm are represented by the bars and circles, respectively, where the associated axes are at the left and right. Although the search results could include papers without exclusive coverage of L_1 PCA or L_1 norm, we can observe that the interest in the L1 version of the problems is generally increasing.

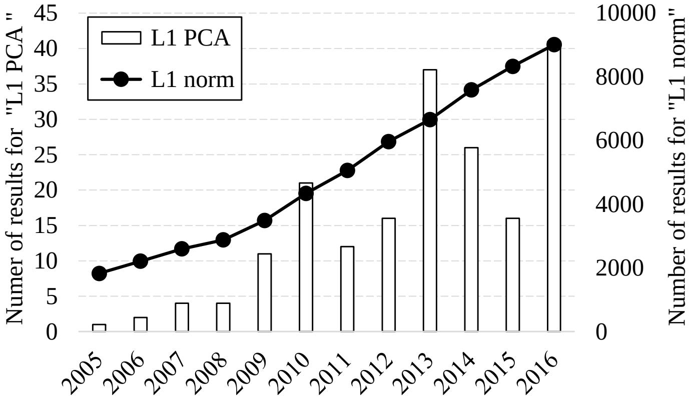


Figure 1: Search results from Google Scholar for L1 PCA and L1 Norm

For P1, instead of the L_2 norm, we minimize the sum of the L_1 norm of the reconstruction error vectors over all observations. A few heuristics have been proposed for this minimization problem. The heuristic proposed in [1] is based on a canonical correlation analysis. The iterative algorithm in [16] assumes that the projected-and-lifted data is a product of two matrices and is then iteratively optimizing by fixing one of the two matrices. The algorithm in [3] sequentially reduces the dimension by one. The algorithm is based on the observation that the projection from k to the best fit $k-1$ dimension can be found by solving several linear program (LP) problems for least absolute deviation regression. The algorithms in [16] and [3] actually try to find the best fitting subspace, where in the objective function the original data is approximated by the multiplication of two matrices, PC and score matrices. This approximation is not the same as the reconstructed matrix by PCs, while the ultimate goal is still minimizing the reconstruction errors.

The L_1 norm for P2 has also been studied. This problem is often called the *projection pursuit* L_1-*PCA*. In this context, we maximize the sum of the L_1 norm of the projected observation vectors over all observations. However, in contrast to the conventional L_2 norm based PCA, the solutions of P1 and P2 with the L_1 norm might not be same. The work in [9] studies L_1-norm based covariance matrix estimation, while the works in [6, 7, 17, 19] and [22] directly consider P2. The algorithm in [17] finds a local optimal solution by sequentially obtaining one PC that is orthogonal to the previously obtained PCs based on a greedy strategy. Recently, the work in [22] extended the algorithm in [17] using a non-greedy strategy. The works in [20] and [21] show that P2 with one PC is NP-hard when the number of observations and attributes are jointly arbitrarily large. The work in [20] provides a polynomial algorithm when the number of attributes is fixed.

As the objective functions are different, solving for P1 and P2 with different norms give different solutions in terms of the signs and order of the PCs. In Figure 2, we present heat maps of PCs obtained by solving L_2-PCA, P1 with the L_1 norm, and P2 with the L_1 norm with $p = 5$ for data set *cancer_2* presented in Table 1 and used in the experiment in Section 3.3. The three heat maps represent the matrices of PCs of L_2-PCA (left), P1 with the L_1 norm (center), and P2 with the L_1 norm (right). The rows and columns of each matrix represent original attributes and PCs, respectively. The blue, white, and red cells represent the intensity of positive, zero, and negative values. Note that both L_1-PCA variations give Attr 1 a large negative loading in either of the first or second PCs, whereas L_2-PCA gives Attr 1 a large positive loading in the third PC. Attr 9 has a large negative loading for the fifth PC of P1 with L_1 norm, while L_2-PCA gives Attr 9 a large positive loading in the second PC.

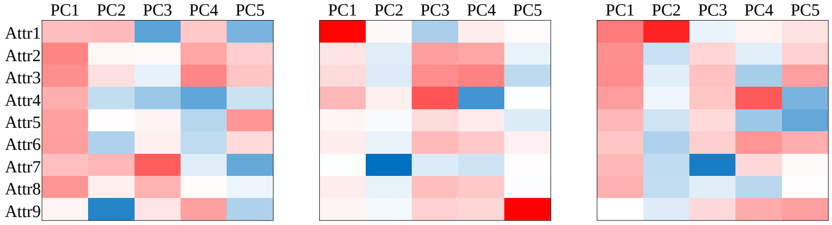


Figure 2: Heat maps of five principal components by L_2-PCA (left), P1 with L_1 norm (center), and P2 with L_1 norm (right) for *cancer_2* data set

In this paper, we propose three iterative algorithms for P1 with the L_1 norm, and provide analytical convergence results. The first two algorithms are proposed in Park and Klabjan [23] and the third algorithm is proposed here to further improve scalability by extending the second algorithm. Although we propose iterative algorithms, our focus is the L_1 objective function and thus our algorithms do not directly compare with iterative algorithms for the standard L_2-PCA such as algorithms based on EM [24] or NIPALS [10]. For P1 with the L_1 norm, the first proposed algorithm, the exact reweighted version, is based on iteratively reweighted least squares (IRLS)[1] that gives a weight to each observation. The weighted least square problem is then converted into the standard L_2-PCA problem with a weighted data matrix, and the algorithm iterates over different weights. We show that the algorithm gives convergent weights and the weighted data matrices have convergent eigenvalues. We also propose an approximate version in order to speed up the algorithm and show the convergent eigenvectors. The third and new algorithm is proposed by extending the approximate version of the algorithm. The extension is based on stochastic SVD (or randomized SVD) [11] and is designed to further speed up the algorithm while accepting increased approximation error. The stochastic SVD is originally proposed by Halko *et el.* [11] and is an approximation algorithm to replace the exact SVD algorithms to speed up the computation.

Note that all of the three proposed algorithms are based on IRLS scheme. The work in [8] provides an IRLS algorithm for sparse recovery. The reader is referred to [14] for a review of IRLS applied in different contexts. Recently, the work in [27] studied the minimization of a differentiable function of an orthogonal matrix. In the computational experiment, we compare our algorithms with benchmark algorithms in [3, 16, 17, 22]. The experiment shows that our algorithm for P1 regularly outperforms the benchmark algorithms with respect to the solution quality and its computational time is of the same order as the fastest among the other four. Even though L_1-PCA can be used for building robust PCs and is an alternative to other robust PCA approaches such as the work in [5], we limit the comparison for the L_1-PCA objective functions introduced in Section 2, as our goal is to directly optimize the L_1 norms for P1.

Our contributions can be summarized as follows.

[1] Iteratively reweighted least squares (IRLS) is an algorithmic framework to solve weighted least squares where the weights depend on the model parameters. Since the weights change based on the model parameters, iterations are needed until some convergence or a termination criteria are met.

1. For P1 with the L_1 norm, we propose the first IRLS algorithm in the literature, and show the convergence of the eigenvalues of the weighted matrices. The algorithm directly minimizes the reconstruction error, while the other benchmark algorithms primarily try to find the optimal subspace. An additional advantage of our algorithm is that it uses an L_2-PCA algorithm as a black box. Hence, by using a more scalable algorithm, the practical time complexity of our algorithm can further be reduced.
2. We propose an approximate version to speed up the algorithm and to guarantee convergent eigenvectors. The difference is that the approximate version uses a formula to update the eigenpairs when the changes in the weighted data matrices become relatively small.
3. We also propose an extension of the approximate algorithm to improve scalability. The difference is the SVD calculation, which is the bottleneck of the algorithm. Instead of the exact SVD calculation, we use stochastic SVD. This enables the algorithm to deal with larger size data as well as an out-of-core situation. The algorithm is especially efficient when the number of PCs is small.
4. The results of the computational experiment show that the proposed algorithms for P1 outperform the benchmark algorithms in most cases. The approximate algorithms based on eigenpair approximation and stochastic SVD are shown to be more scalable while solution qualities remain similar.

The rest of the paper is organized as follows. In Section 2, we present the algorithms and show all analytic results. Then in Section 3, the computational experiment is presented.

2 Algorithms for L1 PCA

In this section, we present the algorithms for P1 and show the underlying analytic results. We use the following notation.

- n: number of observations of the data
- m: number of attributes of the data
- p: number of principal components (target dimension)
- $I = \{1, \cdots, n\}$: index set of the observations
- $J = \{1, \cdots, m\}$: index set of the attributes
- $P = \{1, \cdots, p\}$: index set of the PCs
- $A \in \mathbb{R}^{n\times m}$: data matrix with elements a_{ij} for $i \in I, j \in J$
- $X \in \mathbb{R}^{m\times p}$: principal components matrix with elements x_{jk} for $j \in J, k \in P$
- $Y \in \mathbb{R}^{n\times p}$: projected data matrix with elements y_{ik} for $i \in I, k \in P$, defined as $Y = AX$
- $E \in \mathbb{R}^{n\times m}$: reconstruction error matrix with elements e_{ij} for $i \in I, j \in J$, defined as $E = A - YX^\top$
- $I_k \in \mathbb{R}^{k\times k}$: k by k identity matrix

For a matrix $R \in \mathbb{R}^{n_r\times m_r}$ with elements r_{ij}, $i = 1, \cdots, n_r$, $j = 1, \cdots, m_r$, we denote by

$$\|R\|_F = \sqrt{\textstyle\sum_{i=1}^{n_r} \sum_{j=1}^{m_r} r_{ij}^2}$$

the Frobenius norm.

The conventional PCA problem, P1 with the L_2 norm, can be written as

$$\min_{X\in\mathbb{R}^{m\times p}, X^\top X = I_p} \|A - AXX^\top\|_F^2. \tag{1}$$

Note that $X^\top X = I_p$ is different from $XX^\top \in \mathbb{R}^{m\times m}$ in the objective function.

We consider (1) with the L_1 norm instead of the L_2 norm in the objective function. The resulting P1 problem is written as

$$\min_{X\in\mathbb{R}^{m\times p}} \sum_{i\in I} \sum_{j\in J} |e_{ij}| \text{ s.t. } X^\top X = I_p, E = A - AXX^\top. \tag{2}$$

2.1 IRLS Algorithm

Next we present an iterative algorithm for (2) to minimize the reconstruction error. Instead of solving (2) directly, we iteratively solve a weighted version of (1) by giving a different weight for each observation.

We rewrite (2) in the following non-matrix form.

$$
\begin{aligned}
\min \quad & \sum_{i\in I}\sum_{j\in J}|e_{ij}| && && \text{(3a)}\\
s.t. \quad & y_{ik} = \sum_{j\in J} a_{ij}x_{jk}, && i\in I, k\in P, && \text{(3b)}\\
& e_{ij} = a_{ij} - \sum_{k\in P} y_{ik}x_{jk}, && i\in I, j\in J, && \text{(3c)}\\
& \sum_{j\in J} x_{jk}x_{jq} = 0, && k\in P, q\in P, k<q, && \text{(3d)}\\
& \sum_{j\in J} x_{jk}x_{jk} = 1, && k\in P, && \text{(3e)}\\
& E, X, Y \text{ unconstrained.} && && \text{(3f)}
\end{aligned}
$$

Note that the corresponding L_2-PCA problem (1) can be also written as

$$\min_{E,X,Y} \sum_{i\in I}\sum_{j\in J} e_{ij}^2 \quad \text{s.t.} \quad (3b)-(3f), \tag{4}$$

since the only difference between (1) and (2) is the objective function. However, there is no known algorithm that solves (3) optimally, whereas (4) can be solved by SVD or EVD. Hence, we want to take advantage of the fact that (4) can be optimally solved.

Let us consider a weighted version of (4):

$$g(w) = \min_{E,X,Y} \sum_{i\in I}\sum_{j\in J} w_i e_{ij}^2 \quad \text{s.t.} \quad (3b)-(3f), \tag{5}$$

with $w_i > 0$ for every i in I. Note that (4) and (5) are equivalent when $w_i = 1$ for all i in I. However, solving (5) with non-constant w_i's is not easy in its original form due to the orthogonality constraint. Instead, let us define weighted data matrix $\bar{A} \in \mathbb{R}^{n\times m}$ with each element defined as

$$\bar{a}_{ij} = \sqrt{w_i}a_{ij}, \text{ for } i\in I, j\in J. \tag{6}$$

In the following proposition, we show that an optimal solution to (5) can be obtained by SVD of $\bar{A}$.

Proposition 1. Solving (5) with A is equivalent to solving (4) with $\bar{A}$.

Proof. Let $(\bar{E},\bar{X},\bar{Y})$ be a solution to (4) with $\bar{A}$. We claim that, for any $(\bar{E},\bar{X},\bar{Y})$, there exists (E,X,Y) with the same objective function value for (5) with A, and vice versa. Let $y_{ik} = \frac{1}{\sqrt{w_i}}\bar{y}_{ik}$, $e_{ij} = \frac{1}{\sqrt{w_i}}\bar{e}_{ij}$, and $x_{jk} = \bar{x}_{jk}$. We derive

$$
\begin{aligned}
&y_{ik} = \tfrac{\bar{y}_{ik}}{\sqrt{w_i}} = \tfrac{\sum_{j\in J}\bar{a}_{ij}\bar{x}_{jk}}{\sqrt{w_i}} = \tfrac{\sum_{j\in J}\sqrt{w_i}a_{ij}\bar{x}_{jk}}{\sqrt{w_i}},\\
&e_{ij} = \tfrac{\bar{e}_{ij}}{\sqrt{w_i}} = \tfrac{\bar{a}_{ij}-\sum_{k\in P}\bar{y}_{ik}\bar{x}_{jk}}{\sqrt{w_i}} = a_{ij} - \textstyle\sum_{k\in P} y_{ik}\bar{x}_{jk}\\
&\quad\ = a_{ij} - \textstyle\sum_{k\in P} y_{ik}x_{jk},\\
&\textstyle\sum_{i=1}^n\sum_{j\in J}\bar{e}_{ij}^2 = \sum_{i=1}^n\sum_{j\in J} w_i e_{ij}^2.
\end{aligned}
$$

Further, since $\bar{x}_{jk} = x_{jk}$ for all j and k, orthonormality constraints (3d) and (3e) are automatically satisfied. Hence, solving (4) with $\bar{A}$ is equivalent to solving (5) with A. □

Since now we know that (5) can be solved optimally, the remaining task is to define appropriate weights that give a good solution to (3). We first provide intuition behind our choices.

Let (E^*, X^*, Y^*) be an optimal solution to (3) and let

$$w_i^* = \begin{cases} \frac{\sum_{j\in J}|e_{ij}^*|}{\sum_{j\in J}(e_{ij}^*)^2}, & \text{if } \sum_{j\in J}(e_{ij}^*)^2 > 0, \\ M, & \text{if } \sum_{j\in J}(e_{ij}^*)^2 = 0, \end{cases} \tag{7}$$

be weights defined based on E^*, where M is a large number. Note that the value of M, for the case $\sum_{j\in J}(e_{ij}^*)^2 = 0$, does not affect the value of $\sum_{i\in I} w_i^* \sum_{j\in J}(e_{ij}^*)^2$ because $\sum_{j\in J}(e_{ij}^*)^2 = 0$ for the corresponding observation. However, considering the fact that we want to give less weight to the outliers in order to reduce their effect on the objective function, it is reasonable to assign a big number to the observations with zero error.

With w^* defined in (7), it is trivial to show

$$\sum_{i\in I}\sum_{j\in J}|e_{ij}^*| = \sum_{i\in I} w_i^* \sum_{j\in J}(e_{ij}^*)^2 \text{ and } g(w^*) \le \sum_{i\in I}\sum_{j\in J}|e_{ij}^*|. \tag{8}$$

The equality in (8) implies that, given w^* and E^*, the objective function value of (3) and (5) are equal. The inequality in (8) implies that, given w^*, the objective function value of (5) gives a lower bound on the optimal objective function value of (3). Hence, we aim to minimize the objective function of (3) by solving (5), hoping $g(w^*)$ and $\sum_{i\in I}\sum_{j\in J}|e_{ij}^*|$ are not far from each other.

The equality and inequality in (8) give motivation to use a weight formula similar to (7). Before presenting the weight update formula and the algorithm, let us define the following notation for the algorithm.

- t: current iteration
- $w^t \in \mathbb{R}^{n\times 1}$: weight vector used in iteration t with elements w_i^t for $i \in I$
- $W_t \in \mathbb{R}^{n\times n}$: diagonal matrix in iteration t with $\sqrt{w_i^t}$'s on the diagonal
- $A_t \in \mathbb{R}^{n\times m}$: weighted data matrix defined in (6) with w^t in iteration t, defined as $A_t = W_t A$
- $X_t \in \mathbb{R}^{m\times p}$: the principal component matrix obtained by SVD of A_t in iteration t
- $E_t \in \mathbb{R}^{n\times m}$: reconstruction error matrix in iteration t with elements e_{ij}^t for $i \in I, j \in J$, defined as $E_t = A - AX_tX_t^\top$
- $L2PCA(A_t, p)$: subroutine that returns $X_t \in \mathbb{R}^{m\times p}$ by solving (4) with A_t
- $F(X_t)$: objective function value of X_t for (3), defined as $F(X_t) = \sum_{i\in I}\sum_{j\in J}|e_{ij}^t|$
- F^{best}: current best objective function value
- X^{best}: principal component matrix associated with F^{best}

Note that X_t is obtained by SVD of A_t, but $E_t = A - AX_tX_t^\top$ is based on A and is different from $A_t - A_tX_tX_t^\top$.

Motivated by (7), for iteration $t+1$, we define

$$u_i^{t+1} = \begin{cases} \frac{\sum_{j\in J}|e_{ij}^t|}{\sum_{j\in J}(e_{ij}^t)^2}, & \text{if } \sum_{j\in J}(e_{ij}^t)^2 > 0, \\ M_t, & \text{if } \sum_{j\in J}(e_{ij}^t)^2 = 0, \end{cases} \tag{9}$$

where $M_t = \max_{i\in I_t^+} \frac{\sum_{j\in J}|e_{ij}^t|}{\sum_{j\in J}(e_{ij}^t)^2}$ is the largest weight among the observations in $I_t^+ = \{i \in I | \sum_{j\in J}(e_{ij}^t)^2 > 0\}$.

Note that the value of M_t, for the case $\sum_{j\in J}(e_{ij}^t)^2 = 0$, does not affect the value of $\sum_{i\in I} w_i^t \sum_{j\in J}(e_{ij}^t)^2$ because $\sum_{j\in J}(e_{ij}^t)^2 = 0$ for the corresponding observation. However, considering the fact that we want to give less weight to the outliers in order to reduce their effect to the objective function, it is reasonable to assign a big number to the observations with zero error. Let us consider a few examples to see how u^t in (9) discriminate outliers.

Example. Suppose that we have two observation vectors $e_1 = (3, 3, 3)$ and $e_2 = (5, 1, 1)$. Then both e_1 and e_2 have the same value of squared sum: $\sum_{j=1}^{3} e_{1j}^2 = 3^2+3^2+3^2 = 27$ and $\sum_{j=1}^{3} e_{2j}^2 = 5^2+1^2+1^2 = 27$. However, the absolute sum of e_1 and e_2 are different: $\sum_{j=1}^{3} |e_{1j}| = 3+3+3 = 9$ and $\sum_{j=1}^{3} |e_{2j}| = 5+1+1 = 7$. Hence, we have $u_1 = \frac{9}{27}$ and $u_2 = \frac{7}{27}$. Since $u_1 > u_2$, we prefer e_1, which is less likely to be an outlier since it does not have components with extreme values.

Let us consider another example. Now, we have $e_1 = (10, 2, 2)$ and $e_2 = (5, 1, 1)$. The error distributions for both observations are the same except for the scale. However, the weights are $u_1 = \frac{14}{108}$ and $u_2 = \frac{7}{27}$. Since $u_1 < u_2$, we prefer e_2, which is less likely to be an outlier. □

Using u^{t+1} in (9) for the weights is natural and we empirically observe that the algorithm is convergent. However, it is not trivial to show the convergence with u^{t+1} for (5). Hence, in order to show the convergence of the algorithm, we present a modified update formula based on u^{t+1}.

$$w_i^{t+1} = \begin{cases} w_i^t(1-\beta^t), & \text{if } u_i^{t+1} < w_i^t(1-\beta^t), \\ u_i^{t+1}, & \text{if } w_i^t(1-\beta^t) \le u_i^{t+1} \le w_i^t(1+\beta^t), \\ w_i^t(1+\beta^t), & \text{if } u_i^{t+1} > w_i^t(1+\beta^t), \end{cases} \tag{10}$$

where $\beta \in (0, 1)$. Note that β^t is the β to the power of t and is different from other superscript-containing notations such as w_i^t or u_i^t. The role of (10) is to enable bounds of the change for w^{t+1} from w^t. If u_i^{t+1} is too small compared to w_i^t, then w_i^{t+1} is assigned a value between u_i^{t+1} and w_i^t. If u_i^{t+1} is too large compare to w_i^t, then w_i^{t+1} obtains a value between u_i^{t+1} and w_i^t. Otherwise, w_i^{t+1} follows the weight formula in (9). Given $\beta \in (0, 1)$, we have $\lim_{t\to\infty} \beta^t = 0$, which implies $\lim_{t\to\infty} w_i^t - w_i^{t+1} = 0$. Further, since u and w are bounded above and below, we can show that w^t is convergent. By setting β close to 1, we would have $w_i^{t+1} = u_i^{t+1}$ in most cases, as $1 - \beta^t$ and $1 + \beta^t$ are close to 0 and 2 for small values of t, i.e., early iterations. From all these facts and by using elementary mathematics, the following lemma follows.

Lemma 1. *With w^t defined in (10), w^t and A_t are convergent in t.*

We present the overall algorithmic framework in Algorithm 1. The algorithm requires data matrix A, target dimension p, and tolerance parameter ε as input. After the initialization of weights and the best objective function value F^{best} in Step 1, the while loop is executed until w^t and w^{t+1} are close enough. In each iteration of the while loop, A_t is constructed based on w_t and X_t is obtained by SVD of A_t (Steps 3 and 4). If X_t gives a lower objective function value than X^{best}, then X^{best} and F^{best} are updated. Recall that X_t is obtained by using A_t, but $F(X_t)$ uses the original data matrix A. Each iteration ends after the update of weights in Step 6. Observe that the termination criteria in Step 2 solely depends on the convergence of w_t. Hence, the algorithm terminates in a finite number of iterations. Note that Algorithm 1 uses the standard L_2-PCA algorithm as a black box tool in each iteration. As the time complexity of the standard L_2-PCA algorithm via exact SVD is $O(\min(nm^2, n^2m))$, the time complexity of Algorithm 1 per iteration is $O(\min(nm^2, n^2m))$. Therefore, when the data size increases, Algorithm 1 becomes slower because a single execution of SVD takes a long time.

Lemma 2. *Eigenvalues of $A_t^\top A_t$ are convergent in t.*

Proof. Recall that the weights w_t and weighted matrix A_t are convergent, which also implies $A_t^\top A_t$ is also convergent. Since the eigenvalues of symmetric matrices are point-wise convergent if the matrices are convergent [18], it is trivial to see that the eigenvalues of $A_t^\top A_t$ are convergent. □

Hence, Algorithm 1 gives convergent eigenvalues. Although eigenvalues are convergent, it is not trivial to show the convergence of X_t. This is because even a slight change in a matrix can cause a change in an eigenvector, and eigenvalue-eigenvector pairs are not unique.

Algorithm 1 wPCA (Weight-based algorithm for L_1-PCA)

```
Input: $A$ (data), $p$ (target dimension), $\varepsilon$ (tolerance), $\beta$
Output: principal components $X^{best} \in \mathbb{R}^{m \times p}$
$t \leftarrow 1$, $w_i^0 \leftarrow 2$, $w_i^1 \leftarrow 1$, $F^{best} \leftarrow \infty$
while $\|w^t - w^{t-1}\|_1 > \varepsilon$ do
    set $A_t$ based on $w^t$
    $X_t \leftarrow L2PCA(A_t, p)$
    if $F(X_t) < F^{best}$ then $X^{best} \leftarrow X_t$, $F^{best} \leftarrow F(X_t)$
    update $w_{t+1}$ by using (10) given $\beta$
    $t \leftarrow t + 1$
end while
```

2.2 Approximated IRLS Algorithm

In order to provide convergent eigenpairs and accelerate the algorithm for large scale data, we use the first order eigenpair approximation formula from [26]. Let $(X_{t-1}^i, \lambda_{t-1}^i)$ be an approximate eigenpair of A_{t-1} and $A_t = A_{t-1} + \Delta_t$. Then, the approximate eigenpairs of A_t can be obtained by

$$\lambda_t^i = \lambda_{t-1}^i + (X_{t-1}^i)^\top \Delta_t X_{t-1}^i, \tag{11}$$

$$X_t^i = X_{t-1}^i + \sum_{j \neq i} \Big(\frac{(X_{t-1}^j)^\top \Delta_t X_{t-1}^i}{\lambda_{t-1}^i - \lambda_{t-1}^j} \Big) X_{t-1}^j, \tag{12}$$

using the formula in [26]. The error is of the order of $o(\|\Delta_t\|^2)$. Let *L2PCA_Approx* be a function that returns principal components by formula (11) - (12). The modified algorithm is presented in Algorithm 2.

Algorithm 2 awPCA (Approximated weight-based algorithm for L_1-PCA)

```
Input: $A$ (data), $p$ (target dimension), $\varepsilon$ (tolerance), $\beta$, $\gamma$
Output: principal components $X^{best} \in \mathbb{R}^{m \times p}$
1: $t \leftarrow 1$, $w_i^0 \leftarrow 2$, $w_i^1 \leftarrow 1$, $F^{best} \leftarrow \infty$
2: while $\|w^t - w^{t-1}\|_1 > \varepsilon$ do
3:     set $A_t$ based on $w^t$, $\Delta_t \leftarrow A_t - A_{t-1}$
4:     If $\|\Delta_t\|^2 > \gamma \cdot \|A\|^2$ then $X_t \leftarrow L2PCA(A_t, p)$
5:     Else $(X_t, \lambda_t) \leftarrow L2PCA\_approx(X_{t-1}, \lambda_{t-1}, p)$
6:     if $F(X_t) < F^{best}$ then $X^{best} \leftarrow X_t$, $F^{best} \leftarrow F(X_t)$
7:     update $w_{t+1}$ by using (10) given $\beta$
8:     $t \leftarrow t + 1$
9: end while
```

The difference is only in Lines 4 and 5. If the change in A_t is large (greater than $\gamma \cdot \|A\|^2$), we use the original procedure *L2PCA*. If the change in A_t is small (less than or equal to $\gamma \cdot \|A\|^2$), then we use the update formula (11) and (12). Algorithm 2 has the following convergence result.

Proposition 2. *Eigenpairs of $A_t^\top A_t$ in Algorithm 2 are convergent in t.*

Proof. Note that the convergence of Δ_t in Lemma 2 does not depend on how the eigenvectors are obtained and thus it holds whether we execute Lines 4 or 5 in each iteration. Hence, we have $\lim_{t \to \infty} \Delta_t = 0$ in Algorithm 2.

Since Δ_t converges to zero, after a certain number of iterations $\bar{t}$, we have $\|\Delta_t\|^2 < \gamma \cdot \|A\|^2$ for all $t > \bar{t}$. From such large $\bar{t}$, the approximation rule applies. In [26], it is shown that

$$\begin{aligned}\bar{\lambda}_t^i &= \bar{\lambda}_{t-1}^i + (\bar{X}_{t-1}^i)^\top \Delta_t \bar{X}_{t-1}^i + o(\|\Delta_t\|^2),\\ \bar{X}_t^i &= \bar{X}_{t-1}^i + \textstyle\sum_{j\neq i}\left(\frac{(\bar{X}_{t-1}^j)^\top \Delta_t \bar{X}_{t-1}^i}{\bar{\lambda}_{t-1}^i - \bar{\lambda}_{t-1}^j}\right)\bar{X}_{t-1}^j + o(\|\Delta_t\|^2),\end{aligned}$$

when $(\bar{X}_{t-1}, \bar{\lambda}_{t-1})$ are the exact eigenpairs. Since all of the terms in the formula are bounded and $\lim_{t\to\infty} \Delta_t = 0$, we conclude that the eigenpairs of $A_t^\top A_t$ are convergent. □

Note that Algorithm 2 is identical to Algorithm 1 if $\gamma = 0$. On the other hand, if γ is very large, the expensive SVD operation in Line 4 can be prevented. With reasonable γ values, Algorithm 2 executes Line 4 in the first few iterations and executes Line 5 in the later iterations. The time complexity of Algorithm 2 per iteration is $O(\min(nm^2, n^2m))$.

2.3 Approximated IRLS with Stochastic SVD

Algorithm 2 is an approximate algorithm that reduces the execution time of Algorithm 1 by using an approximation formula when the weighted data matrix A_t is almost converged ($\|\Delta_t\|^2 \leq \gamma \cdot \|A\|^2$). Unfortunately, until this condition is not met, the exact SVD (Line 5 of Algorithm 2) needs to be calculated. However, the run time for calculating the exact SVD is $O(\min(nm^2, n^2m))$ and it may not be practical for large-scale data. Further, both Algorithms 1 and 2 use the SVD algorithm many times as a subroutine. Therefore, in order to improve the computational scalability, we replace the exact computation of Line 4 in Algorithm 2 by stochastic SVD.

Stochastic SVD, sometimes referred as randomized SVD, is proposed by Halko *et al.* [11]. It uses random sampling to decrease the size of the matrix that needs to be decomposed and then the decomposition solution for the original data is recovered using the output of the small data decomposition. It is especially useful when p is small or in out-of-core situations.

We first review the standard stochastic SVD (SSVD) algorithm. Given $n \times m$ matrix A with target dimension p, SSVD computes an approximate rank p singular value decomposition $A \approx U\Sigma V^\top$. One of the standard versions of SSVD accepts two more parameters compared to the typical SVD: an oversampling parameter o and a number of power iterations q. The oversampling helps improving the error bound of SSVD and is used when the rank of the matrix is not trivial to guess. It is also reported that a few power iterations at the end of the algorithm improve the approximation error [11]. In Algorithm 3, we present a version of SSVD.

Algorithm 3 Stochastic SVD (SSVD)

Input: A (data), p (target dimension), o (over sampling), q (power iterations)
Output: U, Σ, V

```
 1: Draw an $m \times (p+o)$ random Gaussian matrix $\Omega$
 2: $Y \leftarrow A\Omega$
 3: $Q \leftarrow$ QR factorization of $Y = QR$
 4: for $i = 1, \cdots q$
 5:     $Y \leftarrow AA^\top Q$
 6:     $Q \leftarrow$ QR factorization of $Y = QR$
 7: end for
 8: $B \leftarrow Q^\top A$
 9: SVD of $B$ to obtain $B = \tilde{U}\Sigma\tilde{V}^\top$
10: $V \leftarrow (\Sigma^{-1}\tilde{U}^\top B)[\ ,1:p]$, $U \leftarrow (Q\tilde{U})[\ ,1:p]$
```

In Line 1, random Gaussian matrix $\Omega \in \mathbb{R}^{m\times(p+o)}$ is drawn. Then, in Line 2, the original matrix is projected based on Ω. In Line 3, the columns of Y are orthogonalized by QR decomposition of Y. In Lines 4-7, if $q > 0$, power iterations are executed to improve accuracy. In Line 8, A is projected to B by $Q^\top$. Then, in Line 8, SVD of smaller matrix B is executed to obtain $\tilde{U}$. Finally, in Line 10, $U \in \mathbb{R}^{n\times p}$ and $V \in \mathbb{R}^{m\times p}$ are obtained as an approximation of SVD to original data A.

The new algorithm for L_1 PCA can be obtained by simply replacing Line 4 of Algorithm 2 with the following line.

$$\textbf{If } \|\Delta_t\|^2 > \gamma \cdot \|A\|^2 \textbf{ then } X_t \leftarrow SSVD(A_t, p, o, q)$$

We will denote this algorithm as awPCAs in the rest of the paper. Note that the time complexity of Algorithm 3 is $O(nm(p+o) + nq(p+o)^2)$, where the bottlenecks are the QR factorization, matrix multiplication, and SVD. Therefore, the time complexity of awPCAs per iteration is $O(nm(p+o)+nq(p+o)^2)$.

3 Computational Experiment

We compare the performance of the proposed and benchmark algorithms for varying instance sizes (m, n) and number of PCs (p). All experiments were performed on a personal computer with 8 GB RAM and Intel Core i7 (2.40GHz dual core). We implement Algorithms 1 and 2, and the stochastic SVD version of the algorithms in R [25], which we denote as wPCA, awPCA, and awPCAs, respectively. The R script for wPCA, awPCA, and awPCAs is available on a web site [1]. Also, the proceeding version of wPCA and awPCA are included in R package pcaL1 [15] since version 1.4.1. For the awPCA implementation, we use condition $\|w^t - w^{t-1}\|_1 > \gamma \cdot \|w^t\|_1$ instead of $\|\Delta_t\|^2 > \gamma \cdot \|A\|^2$, to avoid unnecessary calculation of $\|\Delta_t\|^2$ in Line 4 of Algorithm 2. The weight-based condition is similar to the original condition as the difference in the weight captures Δ_t. For the experiment, we use parameters $\varepsilon = 0.001$ and $\beta = 0.99$ for wPCA, awPCA, and awPCAs, and $\gamma = 0.1$ for awPCA and awPCAs, where the parameters are tuned based on pilot runs to balance the solution quality and execution time. For awPCAs, we use $o = 0$ oversampling and $q = 3$ power iterations. We also set up maximum number of iterations to 200. We compare our algorithms with the algorithms in [3, 16, 17, 22], and the standard L_2-PCA algorithm, which we denote as L2PCA. The work in [4] provides R implementations of the algorithms in [3, 16, 17], which we denote as Brooks, Ke, Kwak, respectively. We implement the algorithm in [22] in R, which we denote as Nie. For L2PCA, R function svd is used.

Although algorithms Ke and Brooks are for (2) and Kwak and Nie are for the L_1 norm version of P2, we evaluate the objective function value for all benchmark algorithms and compare them against our algorithms. Especially, Kwak and Nie, which solve different L_1-PCA problem, are included because

- we observed that Kwak and Nie are better than Ke and Brooks for (2) for some instances, and
- we found that Kwak and Nie are more scalable and solve larger data sets in a reasonable time in the experiment.

Therefore, we include Kwak and Nie for the comparison for solving P1.

It is worth to note that Ke and Brooks try to find the best fitting subspace, where definition of E in (2) is replaced by $E = A - UX^\top$ and $U \in R^{n \times p}$. The optimal solutions of the two formulations may be different despite both minimizing the L_1 distance from the original data.

Let F_{algo} represent the objective function value obtained by $algo \in \{$Ke, Brooks, Kwak, Nie, wPCA, awPCA, L2PCA$\}$, with respect to (2). For the comparison purposes for awPCA, we use the gap from the best objective function value defined as

$$\Delta_{algo} = \min\left\{\frac{F_{algo}}{\min\{F_{\mathsf{awPCA}}, F_{\mathsf{Ke}}, F_{\mathsf{Brooks}}, F_{\mathsf{Kwak}}, F_{\mathsf{Nie}}, F_{\mathsf{L2PCA}}\}} - 1, 1\right\},$$

for each $algo \in \{$awPCA, Ke, Brooks, Kwak, Nie, L2PCA$\}$. Similarly, for wPCA, we define

$$\Delta_{algo} = \min\left\{\frac{F_{algo}}{\min\{F_{\mathsf{wPCA}}, F_{\mathsf{Ke}}, F_{\mathsf{Brooks}}, F_{\mathsf{Kwak}}, F_{\mathsf{Nie}}, F_{\mathsf{L2PCA}}\}} - 1, 1\right\},$$

[1] http://dynresmanagement.com/uploads/3/3/2/9/3329212/wl1pca.zip

for each $algo \in \{\mathsf{wPCA}, \mathsf{Ke}, \mathsf{Brooks}, \mathsf{Kwak}, \mathsf{Nie}, \mathsf{L2PCA}\}$. Note that Δ_{algo} represents the gap between $algo$ and the best of all algorithms. Note also that we set up an upper bound of 1 for Δ_{algo}. Hence, if the gap is larger than 1 (or 100%), then Δ_{algo} is assigned value of 1 (or 100%).

For all of the instances used in the experiment, we first standardize each column and deal exclusively with the standardized data. Hence, the reconstruction errors are also calculated based on the standardized data.

In the computational experiment, we observed that Δ_{awPCA} and Δ_{wPCA} are very similar while the execution time of awPCA is much faster. Hence, in this section, we first focus on presenting the performance of awPCA against the benchmark algorithms and after on comparing the difference between wPCA and awPCA. Also, awPCAs is only compared against awPCA with selected large scale data.

The rest of the section is organized as follows. In Section 3.1, we present synthetic instance generation procedures and explain the instances from the UCI Machine Learning Repository [2]. In Sections 3.2 and 3.3, we present the performance of awPCA for the synthetic and UCI instances, respectively. In Section 3.4, we compare the performance of wPCA and awPCA for the UCI instances. Finally, in Section 3.6, we present performance comparison of awPCA and awPCAs for large UCI data instances.

3.1 Instances

3.1.1 Synthetic Instances

In order to provide a systematic analysis, we generate synthetic instances with presence of outliers and various (m, n, r), where $m \in \{20, 50\}$, $n \in \{100, 300\}$, and $r \in \{0, 0.1, 0.2, 0.3\}$. For each (m, n, r), we generate 5 distinguished instances. Hence, we have a total of 80 generated instances. The synthetic instances used in the experiment are available on a web site [2]. The detailed algorithm is presented at the end of this section. In the generation procedure, r% of observations are generated to have a higher variance than the remaining normal observations. The instance generation algorithm needs additional parameter q (target rank), which we fix to 10 for the instances we generated in this experiment. Hence, the instances we use in the experiment are likely to have rank equal to 10. We consider different p values, where $p \in \{8, 9, 10, 11, 12\}$. Given that $q = 10$, we select p values around 10.

The purpose of the instance generation algorithm is to generate instances with some of the observations as outliers, so that L_1-PCA solutions are more likely to be away from L_2-PCA solutions. In order to generate instances, we use the procedure described in [16] with a slight modification. The instances used in the experiment in [16] have fixed parameters and constant valued outliers to simulate data loss. To check the performance of the algorithms over various parameters and different (non-constant) patterns of outliers, we generate our own instances. In the generation procedure of [16], a matrix with a small fixed rank is generated and then extremely large constant values randomly replace the original data matrix. In their instances, outliers have the same value, which can be interpreted as data loss, but they do not consider outliers due to incorrect measurements or cases with only a few observations with outliers. Our procedure addresses all these cases.

We present the procedure in Algorithm 4.

In Step 1, we first generate random matrix P, where each p_{ij} is from the uniform distribution between -100 and 100. Next in Steps 2 - 10, we generate random perturbation matrix H with approximately r percent of rows having extremely large perturbations, where each row has approximately 10% extreme value entries. After SVD of $PU\Sigma V^\top$ in Step 12, data matrix A is generated, where $U[,1:q]$ is the submatrix of U with the first q columns, $\Sigma[1:q,1:q]$ is the submatrix of Σ with the first q columns and q rows, and $V^\top[,1:q]$ is the submatrix of $V^\top$ with the first q columns. The final data matrix A is generated in Step 14 after adjusting it to have 0 column means.

[2] http://dynresmanagement.com/uploads/3/3/2/9/3329212/pca_instance_park_klabjan.zip

Algorithm 4 PCA instance generation

```
Input: $m$, $n$, $q$ (target rank), $r$ (% outliers)
Output: $A \in \mathbb{R}^{n \times m}$
 1: Generate random matrix $P = [p_{ij}] \in \mathbb{R}^{n \times m}$ with $p_{ij} \sim U(-100, 100)$
 2: for each row
 3:   Generate random number $u_1 \sim U(0, 1)$
 4:   if $u_1 < r$
 5:     for each column $j \leq q$
 6:       Generate $u_2 \sim U(0, 1)$
 7:       if $u_2 < 0.1$, then $h_{ij} \sim N(0, 30)$
 8:       else $h_{ij} \sim N(0, 1)$
 9:     end for
10:   else $h_{ij} \sim N(0, 1)$
11: end for
12: Obtain $P = U\Sigma V^\top$, the SVD of $P$
13: Construct $A = (U[, 1 : q] + H)\Sigma[1 : q, 1 : q]V^\top[, 1 : q]$ with $H = [h_{ij}] \in \mathbb{R}^{n \times q}$ generated in Steps 2 - 10
14: Adjust $A$ to have 0 mean for each column
```

3.1.2 UCI instances

We also consider classification and regression datasets from the UCI Machine Learning Repository [2] and adjust them to create PCA instances. Based on the assumption that observations in the same class of a classification data set have similar attribute distributions, we consider each class of the classification datasets. For each PCA dataset, we partition the observations based on labels (classes). When there exist many labels, we select the top two labels with respect to the number of observations among all labels. For regression data set, we use either the original data or partition the data based on response variable values. For each partitioned data, labels and attributes with zero standard deviation (hence, meaningless) are removed and the matrix is standardized to have zero mean and unit standard deviation for each attribute. One of the regression data set is partitioned based on response variable values.

In Table 1, we list the PCA instances we used and the corresponding original dataset from [2]. In the first column, abbreviate names of the original data sets are presented. The full names of the data sets are Breast Cancer Wisconsin, Indian Liver Patient Dataset, Cardiotocography, Ionosphere, Connectionist Bench (Sonar), Landsat Satellite, Spambase, Magic Gamma Telescope, Page Blocks Classification, Pen-Based Recognition of Handwritten Digits, ISOLET Data Set, MiniBooNE particle identification Data Set, Relative location of CT slices on axial axis Data Set, and YearPredictionMSD Data Set. The *year* UCI data is partitioned into *year20* and *year21*, where *year20* contains observations with response variable (year) less than or equal to 2000 and *year21* contains observations with year > 2000. For *mini* and *year* UCI data, the partitioned PCA instances are merged to create larger size instances.

Each PCA instance is classified as small or large based on m and n. If $mn \leq 15,000$, the instance is classified as small, if $15,000 < mn \leq 1,000,000$, the instance is classified as large, otherwise, the instance is classified as extra large. In the last column in Table 1, the small, large, and extra large instances are indicated by S, L, and XL, respectively. For the large instances, only Kwak and Nie are compared with the proposed algorithms, due to scalability issues of the other benchmark algorithms. For the extra large instances, only awPCA and awPCAs are used to compare the performance of the two algorithm.

3.2 Performance of awPCA for Synthetic Instances

In Table 2, we present the result for awPCA for the synthetic instances. Although we created synthetic instances with varying r (% of outliers) values, we observed that the performances of the algorithms are very similar over different r values for each (m, n, p) triplet. Hence, in Table 2, we present the average

Table 1: PCA instances created based on the datasets from the UCI Machine Learning Repository [2]

Original dataset from UCI			PCA instance		
Name	$(n;m)$	Num labels	Name	$(n;m)$	size
cancer	(699;9)	2	cancer_2	(444;9)	S
			cancer_4	(239;9)	S
ilpd	(583;10)	2	ilpd_1	(416;10)	S
			ilpd_2	(167;10)	S
cardio	(2,126;21)	10	cardio_1	(384;19)	S
			cardio_2	(579;19)	S
iono	(351;34)	2	iono_b	(126;33)	S
			iono_g	(225;32)	S
sonar	(208;60)	1	sonar_g	(111;60)	S
			sonar_r	(97;60)	S
landsat	(4,435;36)	7	landsat_1	(1072;36)	L
			landsat_3	(961;36)	L
spam	(4,601;57)	2	spam_0	(2,788;57)	L
			spam_1	(1,813;57)	L
magic	(19,020;10)	2	magic_g	(12,332;10)	L
			magic_h	(6,688;10)	L
blocks	(5,473;10)	5	blocks_1	(4,913;10)	L
hand	(10,992;16)	10	hand_0	(1,142;16)	L
			hand_1	(1,143;16)	L
isolet	(7,797;617)	5	isolet	(6,238;617)	XL
mini	(130,064;50)	2	mini_1	(36,499;50)	XL
			mini_2	(93,565;50)	XL
			mini	(130,064;50)	XL
ctslice	(53,500;386)	N/A	ctslice	(53,500;384)	XL
year	(515,345;90)	N/A	year_20	(226,230;90)	XL
			year_21	(289,115;90)	XL
			year	(515,345;90)	XL

value over all r. That is, each row of the table is the average of 20 instances for the corresponding (m, n) pair given p. The first two columns are the instance size and number of PCs, the next five columns are Δ_{algo} for all algorithms, and the last five columns are the execution times in seconds. For each row, the lowest Δ_{algo} value among the five algorithms is boldfaced.

Table 2: Performance of awPCA for synthetic instances

Instance		Gap from the best (Δ_{algo})						Time (seconds)					
$(n;m)$	p	awPCA	Ke	Brooks	Kwak	Nie	L2PCA	awPCA	Ke	Brooks	Kwak	Nie	L2PCA
(100;20)	8	**1%**	6%	2%	12%	18%	4%	0.0	0.7	1.3	0.0	0.0	0.0
	9	4%	22%	**3%**	16%	26%	9%	0.0	0.5	1.3	0.0	0.0	0.0
	10	**0%**	**0%**	1%	7%	7%	**0%**	0.0	0.4	1.3	0.0	0.0	0.0
	11	**0%**	69%	2%	7%	12%	**0%**	0.0	0.4	1.3	0.0	0.0	0.0
	12	**0%**	70%	2%	7%	16%	**0%**	0.0	0.4	1.2	0.0	0.0	0.0
(300;20)	8	**2%**	8%	3%	10%	12%	7%	0.0	4.9	9.5	0.0	0.1	0.0
	9	**3%**	10%	**3%**	13%	19%	9%	0.0	2.8	9.6	0.0	0.1	0.0
	10	**0%**	**0%**	1%	3%	3%	**0%**	0.0	1.2	9.6	0.0	0.1	0.0
	11	**0%**	9%	1%	3%	6%	**0%**	0.0	1.2	9.6	0.0	0.1	0.0
	12	**0%**	7%	1%	3%	8%	**0%**	0.0	1.2	9.6	0.0	0.1	0.0
(100;50)	8	**1%**	3%	2%	13%	16%	3%	0.0	3.8	19.2	0.0	0.0	0.0
	9	**1%**	7%	3%	15%	21%	7%	0.0	2.9	19.1	0.0	0.0	0.0
	10	**0%**	**0%**	3%	7%	7%	**0%**	0.0	2.9	19.3	0.0	0.0	0.0
	11	**0%**	100%	2%	7%	8%	**0%**	0.0	4.3	19.2	0.0	0.0	0.0
	12	**0%**	100%	3%	7%	10%	**0%**	0.0	4.8	19.2	0.0	0.0	0.0
(300;50)	8	**1%**	3%	2%	9%	13%	6%	0.1	27.0	227.4	0.0	0.1	0.0
	9	**2%**	5%	3%	10%	16%	8%	0.1	22.0	226.5	0.0	0.2	0.0
	10	**0%**	**0%**	1%	3%	3%	**0%**	0.0	18.7	226.7	0.0	0.2	0.0
	11	**0%**	86%	1%	3%	4%	**0%**	0.0	23.1	228.1	0.0	0.2	0.0
	12	**0%**	98%	1%	3%	5%	**0%**	0.0	27.2	226.0	0.0	0.2	0.0

Note that Δ_{awPCA} values are near zero for all instances. Further, Δ_{awPCA} has the lowest gaps (boldfaced numbers) for all instances among all algorithms except for one instance class. L2PCA also gives zero percent gaps when the number of PCs is greater than or equal to the target rank of the data ($p \geq q$). Since awPCA uses L2PCA in the first iteration, we can conclude that the good performance of awPCA when $p \geq 10$ is closely related to L2PCA. This also implies that, when the number of PCs is greater than the true rank, L2PCA is the best algorithm. However, when $p < q$, Δ_{L2PCA} values are larger than the other benchmark algorithms. Brooks constantly gives the third best gaps while Ke gives 0% gaps for $p = 10$, third best gaps for $p < 10$ and worst gaps for $p > 10$. Nie and Kwak generally give the similar result as they are designed to solve the same problem.

The execution times of the algorithms can be grouped into two groups: awPCA, Kwak, Nie, and L2PCA are in the faster group and Ke and Brooks are in the slower group. Ke and Brooks spend much more time on larger instances compared to the other three algorithms. Although it is not easy to compare, Kwak is the fastest among all algorithms, yet Δ_{Kwak} is not as low as Δ_{awPCA}. It is important to note that the difference in the execution time between Kwak, L2PCA and awPCA is negligible and awPCA is fastest among the algorithms designed to solve P1 with the L_1 norm (i.e., Ke, Brooks, and awPCA). For the faster group, the average execution times across all instances and p values are 0.0286 (awPCA), 0.0133 (Kwak), 0.0836 (Nie), and 0.0019 (L2PCA) seconds. We do not further distinguish or order the execution times of the algorithms because they are too small and the reported time may not represent the actual order.

3.3 Performance of awPCA for UCI Instances

Instance	cancer_2				cancer_4				ilpd_1				ilpd_2				cardio_1					
p	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8	3	6	9	12	15	18
awPCA	0	0	15	61	0	0	3	0	3	0	35	1	1	0	0	4	0	0	0	0	3	0
Ke	15	67	100	100	0	0	5	41	0	8	52	100	0	12	26	90	3	10	20	22	14	100
Brooks	2	22	0	0	0	1	4	4	1	3	0	0	1	6	9	0	4	8	10	6	0	1
Kwak	1	41	38	100	0	1	5	16	3	9	49	45	2	4	12	26	4	7	14	15	13	0
Nie	0	39	69	100	1	1	0	60	12	12	46	56	6	1	28	13	6	10	15	12	19	13
L2PCA	19	76	82	94	0	0	5	14	6	9	45	37	1	13	25	16	6	10	16	11	9	0

Instance	cardio_2						iono_r						iono_g						sonar_g					sonar_r				
p	3	6	9	12	15	18	5	10	15	20	25	30	5	10	15	20	25	30	10	20	30	40	50	10	20	30	40	50
awPCA	0	0	0	0	7	0	1	0	0	0	0	2	0	0	0	0	3	14	0	0	0	0	0	0	0	0	0	0
Ke	0	7	19	43	39	100	0	0	2	2	4	47	0	26	32	51	96	100	0	3	4	19	46	0	1	6	35	67
Brooks	1	7	2	9	0	20	2	3	4	5	7	0	0	1	1	1	0	0	4	6	9	10	10	4	6	9	14	15
Kwak	3	8	8	15	9	1	2	3	4	4	5	8	5	5	5	5	4	15	3	5	5	9	11	3	5	6	10	11
Nie	1	11	13	19	37	6	3	4	5	11	10	13	30	15	13	19	36	98	4	14	17	35	57	7	10	16	33	57
L2PCA	3	13	20	19	11	0	1	1	2	2	2	2	2	7	4	2	3	14	2	2	0	0	0	1	1	0	0	0

(a) Heapt map of gap (%) from the best for small UCI instances

Instance	cancer_2				cancer_4				ilpd_1				ilpd_2				cardio_1					
p	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8	3	6	9	12	15	18
awPCA	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Ke	1	2	3	1	1	1	0	0	3	3	1	1	0	0	1	0	7	7	12	11	3	1
Brooks	2	2	2	2	1	1	1	1	3	3	3	3	1	1	1	1	23	21	19	23	23	21
Kwak	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Nie	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
L2PCA	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Instance	cardio_2						iono_r						iono_g						sonar_g					sonar_r				
p	3	6	9	12	15	18	5	10	15	20	25	30	5	10	15	20	25	30	10	20	30	40	50	10	20	30	40	50
awPCA	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Ke	17	18	11	12	5	1	3	4	4	5	4	3	6	9	12	12	8	2	9	19	27	30	28	8	15	23	27	24
Brooks	39	40	41	37	38	37	10	10	11	10	11	12	33	29	26	29	26	32	45	47	53	51	54	36	35	36	36	36
Kwak	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Nie	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
L2PCA	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(b) Heapt map of the execution times (seconds) for small UCI instances

Instance	landsat_1							landsat_3							spam_0					spam_1				
p	5	10	15	20	25	30	35	5	10	15	20	25	30	35	10	20	30	40	50	10	20	30	40	50
awPCA	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0
kwak	1	1	0	1	1	2	0	1	1	1	1	2	5	0	34	69	100	69	100	24	71	100	100	100
Nie	4	3	3	5	7	13	38	2	2	3	6	6	8	30	38	76	100	83	100	24	73	100	100	100
L2PCA	3	1	0	0	0	0	0	1	0	0	0	1	2	4	28	79	100	95	74	27	91	100	100	100

Instance	magic_g				magic_h				blocks_1				hand_0							hand_1						
p	3	5	7	9	3	5	7	9	3	5	7	9	2	4	6	8	10	12	14	2	4	6	8	10	12	14
awPCA	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
kwak	0	1	1	4	0	4	0	10	5	0	5	17	1	1	1	7	3	1	1	2	2	3	4	3	0	1
Nie	3	3	11	0	3	18	4	3	6	0	7	5	2	5	4	11	13	4	28	3	2	7	9	10	20	14
L2PCA	1	2	3	2	5	8	2	6	17	19	27	19	0	2	4	7	5	1	0	1	3	2	6	4	0	0

(c) Heat map of the gap (%) from the best for large UCI instancess

Instance	landsat_1							landsat_3							spam_0					spam_1				
p	5	10	15	20	25	30	35	5	10	15	20	25	30	35	10	20	30	40	50	10	20	30	40	50
awPCA	2	1	1	1	1	0	0	0	0	1	0	1	1	0	5	6	6	5	5	2	3	3	4	5
kwak	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	2	0	0	1	1	1
Nie	0	1	1	1	1	1	1	0	1	1	1	1	2	2	4	6	7	8	8	1	1	2	5	4
L2PCA	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Instance	magic_g				magic_h				blocks_1				hand_0							hand_1						
p	3	5	7	9	3	5	7	9	3	5	7	9	2	4	6	8	10	12	14	2	4	6	8	10	12	14
awPCA	14	17	18	14	5	15	5	4	3	3	3	2	0	0	1	0	0	0	0	0	0	0	0	0	0	0
kwak	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Nie	6	19	11	16	5	10	8	12	1	4	2	3	0	0	0	1	1	1	0	0	0	0	1	1	1	1
L2PCA	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(d) Heat map of the execution times (seconds) for large UCI instances

Figure 3: Heat maps

For each small and large PCA instance in Table 1, we execute the algorithms with various p values. The number of PCs p covers the entire spectrum $0, \cdots, m$ in increments of 2,3,5, or 10 depending on m: *cancer* and *ilpd* with $p \in \{2, 4, 6, 8\}$, *cardio* with $p \in \{3, 6, 9, 12, 15\}$, *iono* with $p \in \{5, 10, 15, 20, 25, 30\}$, *sonar* and *spam* with $p \in \{10, 20, 30, 40, 50\}$, *landsat* with $p \in \{5, 10, \cdots, 30, 35\}$, *magic* and *block* with

$p \in \{1,3,5,7,9\}$, and *hand* with $p \in \{2,4,\cdots,12,14\}$.

For the small UCI instances, we present heat maps of Δ_{algo} and the execution times of all algorithms in Figures 3(a) and 3(b). In both figures, the numbers are Δ_{algo} in percentage or execution time in seconds, a white cell implies near-zero Δ_{algo} or near-zero execution time, and a dark gray cell implies the opposite. In Figure 3(a), awPCA is consistently best with Brooks usually being the second best algorithm. The value of Δ_{awPCA} is zero except for a few cases. For such cases with $\Delta_{\text{awPCA}} > 0$, Brooks performs the best. The values of Δ_{Ke}, Δ_{Kwak}, and Δ_{Nie} tend to increase in p. L2PCA has small Δ_{L2PCA} values for *iono_r*, *iono_g*, *sonar_g*, and *sonar_r* while it does not perform well for the other data sets. In Figure 3(b), we observe the same trend from Section 3.2: awPCA, Kwak, Nie, and L2PCA are in the faster group and Ke and Brooks become slower as instance size increases. For the faster group, the average execution times across all instances and p values are 0.0698 (awPCA), 0.025 (Kwak), 0.0916 (Nie), and 0.0004 (L2PCA) seconds. We do not further distinguish or order the execution times of the algorithms because they are too small and the reported time may not represent the actual order.

For the large UCI instances, we only compare awPCA against Kwak, Nie, and L2PCA, due to scalability issues of Ke and Brooks. Hence, Δ_{algo} here is the gap from the best of awPCA, Kwak and Nie. In Figures 3(c) and 3(d), we present heat maps of Δ_{algo} and the execution times of the four algorithms. Similar to Figures 3(a) and 3(b), a white cell implies a low value. In Figure 3(c), awPCA is consistently best except for four cases and even for the four cases Δ_{awPCA} are very small. We observe that Δ_{Kwak} and Δ_{Nie} tend to increase in p, where Δ_{Kwak} is slightly smaller than Δ_{Nie} in general. L2PCA performs as good as awPCA for *landsat_1* and *landsat_3*, yet the performance of L2PCA is similar to that of Kwak and Nie in general. In Figure 3(d), the execution times of L2PCA is the fastest, Kwak is the second best, and awPCA and Nie are of the same magnitude, although awPCA is slightly faster than Nie.

Based on the results for the UCI instances, we conclude that awPCA performs the best while the execution time of awPCA is of the same order or lower than the remaining algorithm.

3.4 Comparison of wPCA and awPCA for UCI Instances

In this section, we compare wPCA and awPCA for the UCI instances in terms of solution quality (Δ_{wPCA} and Δ_{awPCA}) and execution time. In Table 3, we present the average performance of wPCA and awPCA for all p values considered in Section 3.3. The fourth column is defined as *diff* $= \Delta_{\text{wPCA}} - \Delta_{\text{awPCA}}$, where a negative *diff* value implies that wPCA gives a better solution and near-zero *diff* value implies that Δ_{wPCA} and Δ_{awPCA} are similar. The seventh column is defined as $\rho =$ execution time of awPCA / execution time of wPCA, where a less-than-one ρ value implies that awPCA is faster than wPCA.

Table 3: Comparison of wPCA and awPCA for UCI instances (small and large)

	Gap from the best (Δ_{algo})			Time (seconds)		
Instance	wPCA	awPCA	diff	wPCA	awPCA	ρ
cancer_2	15.7%	19.0%	**-3.3%**	1.7	0.1	0.1
cancer_4	0.8%	0.8%	0.0%	0.4	0.0	0.1
ilpd_1	1.0%	9.8%	**-8.7%**	0.3	0.1	0.3
ilpd_2	0.9%	1.2%	-0.3%	0.5	0.0	0.0
cardio_1	0.7%	0.6%	0.1%	0.3	0.1	0.3
cardio_2	1.0%	1.2%	-0.2%	1.1	0.1	0.1
iono_r	0.4%	0.4%	0.0%	0.4	0.0	0.1
iono_g	2.8%	2.8%	0.0%	0.2	0.0	0.2
sonar_g	0.1%	0.1%	0.0%	1.0	0.1	0.1
sonar_r	0.0%	0.0%	0.0%	0.9	0.1	0.1
landsat_1	0.0%	0.0%	0.0%	3.2	0.6	0.2
landsat_3	0.6%	0.6%	0.0%	5.7	0.4	0.1
spam_0	0.0%	0.0%	0.0%	16.7	5.2	0.3
spam_1	0.0%	0.0%	0.0%	18.9	3.2	0.2
magic_g	0.1%	0.2%	-0.1%	47.7	15.6	0.3
magic_h	0.6%	0.5%	0.1%	13.8	7.1	0.5
blocks_1	0.0%	0.0%	0.0%	1.8	0.3	0.2
hand_0	0.0%	0.0%	0.0%	1.8	0.3	0.2
hand_1	0.0%	0.0%	0.0%	2.2	0.3	0.1
average			-0.7%			0.2

In Table 3, we observe that Δ_{wPCA} and Δ_{awPCA} are very similar except for two instances (boldfaced values), while awPCA spends only 20% of the time of wPCA on average. Note also that awPCA is not always inferior to wPCA. Although it is rare, awPCA gives a better solution than wPCA for instances *magic_h* and *cardio_*1. In general, we found that Δ_{awPCA} is very similar or slightly larger than Δ_{wPCA}, while awPCA is much faster. On the other hand, we can also ignore the time difference if execution times are within a few seconds. The UCI instances *spam_*0, *spam_*1, *magic_g*, and *magic_h* with clear time difference between wPCA and awPCA have $mn > 50000$. Therefore, we recommend to use wPCA when the data size small and awPCA when the data size is very large.

3.5 Sensitivity Analysis of awPCA

In this section, we present a sensitivity analysis for algorithm parameters β and γ of awPCA. Recall that β determines how to control weight changes and γ determines when to execute eigenpair approximation or exact SVD. The default values $\beta = 0.99$ and $\gamma = 0.1$ were used in all aforementioned experiments. We did not do a full combinatorial design of experiments, instead we tested various β values with a fixed $\gamma = 0.01$ and various γ values with a fixed $\beta = 0.99$. The small UCI instances are used for this experiment and all settings from the previous section are used except for β and γ. In Table 4, the aggregated results by β and γ are presented. The first four columns are obtained by altering β and the last four columns are obtained by modifying γ. The number of iterations (Iteration), execution time (Time), and the improvement from L2PCA (Δ_{awPCA}) are presented. The boldfaced β and γ values indicate the default values for the algorithm.

Table 4: Effect of changing parameters β and γ (UCI small instances)

Changing β with fixed $\gamma = 0.01$				Changing γ with fixed $\beta = 0.99$			
β	Iteration	Time	Δ_{awPCA}	γ	Iteration	Time	Δ_{awPCA}
0.7	4.1	27.4	3.8%	0.01	12.4	85.4	4.2%
0.75	4.6	28.2	4.0%	0.05	11.7	77.8	4.2%
0.8	4.7	29.4	3.9%	**0.1**	9.8	69.2	4.2%
0.85	5.5	34.4	3.8%	0.2	8.6	70.6	4.3%
0.9	6.5	44.6	3.9%	0.3	7.6	49.0	4.3%
0.95	9.4	72.8	4.0%	0.4	7.2	48.6	4.6%
0.99	9.8	68.2	4.2%	0.5	6.2	47.2	4.4%

With increasing β, we observe that the number of iterations and execution time increase. This implies that, when the bounds on weight change are large (large β), the algorithm takes more iterations and time. On the other hand, if β is small, the weight update is very restrictive and the algorithm terminates early. The improvements from L2PCA (Δ_{awPCA}) are similar with changing β, and no clear pattern is observed.

With increasing γ, we observe that the number of iterations and execution time increase. This implies that when the algorithm executes the eigenpair approximation earlier (γ is large and the weights are father from convergence), the algorithm can save time compared to wPCA. The Δ_{awPCA} values indicate that setting $\gamma > 0.1$ gives slightly worse objective function values.

In conclusion, decreasing β and increasing γ can further reduce the execution time of the algorithm. However, it could interfere with the weight updating formula and make the algorithm to use eigenpair approximation too earlier. The default value $\beta = 0.99$ is selected as to have least restrictive bounds for weight update and set default value $\gamma = 0.01$ is selected because it is the largest γ without sacrificing the execution time.

3.6 Comparison of awPCA and awPCAs for UCI Instances

In this section, we compare the solution time and quality of awPCA and awPCAs. Because we are comparing only two algorithms, solution quality measure Δ_{awPCAs} is now defined as

$$\Delta_{\text{awPCAs}} = \frac{F_{\text{awPCA}} - F_{\text{awPCAs}}}{F_{\text{awPCAs}}}.$$

If $\Delta_{\mathsf{awPCAs}} < 0$, then awPCAs gives a worse (larger) objective function value, otherwise, awPCAs gives better objective function value than awPCA. In addition, the solution time ratio $\rho =$ (execution time of awPCA)/(execution time of awPCAs) is reported. Note that comparing the overall time might not give a good idea on which algorithm is faster because the number of iterations is different. Hence, we also check $\rho_{\mathsf{tpi}} =$ (time per iteration of awPCA)/(time per iteration of awPCAs), where *tpi* is the time per iteration defined as the total solution time divided by the number of iterations. Similar to ρ, we can conclude that awPCAs has faster execution time per iteration if $\rho_{\mathsf{tpi}} < 1$.

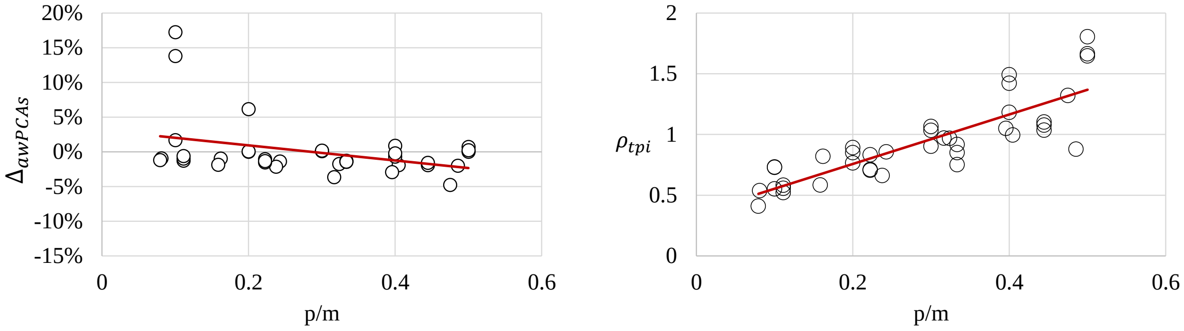
In Table 5, we present the result for the extra large UCI instances. For each extra large PCA instance in Table 1, we execute the algorithms with various p values. The number of PCs p covers the half of the entire spectrum, i.e., $0, \cdots, \lfloor \frac{m}{2} \rfloor$, in increments of 5,10,30, or 50 depending on m: $isolet \in \{50, 100, 150, 200, 250, 300\}$, $mini1$, $mini2$, and $mini \in \{5, 10, 15, 20, 25\}$, $ctslice \in \{30, 60, 90, 120, 150, 180\}$, $year20$, $year21$, and $year \in \{10, 20, 30, 40\}$. Value p is bounded by $\lfloor \frac{m}{2} \rfloor$ because stochastic SVD is designed for smaller p and known to be less efficient when p is large. When $p > \lfloor \frac{m}{2} \rfloor$, we can assume that SSVD is less efficient.

Table 5: Comparison of awPCA and awPCAs for UCI instances (extra large)

				awPCA			awPCAs			Comparison		
Instance	n	m	p	time	iter	tpi	time	iter	tpi	ρ	ρ_{tpi}	Δ_{awPCAs}
isolet	6,238	618	50	41.0	6	6.8	18.4	5	3.7	**0.45**	**0.54**	-1.0%
			100	59.4	5	11.9	48.7	5	9.7	**0.82**	**0.82**	-1.1%
			150	211.9	9	23.5	100.9	5	20.2	**0.48**	**0.86**	-1.3%
			200	180.1	6	30.0	174.4	6	29.1	**0.97**	**0.97**	-1.7%
			250	263.8	6	44.0	262.8	6	43.8	1.00	1.00	-2.3%
			300	540.0	7	77.1	1152.0	17	67.8	2.13	**0.88**	-1.8%
mini1	36,499	50	5	7.1	8	0.9	3.4	7	0.5	**0.48**	**0.55**	1.8%
			10	5.6	8	0.7	4.3	8	0.5	**0.77**	**0.77**	0.0%
			15	5.1	7	0.7	4.6	7	0.7	**0.90**	**0.90**	-2.9%
			20	1.8	3	0.6	5.1	7	0.7	2.76	1.18	0.4%
			25	1.8	3	0.6	26.4	24	1.1	14.43	1.80	3.4%
mini2	93,565	50	5	11.2	7	1.6	8.2	7	1.2	**0.73**	**0.73**	15.5%
			10	10.4	7	1.5	9.3	7	1.3	**0.89**	**0.89**	4.8%
			15	4.7	3	1.6	9.8	6	1.6	2.07	1.03	0.1%
			20	4.9	3	1.6	16.2	7	2.3	3.32	1.42	-0.2%
			25	5.0	3	1.7	19.1	7	2.7	3.84	1.65	0.0%
mini	130,064	50	5	15.4	7	2.2	11.3	7	1.6	**0.73**	**0.73**	13.3%
			10	24.7	11	2.2	42.0	22	1.9	1.70	**0.85**	0.1%
			15	6.0	3	2.0	12.7	6	2.1	2.13	1.07	0.2%
			20	6.1	3	2.0	33.2	11	3.0	5.47	1.49	-0.7%
			25	6.3	3	2.1	24.3	7	3.5	3.88	1.66	-0.1%
ctslice	53,500	379	30	87.8	5	17.6	35.9	5	7.2	**0.41**	**0.41**	-1.1%
			60	96.5	5	19.3	56.3	5	11.3	**0.58**	**0.58**	-1.5%
			90	142.7	6	23.8	94.5	6	15.7	**0.66**	**0.66**	-2.2%
			120	315.0	12	26.3	331.0	13	25.5	1.05	**0.97**	-1.9%
			150	1034.9	32	32.3	1154.6	34	34.0	1.12	1.05	-2.4%
			180	1024.4	30	34.1	2663.0	59	45.1	2.60	1.32	-2.3%
year20	226,230	90	10	36.4	5	7.3	20.2	5	4.0	**0.56**	**0.56**	-0.9%
			20	38.1	5	7.6	26.8	5	5.4	**0.71**	**0.71**	-1.2%
			30	53.9	6	9.0	45.8	6	7.6	**0.85**	**0.85**	-1.4%
			40	56.4	6	9.4	62.1	6	10.4	1.10	1.10	-1.8%
year21	289,115	90	10	53.1	5	10.6	27.7	5	5.5	**0.52**	**0.52**	-0.8%
			20	57.2	5	11.4	48.8	6	8.1	**0.85**	**0.71**	-1.2%
			30	87.1	6	14.5	65.4	6	10.9	**0.75**	**0.75**	-1.2%
			40	80.1	6	13.3	82.9	6	13.8	1.03	1.03	-1.3%
year	515,345	90	10	95.2	5	19.0	55.4	5	11.1	**0.58**	**0.58**	-0.6%
			20	106.1	5	21.2	106.1	6	17.7	1.00	**0.83**	-1.2%
			30	158.7	6	26.4	145.5	6	24.2	**0.92**	**0.92**	-1.2%
			40	189.6	6	31.6	204.3	6	34.1	1.08	1.08	-1.8%
									average	1.70	**0.93**	0.0%

In Table 5, for columns of ρ and ρ_{tpi}, the numbers are in bold face if the value is less than one, which indicates awPCAs is faster. By comparing ρ values, we observe that awPCAs is not always fast. In fact, the overall average of ρ is 1.70 which indicates that awPCAs is 70% slower. However, for the smallest p value for each instance (for example, $p = 50$ for *isolet*), awPCAs is approximately two times faster than awPCA. As p increases, ρ values also tend to increase. Similar to the previous comparison for ρ, the values of ρ_{tpi} increase in p and there are slightly more cases such that the time per iteration of awPCAs is faster. In terms of solution quality, awPCA and awPCAs give very similar results. Except for three cases, $|\Delta_{\text{awPCAs}}|$ is less than 3%. For the three cases with $|\Delta_{\text{awPCAs}}| > 3\%$, awPCAs actually improves the solution of awPCA by 11.2% on overage. Overall, awPCA gives a slightly better solution in most cases. However, the overall average of Δ_{awPCAs} is zero percent because there are cases such that awPCAs gives significantly improved solutions.

To check what drives the changes of ρ_{tpi} and Δ_{awPCAs}, we present scatter plots in Figure 4. In both plots, the horizontal axis is $\frac{p}{m}$ ranging from 0 to 0.5. The vertical axes are Δ_{awPCAs} and ρ_{tpi} in Figures 4(a) and 4(b), respectively. In each of the plots, the trend line is also displayed. In Figure 4(a), we observe that the slope of the trend line is near zero especially if we exclude the three points with awPCAs performing better. In Figure 4(b), we observe a straightforward trend indicating ρ_{tpi} increases in increasing $\frac{p}{m}$. awPCAs tends to be faster only when $\frac{p}{m} \leq 0.3$ for the UCI instances used in the experiment.



(a) Δ_{awPCAs}

(b) ρ_{tpi}

Figure 4: Scatter plots for Δ_{awPCAs} and ρ_{tpi} over $\frac{p}{m}$

4 Conclusions

In this paper, we consider the L_1-PCA problem minimizing the L1 reconstruction errors and present iterative algorithms, wPCA, awPCA, and awPCAs, where awPCA is an approximation version of wPCA developed to avoid computationally expensive operations of SVD, and awPCAs is an extended version of awPCA for large data sets and small number of PCs p. The core of the algorithms relies on an iteratively reweighted least squares scheme and the expressions in (8). Although the optimality of L_1-PCA was not able to be shown and remains unknown, we show that the eigenvalues of wPCA and awPCA converge and that eigenvectors of awPCA converge. In the computational experiment, we observe that awPCA outperforms all of the benchmark algorithms while the execution times are competitive. Out of the four algorithms designed to minimize the L1 reconstruction errors (Ke, Brooks, wPCA, awPCA), we observe that awPCA is the fastest algorithm with near-best solution qualities. By proposing awPCAs, we improve the scalability of the algorithm while the solution quality is very similar to the original algorithm.

References

[1] A. Baccini, P. Besse, and A. de Faguerolles. A L1-norm PCA and a heuristic approach. In *Proceedings of the International Conference on Ordinal and Symbolic Data Analysis*, pages 359–368, March 1996.

[2] K. Bache and M. Lichman. UCI machine learning repository, 2013.

[3] J. Brooks, J. Dula, and E. Boone. A pure L1-norm principal component analysis. *Computational Statistics and Data Analysis*, 61:83–98, May 2013.

[4] J. Brooks and S. Jot. pcaL1: An implementation in R of three methods for L1-norm principal component analysis. *Optimization Online*, 2012.

[5] E. J. Candés, X. Li, Y. Ma, and J. Wright. Robust Principal Component Analysis? *Journal of the ACM*, 58(3), 2011.

[6] V. Choulakian. L1-norm projection pursuit principal component analysis. *Computational Statistics and Data Analysis*, 50(6):1441–1451, March 2006.

[7] C. Croux and A. Ruiz-Gazen. High breakdown estimators for principal components: the projection-pursuit approach revisited. *Journal of Multivariate Analysis*, 95(1):206 – 226, 2005.

[8] I. Daubechies, R. DeVore, M. Fornasier, and C. S. Gntrk. Iteratively reweighted least squares minimization for sparse recovery. *Communications on Pure and Applied Mathematics*, 63(1):1–38, 2010.

[9] J. Galpin and D. Hawkins. Methods of L1 estimation of a covariance matrix. *Computational Statistics and Data Analysis*, 5(4):305–319, September 1987.

[10] P. Geladi and B. R. Kowalski. Partial least-squares regression: a tutorial. *Analytica Chimica Acta*, 185:1–17, 1986.

[11] N. Halko, P. G. Martinsson and J. A. Tropp. Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions. *SIAM Review*, 53(2):217–288, 2011.

[12] R. A. Horn and C. R. Johnson. *Matrix Analysis.* Cambridge University Press, Cambridge, 2013.

[13] I. Jolliffe. *Principal Component Analysis.* Springer, 2002.

[14] M. Jorgensen. *Iteratively Reweighted Least Squares.* John Wiley & Sons, Ltd, 2006.

[15] S. Jot, P. Brooks, A. Visentin, and Y.W. Park (2016). *pcaL1: L1-Norm PCA Methods.* R package version 1.4.1.

[16] Q. Ke and T. Kanade. Robust L1 norm factorization in the presence of outliers and missing data by alternative convex programming. In *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 739–746, June 2005.

[17] N. Kwak. Principal component analysis based on L1-norm maximization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(9):1672–1680, September 2008.

[18] P. D. Lax. *Linear algebra and its applications.* John Wiley & Sons, 2007.

[19] G. Li and Z. Chen. Projection-pursuit approach to robust dispersion matrices and principal components: primary theory and monte carlo. *Journal of the American Statistical Association*, 80(391):759–766, 1985.

[20] P. P. Markopoulos, G. N. Karystinos, and D. A. Pados. Optimal algorithms for L1-subspace signal processing. *IEEE Transactions on Signal Processing*, 62(19):5046–5058, October 2014.

[21] M. McCoy and J. A. Tropp. Two proposals for robust PCA using semidefinite programming. *Electronic Journal of Statistics*, 5:1123–1160, 2011.

[22] F. Nie, H. Huang, C. Ding, D. Luo, and H. Wang. Robust principal component analysis with non-greedy L1-norm maximization. In *Proceeding of 22nd International Conference on Artificial Intelligence*, pages 1433–1438, June 2011.

[23] Y.W. Park and D. Klabjan. Iteratively Reweighted Least Squares Algorithms for L1-Norm Principal Component Analysis. *2016 IEEE International Conference on Data Mining, Barcelona*, 2016.

[24] S. Roweis. EM Algorithms for PCA and SPCA. In *Advances in Neural Information Processing Systems*, pages 626–632, 1998.

[25] R Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2014.

[26] Y. Shmueli, G. Wolf, and A. Averbuch. Updating kernel methods in spectral decomposition by affinity perturbations. *Linear Algebra and its Applications*, 437:1356–1365, 2012.

[27] Z. Wen and W. Yin. A feasible method for optimization with orthogonality constraints. *Mathematical Programming*, 142(1-2):397–434, 2013.