Project 2: Reducto – Dimensionality Reduction

Heather Michaud and Drew Guarnera

The goal of this project is to use linear algebra matrix operations to shrink large data sets for easier interpretation and compression. Specifically, we utilized singualr value decomposition (SVD) and binary encoding for image compression, and principal component analysis (PCA) to analyze a complex dataset by reducing its dimensionality.

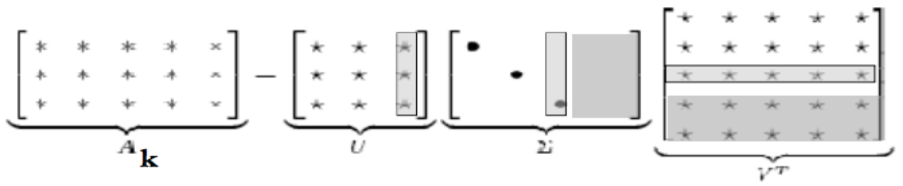
1. SVD & PCA

Singular value decomposition (SVD) and principal component analysis (PCA) are closely related. Both are eigenvalue methods of reducing the dimensionality of high dimensional data while preserving significant information regarding the data. In both cases, an input matrix containing the relevant data is decomposed into three matrices.

A singular value decomposition of an m by n matrix A is a factorization of the form $A = U\Sigma V^T$, where

- U is an m by m unitary matrix whose column vectors are the orthogonal eigenvectors of AA^T,
- Σ is an m by n rectangular diagonal matrix with nonnegative singular values $\sigma_1 \dots \sigma_n$on the diagonal, which are found by taking the square root of the eigenvalues corresponding to the eigenvectors of A^TA such that $\sigma_i = \sqrt{\lambda_i}$, and
- V^T is the transpose of an n by n unitary matrix V whose column vectors are the orthogonal eigenvectors of A^TA.

Intuitively, these three matrices are geometrical transformations: U and V^T can be viewed as rotation matrices, and Σ is a scaling matrix. Because the singular values contained within Σ represent scaling factors for the matrix, we can remove the smaller, insignificant values from the lower portion of the matrix since it contributes so little to the approximation. In this way, we can reduce the dimensionality of the data matrix. A visualization of this dimensionality reduction is shown the following figure, where the approximation matrix of rank 2 is calculated.



Similarly, principal component analysis can reduce the dimensionality of high dimensional data to better present and convey useful information by finding the principal components. The first principal component is the direction of maximum variance from the origin, and the subsequent principal components are orthogonal the first principal component and describe maximum residual variance. Ultimately, the goal is to explain and summarize the underlying variance-covariance structure of a large set of variables through a few linear combinations of these variables.

Given a population measured on p random variables $X_1, \dots, X_p$ that represent the p-axes of the Cartesian coordinate system, we can develop a new set of p axes in the direction of the greatest variability by rotating the axes. From k original variables $x_,, x_2, \dots, x_k$, we can produce k new variables $y_,, y_2, \dots, y_k$, which form a system of k linear equations:

$$y_1 = a_{11}x_1 + a_{12}x_2 + \cdots + a_{1k}x_k$$
$$y_2 = a_{21}x_1 + a_{22}x_2 + \cdots + a_{2k}x_k$$
$$\dots$$
$$y_k = a_{k1}x_1 + a_{k2}x_2 + \cdots + a_{kk}x_k$$

such that y_k's are uncorrelated, hence orthogonal; y_1 explains as much as possible of the original variance in the data set, y_2 explains as much as possible of the remaining variance, and so forth.

PCA is accomplished by obtaining the n by n covariant matrix S of the m by n data set matrix A, such that $S = \frac{1}{n-1}A^TA$. Next, S is decomposed into the form $S = VDV^T$, where

- D is an n by n matrix where the diagonal elements are the eigenvalues of A^TA,
- V is an n by n matrix where the column vectors are the eigenvectors corresponding to the eigenvalues of A^TA.

Common applications of principal component analysis involve the visualization of high dimensional data, such as determining the genetic components responsible for various diseases and illnesses.

SVD and PCA are very closely related. As previously stated, SVD is the process of taking a matrix A and factoring it into the form $A = U\Sigma V^T$. On the other hand, PCA is the process of taking a matrix A and factoring it into the form $A^TA = VDV^T$. We can show how closely related these two forms are by performing SVD on A^TA, so that $A^TA = (U\Sigma V^T)^T(U\Sigma V^T) = (V\Sigma U^T)(U\Sigma V^T) = V\Sigma U^TU\Sigma V^T = V\Sigma^2V^T$. This explains why the D matrix in PCA is the eigenvalues of the corresponding matrix, rather than the singular values – which are the square root of the eigenvalues. In addition, U is no longer required in the calculation because $U^TU = I$.

2. Theoretical Analysis

Theoretically, it can be shown that SVD compression stores less data with lower rank approximations than binary compression does. Given an m by n image, we have mn pixels. With binary compression, each pixel is stored as an integer, which takes up one byte in memory. Thus, disregarding the values found in the header for width, height, and grayscale values, binary compression stores approximately $O(mn)$ bytes. With SVD compression, an m by m matrix U, m by n matrix Σ, and n by n matrix V are calculated. Due to the nature of rank approximation, only a portion of these matrices need stored. Given a rank of k, only the first k columns need stored for U, k number of singular values on the diagonal of Σ, and the first k rows of V^T. The elements of each matrix are stored as floating point values, which take up 4 bytes of memory. Thus, SVD compression takes $4 * (km + k + kn) = O(km + kn + k) = O(k * (m + n + 1))$. While the compression here is directly correlated to the value of the rank, its value is constant and can be removed from the equation. Thus it reduces to $O(m + n)$ complexity in terms of memory, which is significantly smaller than binary compression.

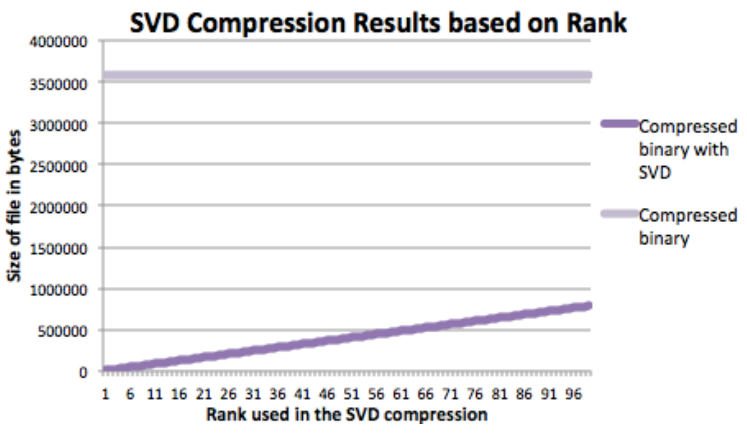
3. Experimental Results

Two image compression procedures were used for the purposes of the project. The first was a simple binary compression in which the original PGM file in ASCII format was compressed into a binary format that is easily decompressed back. The second procedure was singular value decomposition (SVD). The original image is converted into a matrix, where the integer gray-scale values are the elements of the matrix A. A is then factored into three matrices represented by floating point values, whose elements can be reduced based on the rank approximation, as explained in section 1.

3.1 SVD Results

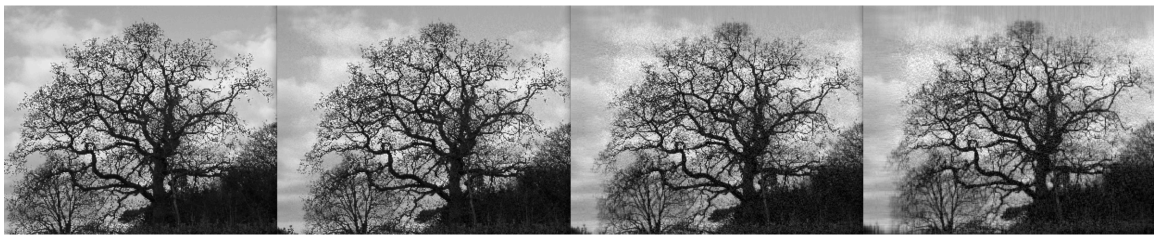
By executing `test storage`, a series of random images are created ranging from size 50 by 50 to 1K by 1K pixels, incrementing by 50 pixels for each step. The size of each image is stored in a resulting CSV file. The size of the compressed file using ASCII to binary is stored and the decompressed image's size. For each image, its header and SVD text file is obtained, which is used to compress and decompress the file using rank reduction with a k value of 1 to 100, incrementing by 1. The resulting size of the compressed and decompressed file is also stored in the results.

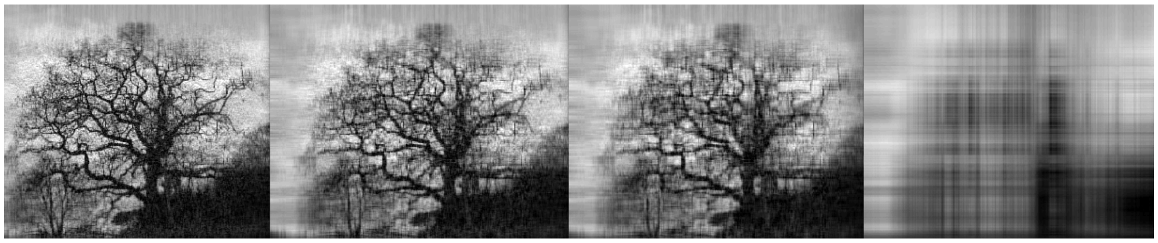
The graph to the right shows the compression rates for both SVD and simply binary on a PGM image, as obtained from the automatic storage test ranging from rank 1 to 10. The graph also shows the linear affect that *k* has on the size of the compressed image using SVD. In addition, with a constant value of k = 1, the following table exemplifies that the storage increases linearly based on the width and height of the image using SVD, while increasing polynomially using simple binary compression.



Visually, the increasing values of rank affect how accurately the image is returned to its approximated original state. The higher the rank, the clearer the decompressed image is. This can be seen in the image of a fractal tree below. The image is 2272 x 1704 pixels. The original image, as well as the resulting images when using SVD of varying rank approximations can be seen below.

m=n	total pixels = m * n	size of original ascii file (bytes)	theoretical binary (m*n)	size of compressed binary file (bytes)	theoretical svd (m+n)	size of compressed binary SVD file (bytes)
50	2500	8980	2500	2505	250	411
100	10000	35802	10000	10005	500	811
150	22500	80509	22500	22505	750	1211
200	40000	143100	40000	40005	1000	1611
250	62500	223568	62500	62505	1250	2011
300	90000	321926	90000	90005	1500	2411
350	122500	438157	122500	122505	1750	2811
400	160000	572274	160000	160005	2000	3211
450	202500	724277	202500	202505	2250	3611
500	250000	894157	250000	250005	2500	4011
550	302500	1081927	302500	302505	2750	4411
600	360000	1287574	360000	360005	3000	4811
650	422500	1511097	422500	422505	3250	5211
700	490000	1752513	490000	490005	3500	5611
750	562500	2011805	562500	562505	3750	6011
800	640000	2288979	640000	640005	4000	6411
850	722500	2584036	722500	722505	4250	6811
900	810000	2896978	810000	810005	4500	7211
950	902500	3227803	902500	902505	4750	7611
1000	1000000	3576512	1000000	1000005	5000	8011

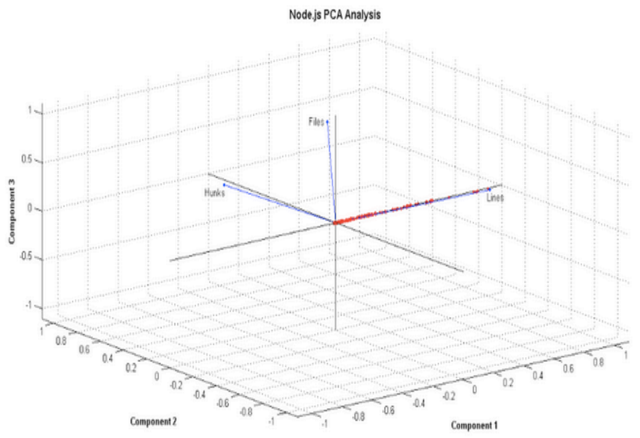

original k = 500 k = 300 k = 200

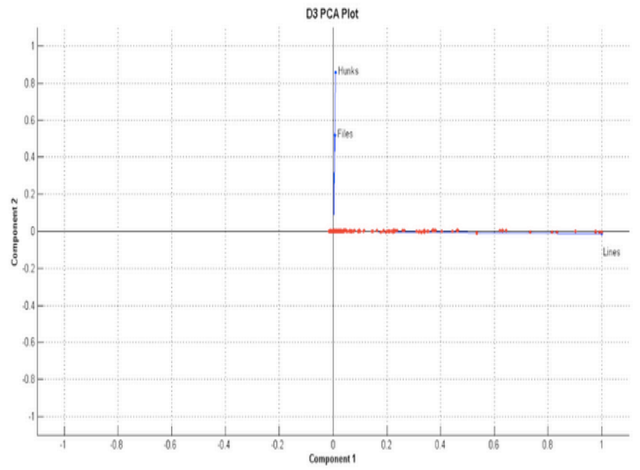

k = 100 k = 50 k = 20 k = 3

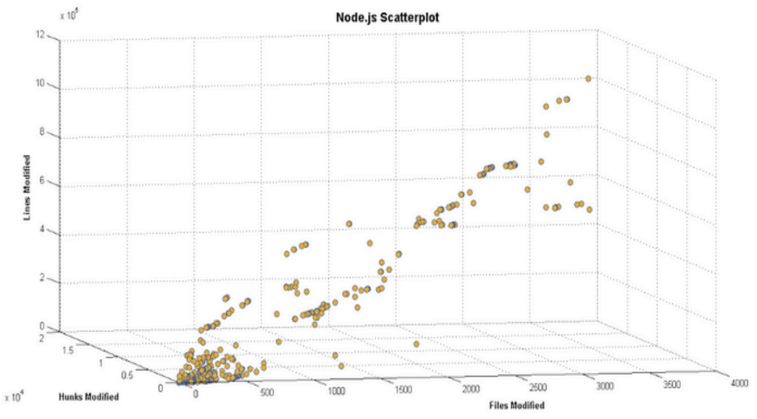
As can be seen in the fractal tree figure above, the upper part of the image that had the clouds was fuzzier intuitively because the sky is the same color, so when the dimensionality is reduced, that portion of the data set varies so little that it would be among one of the smaller singular values to scale the rotated matrices. Thus that difference would be removed with a lower rank approximation.

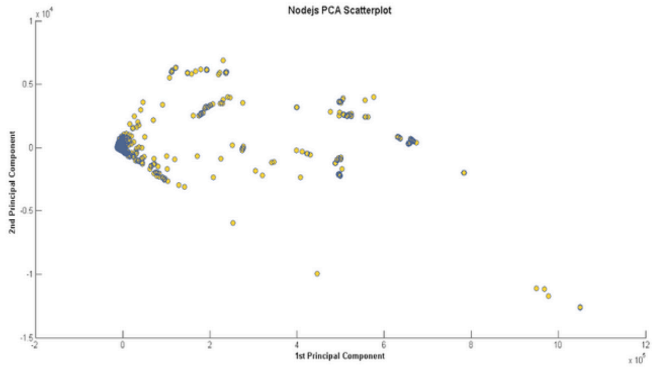
3.2 PCA Results

The data set used for PCA was a series of commit metrics from the git repository history of the Node.js and the D3 javascript library. For each commit, the number of lines modified, hunks modified, and files modified were recorded. These three features were then run through MatLab using principal component analysis to generate the following diagrams.









The upper left graph is a 3D view of the Node.js repository commits being mapped to the 2D plane, which is shown in the upper right graph. The lower left scatterplot is the original data gathered from the repository, based on lines, hunks, and files modified. The lower right scatterplot is the result of the principal component analysis. From this, we can determine that the highest variation is found in the number of lines modified per commit. In addition, the smaller the commit size is, the more likely that it correlates with other commit size metrics. However, larger commits are more varied based on the size metric. For example, a large number of lines modified does not necessarily correlate to a large number of hunks modified.

4. Algorithm and Problem Insights

Problems arise in the SVD algorithm when it is used on smaller data sets. Because the matrices are decimal values, they require more bytes to store, and the reduction is performed on three matrices rather than one. The decompressed file using SVD could actually be larger than the original image with a small amount of pixels. At this point, with a small enough data size, binary is more efficient. In addition, because SVD requires the storage of floating point values, a full rank approximation could still be different from the original image due to rounding errors caused by limited storage space.

5. Additional Extra Credit

In order to perform the necessary tests and visualization without requiring the user to separately translate the image into a matrix and perform SVD on it, our program supports the ability to take a given PGM image and creates both the header text file and the SVD text file that is used in the third option. In this case, the user can perform the following operations:

- reducto 5 image.pgm
    - Translates the image into a matrix and performs singular value decomposition on that matrix. Creates *image_header.txt*, which contains the width, height, and maximum gray-scale values, and *image_svd.txt*, which contains the three matrices factored out of the image that are the result of the SVD operation.
- reducto 3 image_header.txt image_svd.txt k
    - Reduces the dimensionality of the original image by only storing the relevant elements of the factored matrices in *image_svd.txt*.

Conveniently, the first operation, which uses “5” as an argument and the location of the image, needs only to be performed once per image. The same header and svd text files can be reused for varying rank approximations.

6. Statement of Responsibilities

Heather was the team leader and coordinator of the project. She designed and implemented the SVD algorithm, and implemented the binary compression and decompression algorithms. She also contributed to the extra credit opportunity that takes an image and gives the resulting SVD and header file to make testing easier, and wrote the storage tests. Heather also wrote up the README documentation, and report.

Drew developed the binary compression and decompression methods and assisted Heather with implementation. He also contributed to the extra credit opportunity that takes an image and gives the resulting SVD and header file to make testing easier. He performed PCA analysis on the datasets using MatLab and worked out some of the theoretical analysis. Drew prepared the presentation slides and contributed to the report documentation.