Exploration of Imputation Methods for Missingness in Image Segmentation

Christopher Peter Makris
Department of Statistics
Carnegie Mellon University [*]

May 2, 2011

Abstract

Within the field of statistics, a challenging problem is analysis in the face of missing information. Statisticians often have trouble deciding how to handle missing values in their datasets, as the missing information may be crucial to the research problem. If the values are missing completely at random, they could be disregarded; however, missing values are commonly associated with an underlying reason, which can require additional precautions to be taken with the model.

In this thesis we attempt to explore the restoration of missing pixels in an image. Any damaged or lost pixels and their attributes are analogous to missing values of a data set; our goal is to determine what type of pixel(s) would best replace the damaged areas of the image. This type of problem extends across both the arts and sciences. Specific applications include, but are not limited to: photograph and art restoration, hieroglyphic reading, facial recognition, and tumor recovery.

Our exploration begins with examining various spectral clustering techniques using semi-supervised learning. We compare how different algorithms perform under multiple changing conditions. Next, we delve into the advantages and disadvantages of possible sets of pixel features, with respect to image segmentation. We present two imputation algorithms that emphasize pixel proximity in cluster label choice. One algorithm focuses on the immediate pixel neighbors; the other, more general algorithm allows for user-driven weights across all pixels (if desired).

[*]Author's address: Department of Statistics, Baker Hall 132, Carnegie Mellon University, Pittsburgh, PA 15213 Email: cmakris@andrew.cmu.edu

Contents

1 Introduction

Within the field of statistics, a challenging problem is analysis in the face of missing information. Statisticians often have trouble deciding how to handle missing values in their datasets, as the missing information may be crucial to the research problem. Common solutions are to either remove the incomplete observations or incorporate the missingness in their analysis. Some approaches that incorporate observations with missingness are mean/median imputation, "hot deck" imputation, and linear interpolation (Altmayer, 2011). Although these methods are widely used, they are not appropriate in all situations and definitely not flawless. For example, they may be inaccurate if the values are not missing completely at random. Missing values are commonly associated with an underlying reason, which can require additional precautions to be taken with the model. This work explores the situation in which the goal is to "restore" the complete set of observations.

For the purpose of this paper, our main datasets will be images. As can be seen in Figure 1a, all images are comprised of a set of adjacent square pixels. In this setting, a complete set of pixels is analogous to a complete dataset. Any pixels and attributes that are damaged or lost are analogous to missing values in a dataset. To "restore" a complete picture, we must determine the type of pixel that would best replace the "damaged" areas of the image.

A statistical method of imputing missing pixels could be applied in fields across both the arts and sciences (Figure 1b-f). For example, if a painting is tarnished, we could take a digital image of the canvas and use our method to determine which colors should be used to repair the artwork. If an antique oriental rug is torn from old age, we could determine the pattern of the stitching to know how to fill in the tattered areas. Ancient Egyptian hieroglyphics are important historical artifacts, yet many are damaged because of wear over time. The restoration of corrupted areas in the artifact and recovery of that which has been lost could help historians. The same technology could be applied in biology or medicine to determine the location of a patient's tumor if an x-ray is not completely clear. Furthermore, the methods we explore could be used in the computer science field to help develop facial recognition software.

We begin by examining the performance of selected spectral clustering methods used for image segmentation. To do so, we simulate artificial datasets to explore the methods' behavior and performance. Included is a discussion of the stability of such algorithms and how it relates to the choice of tuning parameters. We also consider the advantages and disadvantages of multiple ways of representing or quantifying the image pixels. Finally, we assess the performance of selected methods of imputation on actual image data when attempting to impute missing values. The picture of a dancer jumping in Figure 2 will be used as a running illustrative example throughout the paper. We will demonstrate using the smaller section near the dancer's wrist and bracelet just for computational simplicity.

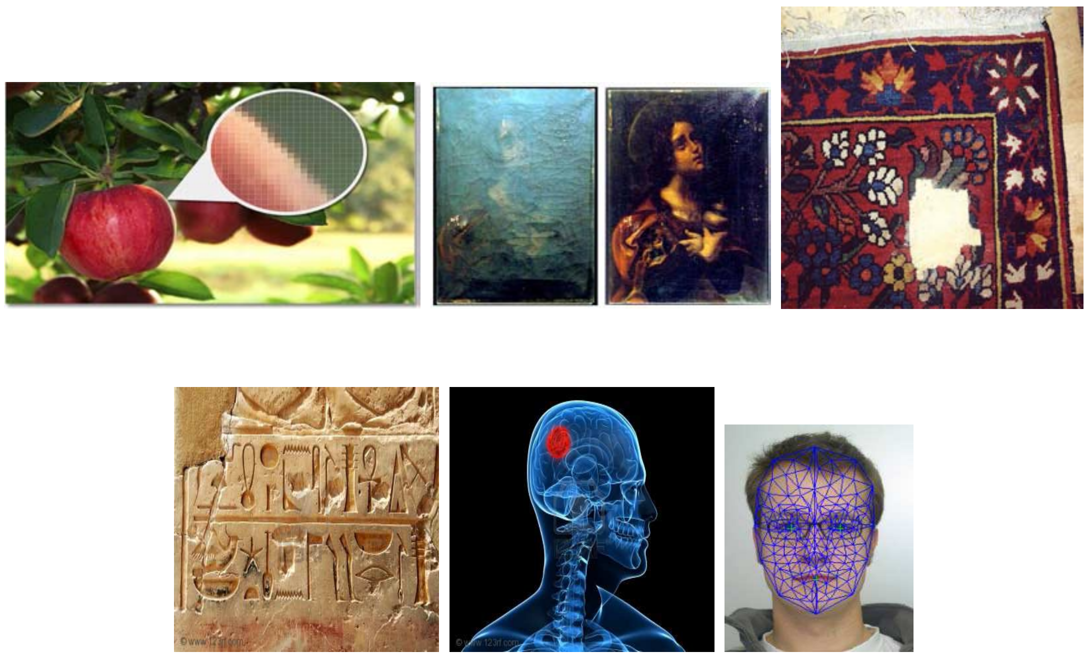


Figure 1: a. Pixelated image of an apple, b. Tarnished artwork, c. Torn Oriental rug, d. Damaged hieroglyphics, e. X-Ray showing tumor, f. Facial recognition software

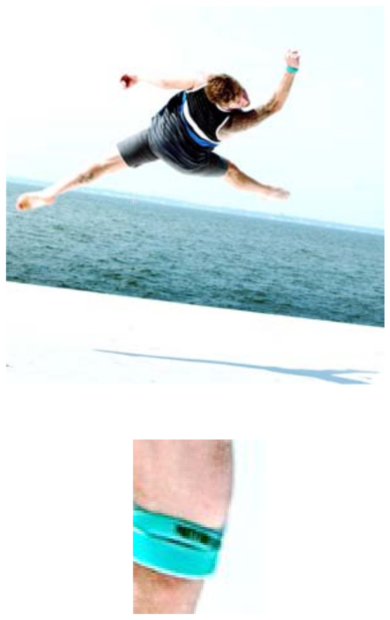


Figure 2: a. Dancer Travis Wall, b. Wrist close-up

2 Image Segmentation Using Spectral Clustering

When given a dataset, we might want to group observations based on their similarities in a process called "clustering." Observations in the same cluster are more similar to each other than to observations in different clusters. Unfortunately, traditional clustering algorithms often rely on assumptions that are not always practical. For example, clustering methods may assume spherical (K-means) (Hartigan, 2011) or elliptical (Model-Based Clustering) (Fraley, 1998) group structure. We propose the use of spectral clustering, a method which relaxes the shape assumption and therefore, in general, can be applied more broadly.

Spectral clustering algorithms consist of two main parts: the representation of a dataset as a connected graph with weighted edges, and the partitioning of the created graph into groups. We sketch the basic framework here and discuss differences in algorithms later. We begin by creating an affinity matrix A which is a function of the distances between each pair of observations:

$$A_{i,j} = e^{-\frac{\|x_i - x_j\|^2}{2\sigma^2}}$$

The matrix A is analogous to a completely connected graph where the nodes are our observations and the edge weights correspond to the "affinity" between observations.

The distance measure which we use to construct the affinity matrix does not necessarily have to be squared Euclidean distance as we present above. The choice of distance measure is dependent on the application, and thus may vary. Furthermore, the affinity matrix also depends upon a tuning parameter σ which defines the neighborhood of each observation. The σ value plays a part in defining how close one observation is to another, and thus how easily two observations may be considered members of the same cluster. Holding all else constant, as σ increases the values of the affinity matrix increase as well. Any arbitrary pair of observations is considered to be closer to one another as σ gets larger. An observation has an affinity of 1 with itself or any exact duplicate; the lower bound of zero theoretically occurs when $\|x_i - x_j\| = \infty$.

To partition our dataset (or segment our image) into K clusters, we find the leading K eigenvectors of the affinity matrix and input them into an algorithm such as K-means. Note that the use of K-means does not imply spherical groups in the original feature space. Instead, the eigenvectors project the data into a space where the groups are considerably further apart and can easily be found with a process such as K-means. This produces a set of labels which end up being the clusters to which we assign the observations.

2.1 The K-Means Clustering Algorithm

For this thesis, we will implement the K-means algorithm (Hartigan, 2011) because it is computationally fast and easy to use. The goal is to group our observations into K clusters such that we minimize the within-cluster sum of square distances. For clusters $C_1, C_2, C_3, ..., C_K$, we minimize the following:

$$\sum_{k=1}^{K} \sum_{x_i \in C_k} (x_i - \overline{x}_k)^2$$

The K-means algorithm begins by randomly assigning K initial cluster centers. Next, the algorithm assigns every observation to the closest cluster center. Lastly, the algorithm updates the definition of its K new centers by finding the centroid of each of the observations in the same cluster. The algorithm iterates through the "assignment" and "updating center" steps until it reaches a convergent solution.

2.2 Initial Test Groups

Within this thesis we explore a selection of spectral clustering approaches: Perona/Freeman (Perona, 2010), Ng/Jordan/Weiss (Ng, 2001), and Scott/Longuet-Higgins (Scott, 1991) algorithms. These methods use the same basic underlying framework but vary the algorithm slightly. We will see that these variations can result in very different segmentation results.

To test and compare the performance of these algorithms, we simulate four different artificial datasets depicted in Figure 3. We use this non-image data to assess performance because we know the true cluster labels. Thus, we will be able to calculate and compare error rates.

- The first dataset consists of four well-separated Gaussian clusters with centers (1,1), (4,1), (1,4), and (4,4), all with standard deviations of .5 in both the x_1 and x_2 directions. Each cluster contains 100 observations.

- The second dataset consists of four overlapping Gaussian clusters with centers (1,1), (2,1), (1,3), and (2,3), all with standard deviations of .5 in both the x_1 and x_2 directions. Each cluster contains 100 observations.

- The third dataset consists of two well-separated splines. The splines are crescent-shaped and are very near each other, but do not intersect. The splines have added random Gaussian noise with mean 0 and standard deviation .5 in the direction orthogonal to its tangential curvature.

- The fourth dataset consists of four overlapping splines which make a design representing the shape of a dragonfly. The four splines make up the body, head, left, and right wing of the dragonfly. Each spline has added random Gaussian noise with mean 0 and standard deviation .05 in the direction orthogonal to its tangential curvature.

Traditional clustering algorithms usually correctly capture the group structure with well-separated datasets, yet tend to falter for datasets in which some type of overlapping or non-Gaussian structure exists. Therefore, we would expect to see few misclassifications for the first dataset of four well-separated clusters, but many for the remaining three datasets.

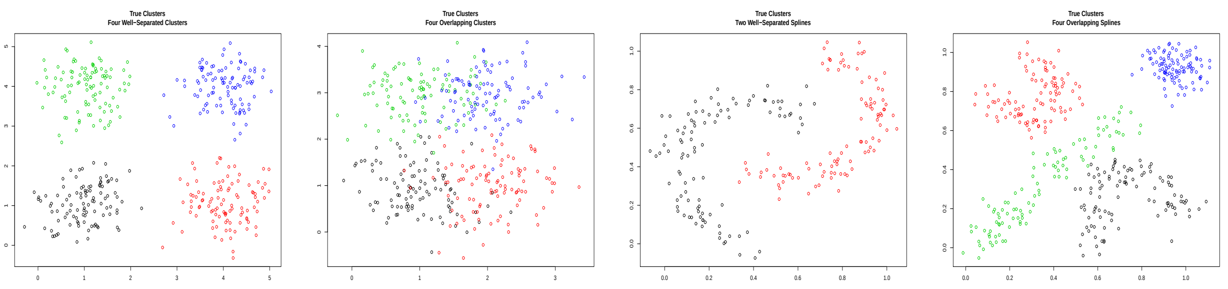


Figure 3: a. Four well-separated clusters, b. Four overlapping clusters, c. Two well-separated splines, d. Four overlapping splines; true cluster labels indicated by color

2.3 Perona/Freeman Method

The Perona/Freeman algorithm (Perona, 2010) is arguably the simplest form of all spectral clustering algorithms. The steps are as follows:

- Construct an affinity matrix $A_{i,j} = e^{-\frac{\|x_i - x_j\|^2}{2\sigma^2}}$
  - Choose a neighborhood σ parameter
  - Choose a method of distance measurement (which for us is the squared Euclidean distance $\|x_i - x_j\|^2$)
- Calculate the eigenvalues and eigenvectors of matrix A
- Use K-means to cluster the leading K eigenvectors

For the purpose of this section on initial spectral clustering solutions, we choose $\sigma = .5$ to be constant and our distance measure to be squared Euclidean distance. We first inspect how the Perona/Freeman algorithm performs when attempting to identify four well-separated clusters (Figure 4).

Figure 4a displays a heat map of the affinity matrix which helps us visually examine the relationships between all pairs of observations. The scales of both axes are irrelevant; however, the axes themselves represent the order in which the observations are indexed. For our artificial datasets, the observations are indexed in order of the true groups. Pairs of observations that have a comparatively high Euclidean distance (and therefore are quite dissimilar and so have a low affinity) appear in the darker deep orange and red colors, whereas pairs of observations that have a comparatively low Euclidean distance (and therefore are quite similar and so have a high affinity) appear in the lighter yellow and white colors. Observing some type of block-diagonal structure within our heat maps would imply that the true groups are being identified. In this case, the four true clusters of equal sizes are being recovered as indicated by the clear block-diagonal structure.

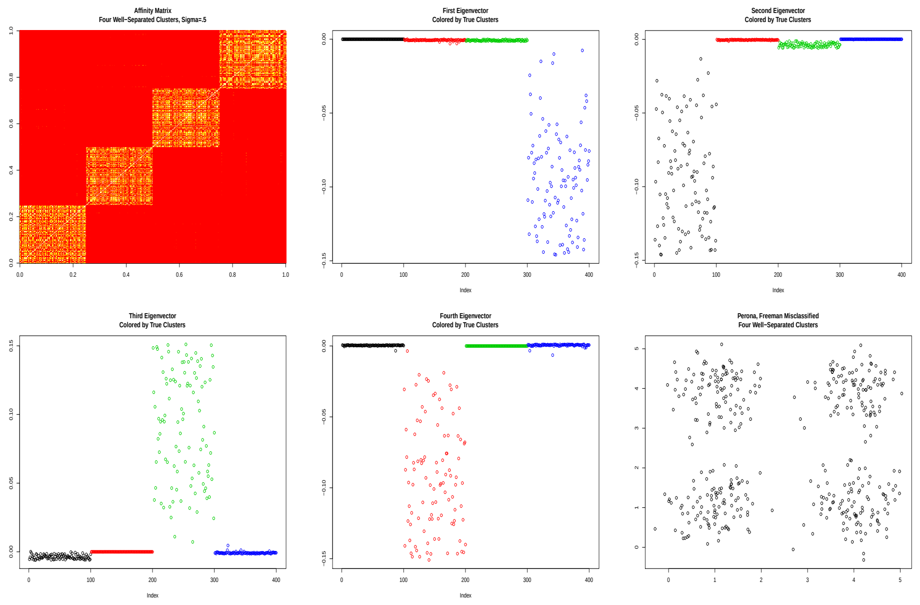


Figure 4: a. Euclidean affinity matrix, b. First eigenvector, c. Second eigenvector, d. Third eigenvector, e. Fourth eigenvector, f. Perona/Freeman misclassification

Figures 4b-e show a graphical representation of the first four leading eigenvectors of the affinity matrix. The x-axis represents the index of each observation, and the y-axis represents the value in each respective entry of the eigenvector. The color of the datapoints corresponds to the true group labels. Observations colored in black, red, green, and blue correspond to groups one through four, respectively. In this case, there is great separation information within the first four leading eigenvectors. We can see that if the scatterplots of the eigenvectors were projected onto the y-axis, we could still very clearly distinguish between the groups of observations. More specifically, the first eigenvector helps recover the fourth group, the second eigenvector helps recover the first group, the third eigenvetor helps recover the third group, and the fourth eigenvector helps recover the second group. Therefore, each eigenvector is associated with the recovery of a true group.

The final graph of Figure 4 displays a scatterplot of the testing group with coloring by misclassification. Any observations that were assigned to the wrong cluster would be highlighted in red, whereas observations that were assigned to the correct cluster are shown in black. Note that in this case none of the observations were misclassified (error rate = 0.0000).

Next we inspect how the algorithm performs when attempting to identify the four overlapping clusters (Figure 5). In this case, the heat map of the affinity matrx still identifies the four main clusters, yet the scattered yellow color around the block-diagonal structure shows that there is some uncertainty especially when trying to distinguish groups one and two from each other and groups three and four from each other. Groups one and two are

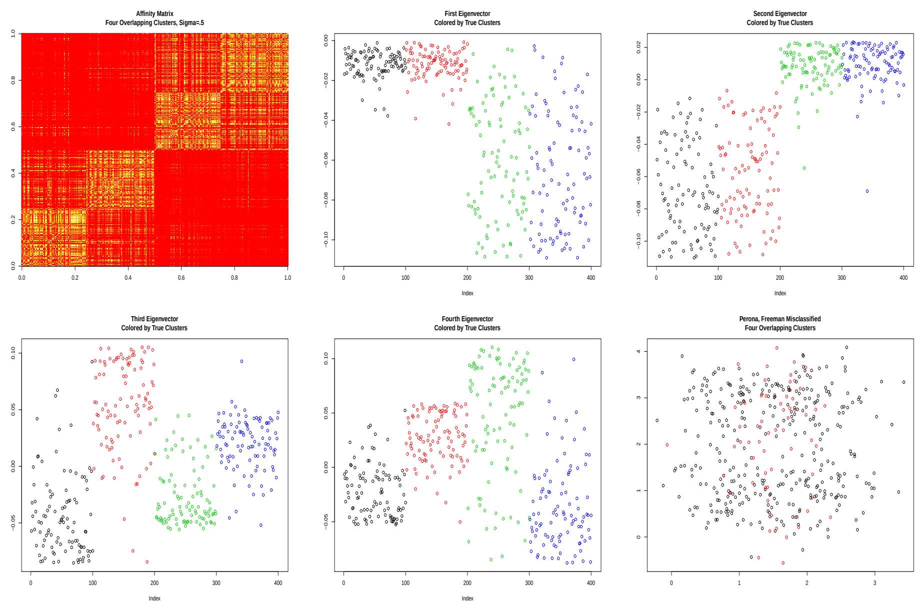


Figure 5: a. Euclidean affinity matrix, b. First eigenvector, c. Second eigenvector, d. Third eigenvector, e. Fourth eigenvector, f. Perona/Freeman misclassification

more similar to each other than to the other groups. The same is true for groups three and four. The first two eigenvectors essentially contain the same information as each other by helping determine whether an observation is in either the union of groups one and two, or the union of groups three and four. Likewise, the third eigenvector somewhat helps distinguish observations in either groups one and three from those in either groups two and four. Lastly, the fourth eigenvector somewhat helps distinguish observations in the union of groups one and four from those in the union of groups two and three. Note that in general, there is less separation information (and thus much more overlap of groups in the dataset) in the eigenvectors as compared to the first initial test dataset of four well-separated clusters. The misclassification scatterplot shows that observations near the center of the feature space are more likely to be misclassified. We would expect this area to be the most problematic to recover this is where the greatest amount of overlap exists. Ultimately, 62 out of the 400 observations were misclassified (error rate = 0.1550).

We also explore the performance of PF on non-Gaussian data, such as the two-well separated splines in Figure 6. Although the two main groups are identified, there seems to be a great uncertainty as shown by the heat map. The complete overlap in the first leading eigenvector highlights this uncertainty. On the other hand, the second leading eigenvector begins to separate the observations into the two true clusters. This may be an indication that using more eigenvectors would yield a better solution; however, the general algorithm for spectral clustering dictates that we use as many eigenvectors as we have clusters. In this case, we know that K=2, so we must use no more than two eigenvectors. The problematic

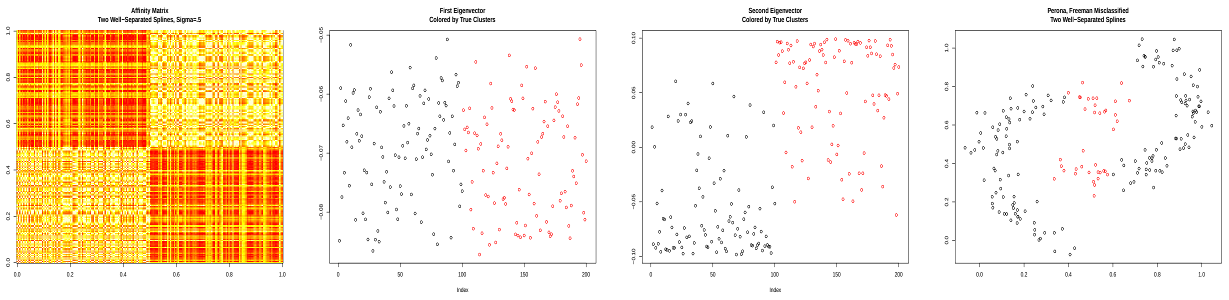


Figure 6: a. Euclidean affinity matrix, b. First eigenvector, c. Second eigenvector, d. Perona/Freeman misclassification

observations appear near the center of the image where the splines begin to come very close to one another. Thirty-eight of the 200 observations ended up being misclassified (error rate = 0.1900).

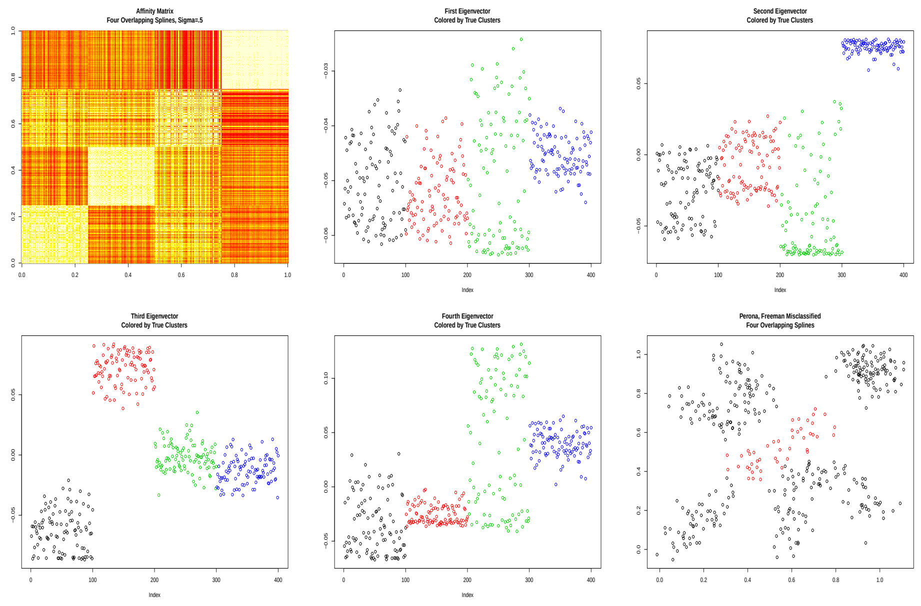


Figure 7: a. Euclidean affinity matrix, b. First eigenvector, c. Second eigenvector, d. Third eigenvector, e. Fourth eigenvector, f. Perona/Freeman misclassification

Lastly, we examine how the algorithm performs when attempting to identify four more complicated splines (Figure 7). Once again, we see block-diagonal structure that recovers the four groups, yet there is some uncertainty especially with observations in group three. Observations truly in group three often may be considered as members of group one or two. The first leading eigenvector contains a lot of overlap, but the second clearly strips group four from the rest of the observations. Similarly, the third leading eigenvector shows a deviation between groups one, two, and the union of three and four. It seems as if the fourth leading eigenvector primarily tries to identify those observations in the fourth group; however, there still exists some separation for the remaining groups with varying levels of overlap. Observations that tend to be misclassified are near the center of the plane where three of the groups nearly meet. We see that 46 out of the 400 observations are misclassified (error rate = 0.1150).

2.4 Ng/Jordan/Weiss Method

We now move on to the Ng/Jordan/Weiss spectral clustering algorithm (Ng, 2001), an extension of the Perona/Freeman algorithm that implements some normalization techniques. The steps are as follows:

- Construct an affinity matrix $A_{ij} = e^{-\frac{\|x_i - x_j\|^2}{2\sigma^2}}$
  - Choose a neighborhood σ parameter
  - Choose a method of distance measurement
- Create a diagonal matrix D whose (i,i)-element is the sum of the affinity matrix A's i-th row: $D_{ii} = \sum_{j=1}^{n} A_{ij}$
- Create the transition matrix $L = D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$
- Calculate the eigenvalues and eigenvectors of matrix L
  - Create the maximum eigenvector matrix X using the largest K column eigenvectors of L, where K is chosen by the user
- Create normalized vector matrix Y from matrix X by renormalizing each of X's rows to have unit length: $Y_{ij} = \frac{X_{ij}}{(\sum_j X_{ij}^2)^{\frac{1}{2}}}$
- Use K-means to cluster the leading K rows of matrix Y

First we use the Ng/Jordan/Weiss algorithm to group the four well-separated Gaussian data (Figure 8). The affinity matrix clearly shows recovery of the four groups. The first eigenvector does not contain much separation information, but just barely distinguishes observations of groups two, three, or the union of groups one and four. The second eigenvector helps distinguish primarily between observations in the union of groups one and three from those in the union of groups two and four, yet also separates groups one and three. The third eigenvector primarily distinguishes observations in the union of groups one and two from those in the union of groups three and four. Finally, the fourth eigenvector primarily helps distinguish between observations in the union of groups one and four from those in the union of groups two and three. Ultimately, none of the observations were misclassified (error rate = 0.0000)

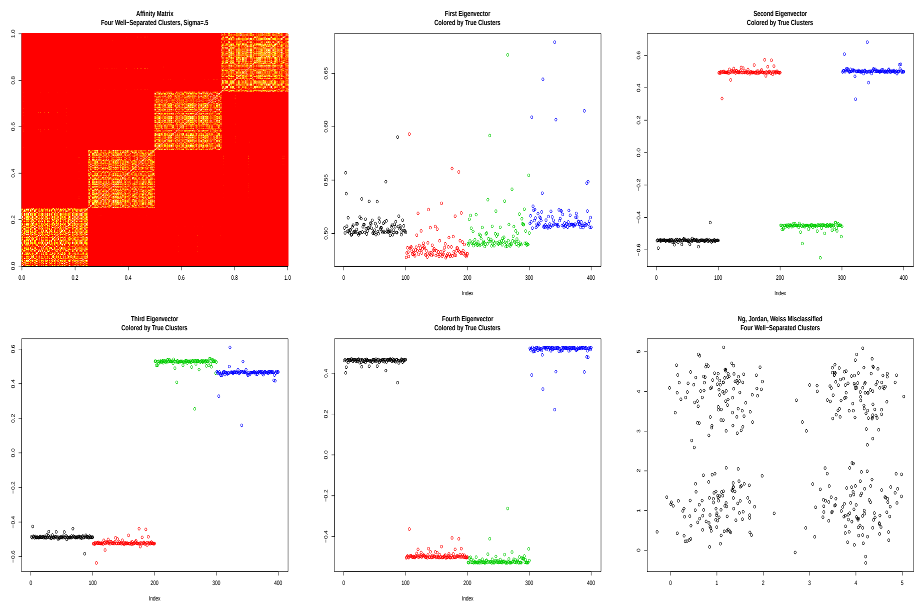


Figure 8: a. Euclidean affinity matrix, b. First eigenvector, c. Second eigenvector, d. Third eigenvector, e. Fourth eigenvector, f. Ng/Jordan/Weiss misclassification

We turn to our four overlapping clusters (Figure 9). The heat map is very similar to the resulting heat map of the affinity matrix produced by the Perona/Freeman algorithm. Once again, the block-diagonal stricture indicates that four groups have been recovered; however, there is some uncertainty between group labels, especially between distinguishing between observations of the first two clusters and the last two clusters. There is virtually no separation information contained in the first eigenvector. The second eigenvector helps distinguish between observations in the union of the first two clusters and observations in the union of the last two clusters. There is some overlap in the third eigenvector, but it mainly helps distinguish observations in the union of the first and third clusters from the those in the union of the second and fourth clusters. There is overlap in the fourth eigenvector, but it serves the same purpose as the second eigenvector. Note that 62 out of the 400 observations are misclassified (error rate = 0.1550). These observations tend to be near the center of the image once again, where there is the most overlap in different clusters, just as we saw when using the Perona/Freeman algorithm.

Next we turn to two well-separated splines (Figure 10). Both the heat map and eigenvectors look similar to those produced by the Perona/Freeman algorithm applied to the same dataset. Once again, 38 out of the 200 observations specifically near where the splines are closest become misclassified (error rate = 0.1900). The same observations are problematic when using the Perona/Freeman algorithm.

The graphs in Figure 11 show how the Ng/Jordan/Weiss algorithm performs when trying to cluster the four overlapping splines dataset. The heat map of the affinity matrix and the

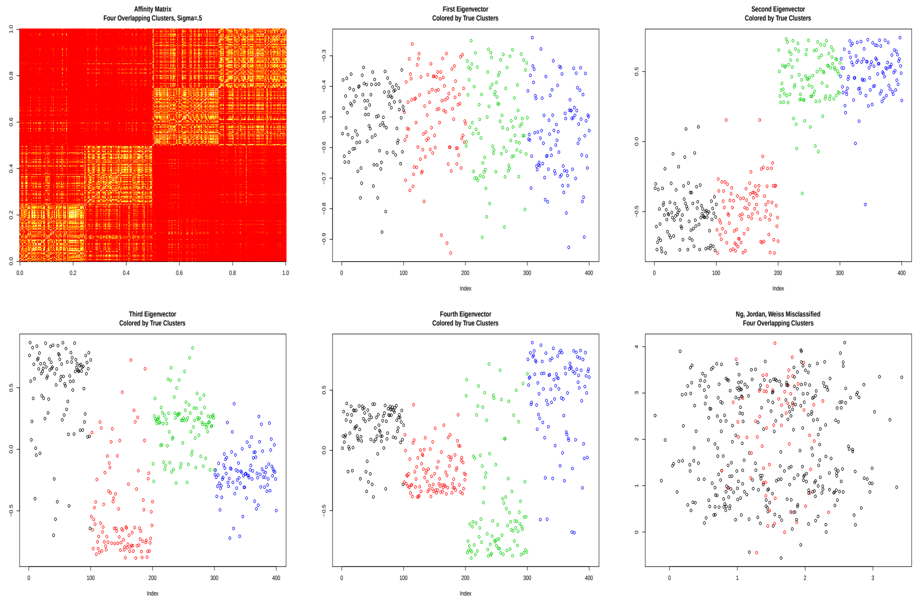


Figure 9: a. Euclidean affinity matrix, b. First eigenvector, c. Second eigenvector, d. Third eigenvector, e. Fourth eigenvector, f. Ng/Jordan/Weiss misclassification

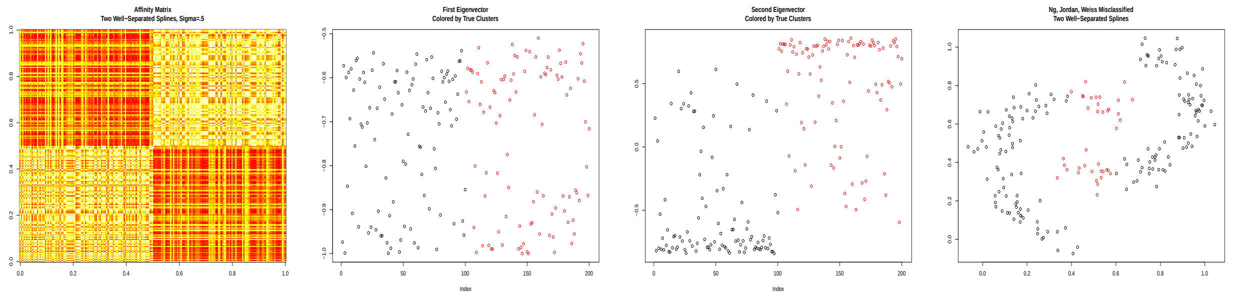


Figure 10: a. Euclidean affinity matrix, b. First eigenvector, c. Second eigenvector, d. Ng/Jordan/Weiss misclassification

graphs of the first four eigenvectors display that the Ng/Jordan/Weiss algorithm performs nearly identically to the Perona/Freeman algorithm in respect to the four overlapping splines. The interpretations of the heat map and eigenvectors, and the general location of misclassified observations is the same. The only difference is that two fewer observations, a total of 44 out of 400, were misclassified (error rate = 0.1100); however, the mislassified observations are the same as when using the Perona/Freeman algorithm.

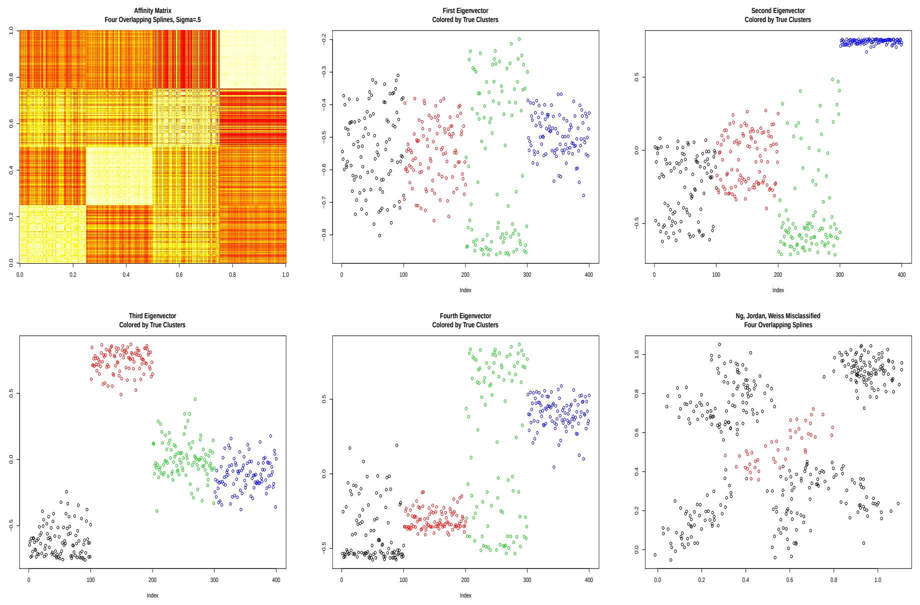


Figure 11: a. Euclidean affinity matrix, b. First eigenvector, c. Second eigenvector, d. Third eigenvector, e. Fourth eigenvector, f. Ng/Jordan/Weiss misclassification

2.5 Scott/Longuet-Higgins Method

The final algorithm we expriment with is the Scott/Longuet-Higgins algorithm (Scott, 1991). The method is primarily based upon spectral clustering and also normalizes similar to the Ng/Jordan/Weiss algorithm. The steps are as follows:

- Construct an affinity matrix $A_{ij} = e^{-\frac{\|x_i - x_j\|^2}{2\sigma^2}}$
  - Choose a neighborhood σ parameter
  - Choose a method of distance measurement
- Create a maximum eigenvector matrix X using the largest K column eigenvectors of A
- Create a normalized vector matrix Y from matrix X so that the rows of Y have Euclidean norm: $Y(i, \rightarrow) = Y(i, \rightarrow)/\|Y(i, \rightarrow)\|$
- Create the matrix $Q = YY^T$
- Use K-means to cluster the leading K rows of matrix Q

We use the Scott/Longuet-Higgins algorithm to group the four well-separated Gaussian groups (Figure 12). Once again, the heat map of the affinity matrix shows a clear recovery of

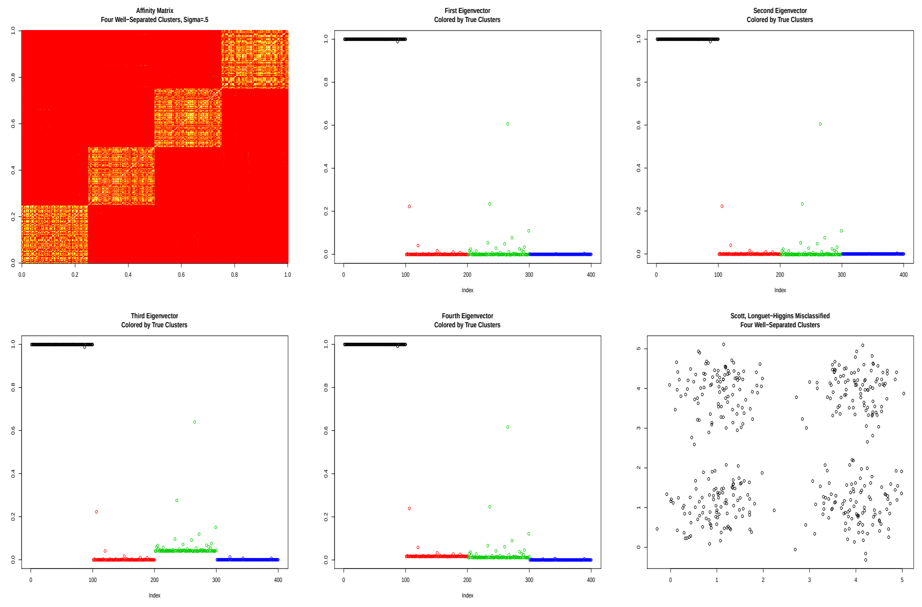


Figure 12: a. Euclidean affinity matrix, b. First eigenvector, c. Second eigenvector, d. Third eigenvector, e. Fourth eigenvector, f. Scott/Longuet-Higgins misclassification

four groups. All four leading eigenvectors of the affinity matrix contain the same information: they separate the first group from the remaining three. The third eigenvector also appears to separate group three from those remaining. Also, the fourth eigenvector seems to separate all four clusters from each other, albeit very slightly. As we would expect, none of the observations were misclassified (error rate = 0.0000).

Next we inspect how the algorithm performs when attempting to identify the four overlapping clusters (Figure 13). The heat map of the affinity matrix shows that the four groups are recovered with some uncertainty when distinguishing between observations of the first two and last two groups. There exists at least partial separation information for all groups within the first three eigenvectors. The fourth eigenvector separates observations in the union of the first two groups from those in the union of the last two groups. Misclassified observations once again fall near the center of the scatterplot where the groups intersect. Fifty-nine out of the 400 observations were misclassified (error rate = 0.1475).

We can see from the graphs in Figure 14 that the Scott/Longuet-Higgins algorithm performs on the well-separated spline data in much the same manner as the other two algorithms. The heat map of the affinity matrix shows the uncertainty of the clustering solution, as do the eigenvectors. The leading eigenvectors both contain separation information; however, there are a handful of observations in both eigenvectors that overlap with the other group. The same number of 38 observations were misclassified out of the total 200 (error rate = 0.1900). These are the same troublesome observations as we saw earlier.

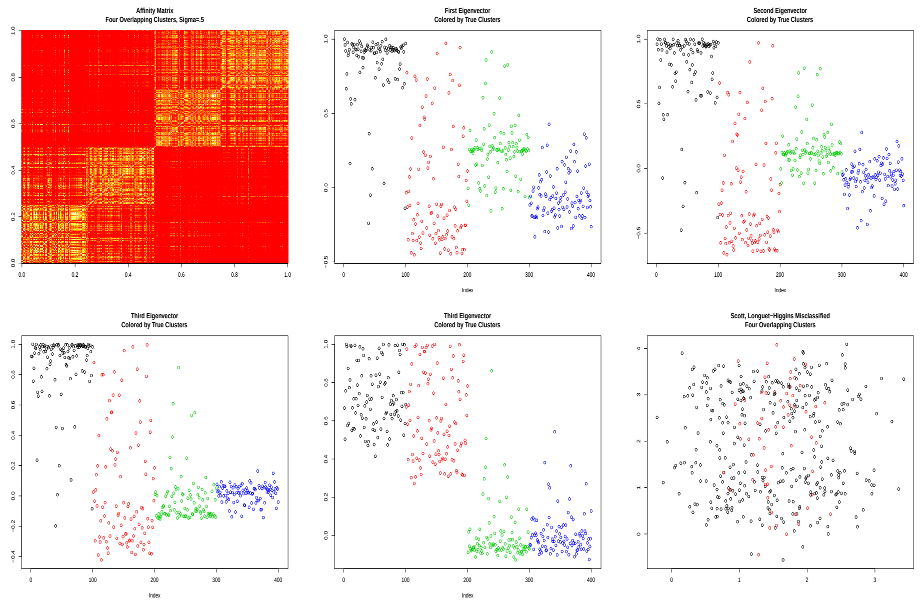


Figure 13: a. Euclidean affinity matrix, b. First eigenvector, c. Second eigenvector, d. Third eigenvector, e. Fourth eigenvector, f. Scott/Longuet-Higgins misclassification

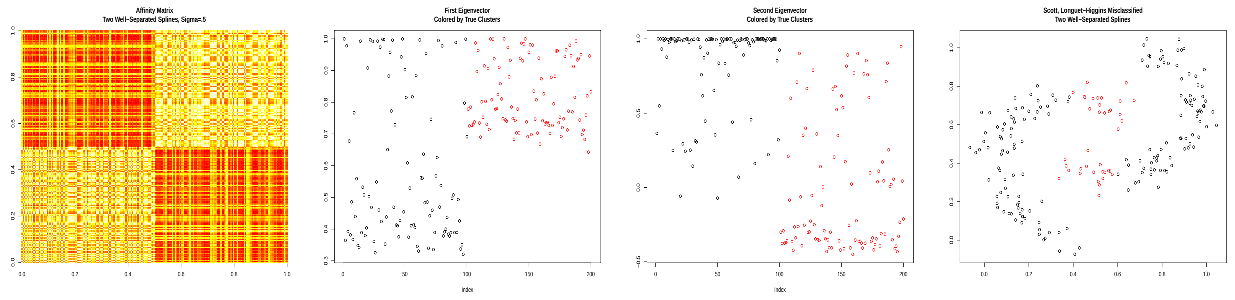


Figure 14: a. Euclidean affinity matrix, b. First eigenvector, c. Second eigenvector, d. Scott/Longuet-Higgins misclassification

Lastly, we explore how the algorithm performs when clustering the four overlapping splines (Figure 15). The same uncertainty is depicted in the heat map of the affinity matrix as compared to the other algorithms. Although there is some overlap in the first, third, and fourth leading eigenvectors, the third group tends to overlap with the other three groups; however, these eigenvectors separate the first group from the other three. The second eigenvector seems to separate all four groups quite well with minimal overlap. Ultimately, 45 out of the 400 observations are misclassified (error rate = 0.1125).

From the table of error rates (Table 1), we can see that all three algorithms perform in a comparable manner. All methods did not encounter any misclassified observations when at-

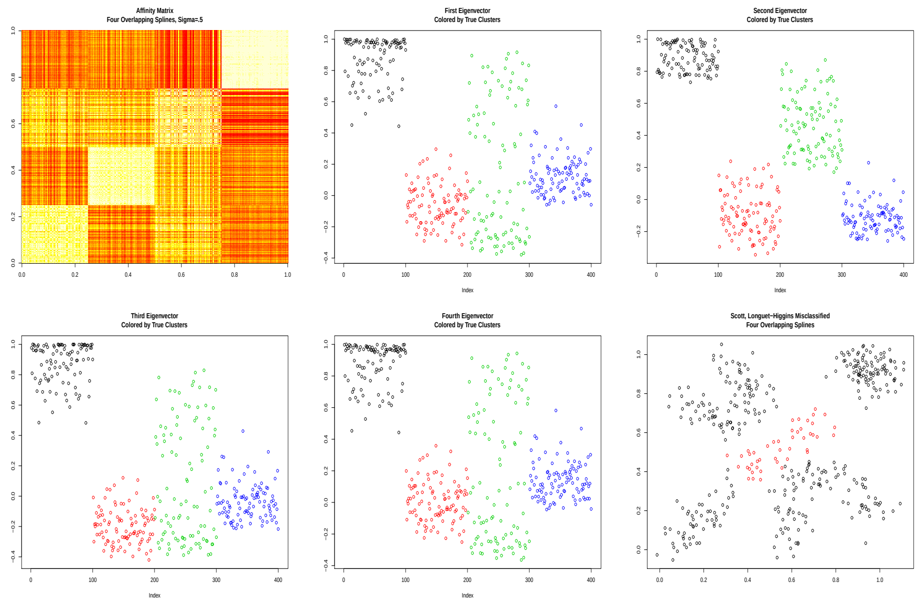


Figure 15: a. Euclidean affinity matrix, b. First eigenvector, c. Second eigenvector, d. Third eigenvector, e. Fourth eigenvector, f. Scott/Longuet-Higgins misclassification

	Perona/Freeman	Ng/Jordan/Weiss	Scott/Longuet-Higgins
Four Well-Separated Gaussians	0.0000	0.0000	0.0000
Four Overlapping Gaussians	0.1550	0.1550	0.1475
Two Well-Separated Splines	0.1900	0.1900	0.1900
Four Overlapping Splines	0.1150	0.1100	0.1125

Table 1: Misclassification rates of algorithms on artificial data

tempting to segment the four well-separated clusters. Sometimes one algorithm outperforms another by just a few observations; however, this may be due to random variation among the K-means clustering solutions. All algorithms have the exact same error rate when classifying observations in the two well-separated splines data. They also perform in approximately the same manner when recovering the true groups of the four overlapping spline data, give or take a few observations.

Our goal is to segment an image. Although we did not extensively explore performance with respect to images, we demonstrate here the slight to no variability seen in segmenting our bracelet picture. Because images are much more complex than our initial test datasets, the choice of the number of clusters for an image may be subjective. For example, if one wanted to cluster the image into sections of the exact same color, then the choice of K would be large (equal to the number of unique pixel colors in the image). On the other hand, fewer clusters would simplify the image segmentation in the sense that pixels comparatively more similar to one another would be grouped together. Because the choice of the optimal number of K clusters can be very subjective and is an open problem in the field of clustering, it is outside the scope of the discussion in this paper.

We apply the Perona/Freeman, Ng/Jordan/Weiss, and Scott/Longuet-Higgins algorithms to the bracelet image (Figure 16). The image is in RGB format (where the image is numerically represented in terms of red, green, and blue color content). We set K=4, and σ=1 (a discussion of the choice of σ values follows in Section 2.5).

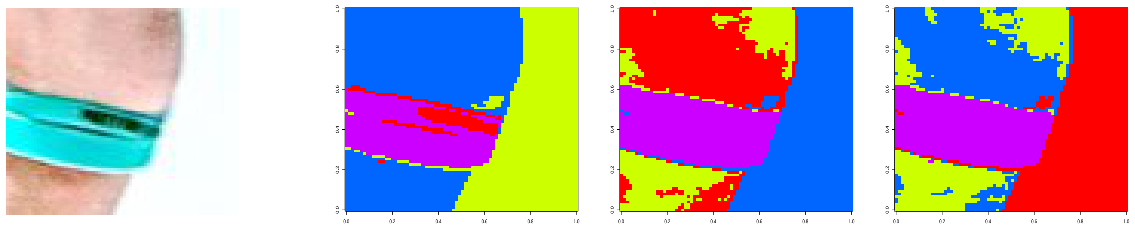


Figure 16: a. Wrist close-up, b. Perona/Freeman, c. Ng/Jordan/Weiss, d. Scott/Longuet-Higgins

For this particular instance, the PF algorithm recovers sharper boundery color changes whereas the NJW and SLH algorithms appear to recover more gradual changes and shading. Depending on whether we desire to pick up more subtle changes when segmenting an image, we may end up choosing a different boundary; however, all three algorithms appear to perform quite well.

2.6 Tuning Parameter σ

The clustering solutions can also heavily depend upon our choice of σ. As σ increases, observations are more likely to have high affinity. Therefore, to determine which σ values would be provide stable solutions for our purposes, we inspect the stability of each algorithm over varying σ values. We simulate 1,000 clustering solutions for each of our artificial datasets since the K-means algorithm is non-deterministic (different random starting centers can result in different sets of clusters). For each set of clusters, we calculate the error rate given the true labels. The σ tuning parameter value ranges from 0.10 to 1.00 for each set of simulations (we increase the σ value by 0.10 each time). Given the context of our analyses, we believe this range of σ values to be reasonable. Additionally, we choose K to be the true number of clusters in the dataset (either 2 or 4 depending on the artifical dataset in question).

We begin with the Perona/Freeman algorithm on four well-separated clusters (Figure 17). In each of the histograms, the dotted black vertical line represents the mean of the error rates, and the solid black vertical line represents the median of the error rates. The black curves are overlaid kernel density estimates with Gaussian shape and default bandwidth. The x-axis

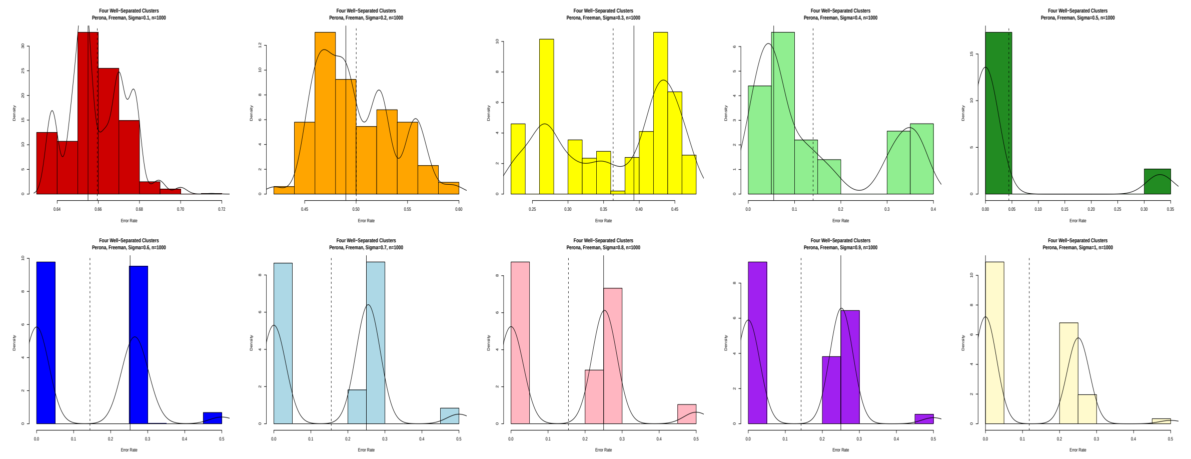


Figure 17: Perona/Freeman error rates for four well-separated Gaussians, σ values from [0.10, 1.00]

represents the error rate for the simulations, and the y-axis represents the density of the histogram bins. We hope to see that for some range of σ values, the histograms appear to have the same features, implying that choosing any σ value within the range would produce comparable clustering solutions. Also, any bimodality that is seen means that there are two common K-means solutions, depending on the starting values.

In this first set of histograms, we see that the error rates are quite erratic for $\sigma \leq 0.40$. For the remaining histograms where $0.50 \leq \sigma \leq 1.00$, there appears to be three clustering solutions with approximate error rates of 0.00, 0.25, and 0.45. The 0.00 error rate clustering solution appears slightly more frequently than the 0.25 error rate; however, they both appear much more frequently than the 0.45 error rate solution. Here, the mean error rate is approximately 0.15 and the median is approximately 0.25.

Next we look at the stability of the PF algorithm on four overlapping clusters (Figure 18). Once again, the error rates span mostly high values for $\sigma \leq 0.30$. On the other hand, when $0.40 \leq \sigma \leq 1.00$, the clustering solutions often yield low error rates around 0.15; however, there is also a very infrequently occuring clustering solution within this span of σ values that produces an error rate of approximately 0.35. The mean error rate is approximately 0.175, and the median error rate is approximately 0.150.

We continue by inspecting the error rates when running PF simulations on the two well-separated splines (Figure 19). The clustering solutions appear to be very stable for any choice of σ between 0.20 and 1.00. All of the histograms within that range display that the most frequent error rate is approximately 0.19. The mean and median error rates are also approximately 0.19. We note that nearly all of the histograms appear to only have one bin, telling us that all the solutions were quite similar to one another. Possibly due to natural variation, the histogram that corresponds to $\sigma = 0.70$ has two bins for error rates of 0.190 and .195 (a difference in one observation being misclassified), but still shows essentially the

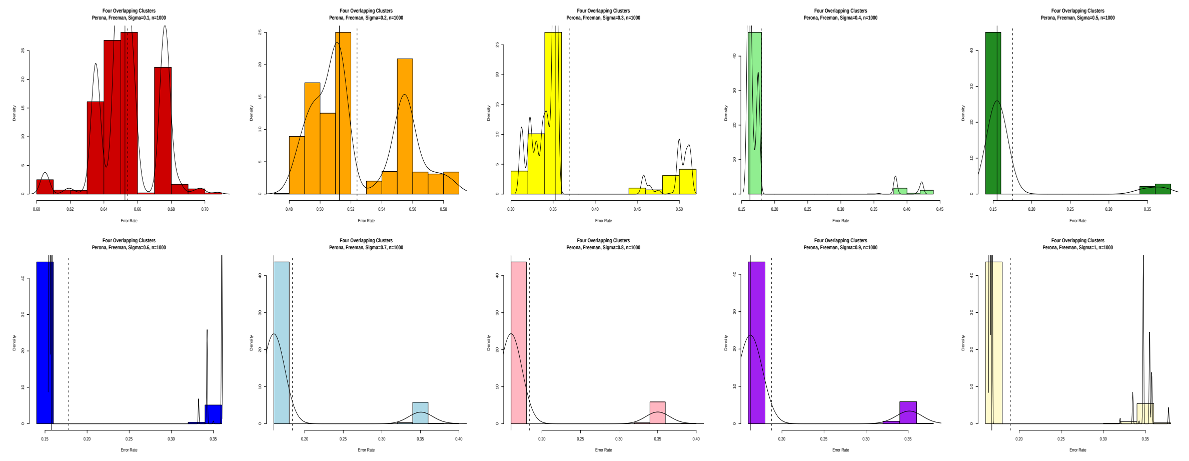


Figure 18: Perona/Freeman error rates for four overlapping clusters, σ values from [0.10, 1.00]

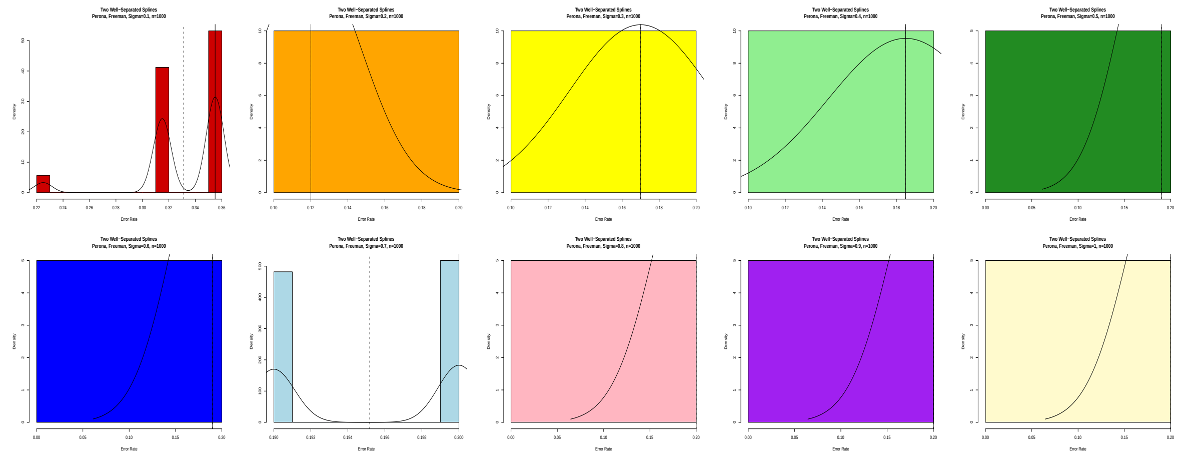


Figure 19: Perona/Freeman error rates for two well-separated splines, σ values from [0.10, 1.00]

same information as the rest of the histograms.

The simulations of the PF algorithm on the four overlapping splines are graphically represented in Figure 20. All of the histograms display that there are three main groups of clustering solution error rates. The most frequently occurring error rate is approximately 0.11. The other two error rate groups of approximately 0.25 and 0.35 appear much less frequently, but in about the same proportion. The overall mean error rate is approximately 0.15, whereas the overall median error rate is approximately 0.11.

We consider the Ng/Jordan/Weiss algorithm and how its clustering solutions vary as the σ value changes. First we graph the distributions of error rates when running the algorithm

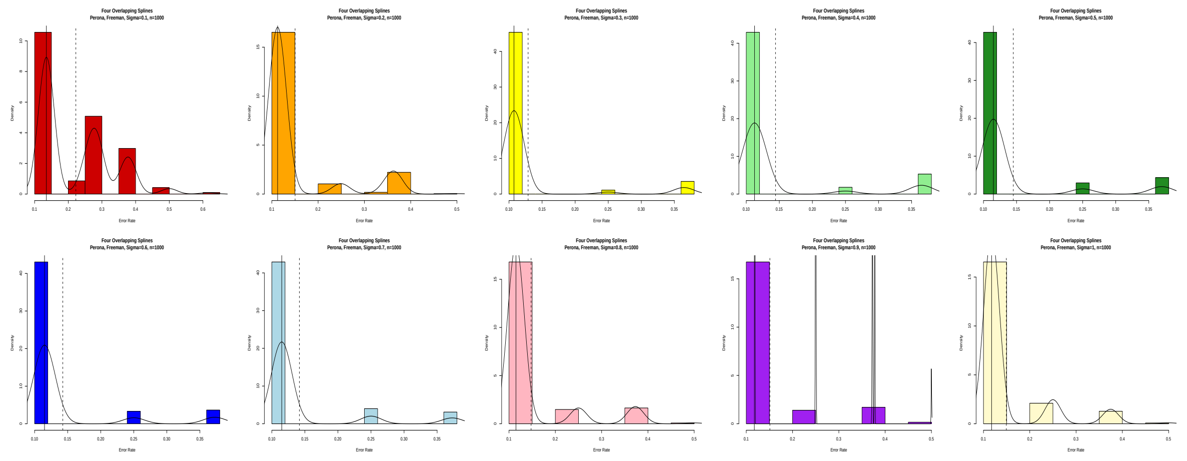


Figure 20: Perona/Freeman error rates for four overlapping splines, σ values from [0.10, 1.00]

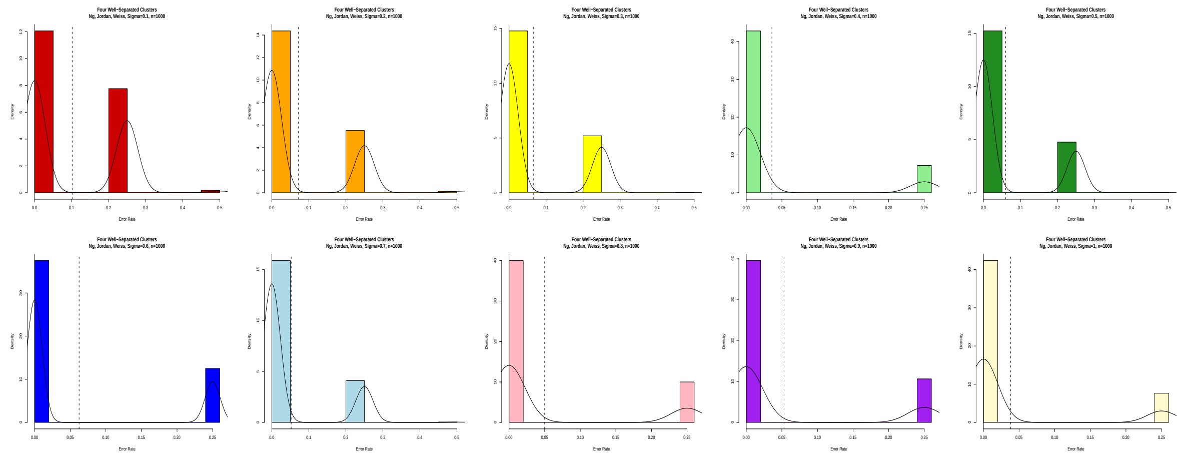


Figure 21: Ng/Jordan/Weiss error rates for four well-separated clusters, σ values from [0.10, 1.00]

with four well-separated clusters (Figure 21). Two clustering solutions are recovered for any value of σ considered: one more frequent with an error rate of approximately 0.000, and one much less frequent with an error rate of approximately 0.25. We note that the frequency of the more inaccurate clustering solution somewhat decreases as we increase the σ value. We also note that for small values of $0.10 \leq \sigma \leq 0.30$ that there is a rare clustering solution that misclassifies nearly half of the observations. The mean and median of the overall error rates is approximately 0.05 and 0.000, respectively.

The stability of the NJW algorithm on four overlapping clusters behaves similarly (Figure 22). It is quite clear by the much higher error rates that values of $0.10 \leq \sigma \leq 0.20$ are a bit unstable for this dataset and algorithm pair. The remaining values of $0.30 \leq \sigma \leq 1.0$

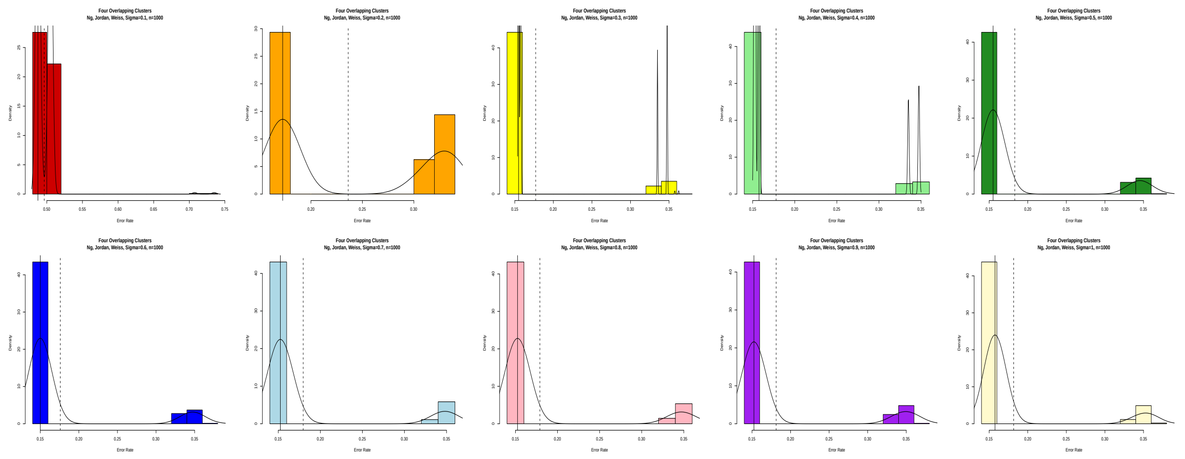


Figure 22: Ng/Jordan/Weiss error rates for four overlapping clusters, σ values from [0.10, 1.00]

all produce similar clustering solution error rates. The most frequent solution has an error rate of approximately 0.15, whereas the rarest solution has an error rate of approximately 0.35. The mean and median of the overall error rates are approximately 0.175 and 0.150, respectively.

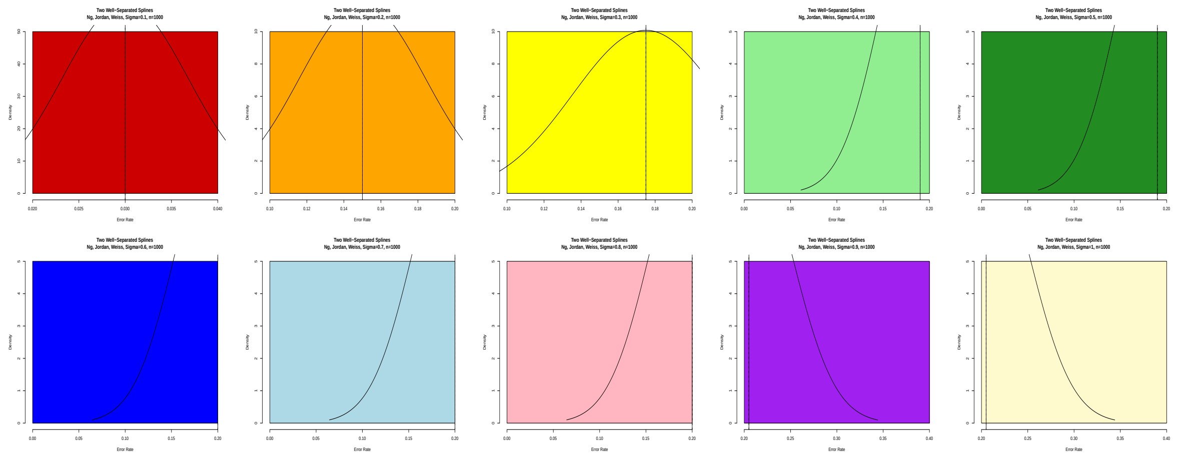


Figure 23: Ng/Jordan/Weiss error rates for two well-separated splines, σ values from [0.10, 1.00]

The results regarding the two well-separated spline dataset using the NJW algorithm display quite interesting results (Figure 23). For any given σ value, the clustering solution appears to yield a consistent error rate. Therefore, we note that within σ values, the NJW clustering solution error rates of the two well-separated splines are very stable. Additionally, we note that in this situation, as the value of σ increases so does the error rate. When

$\sigma = 0.10$, the error rate is extremely small at approximately .003, but when we increase to $\sigma = 0.20$, the error rate increases greatly to approximately 0.15. The error rate continues to grow slowly and steadily as we increase σ, and tends to level-off at around 0.20. Considering all siulation clustering solutions for values of σ, the overall mean and median error rates are both approximately 0.19.

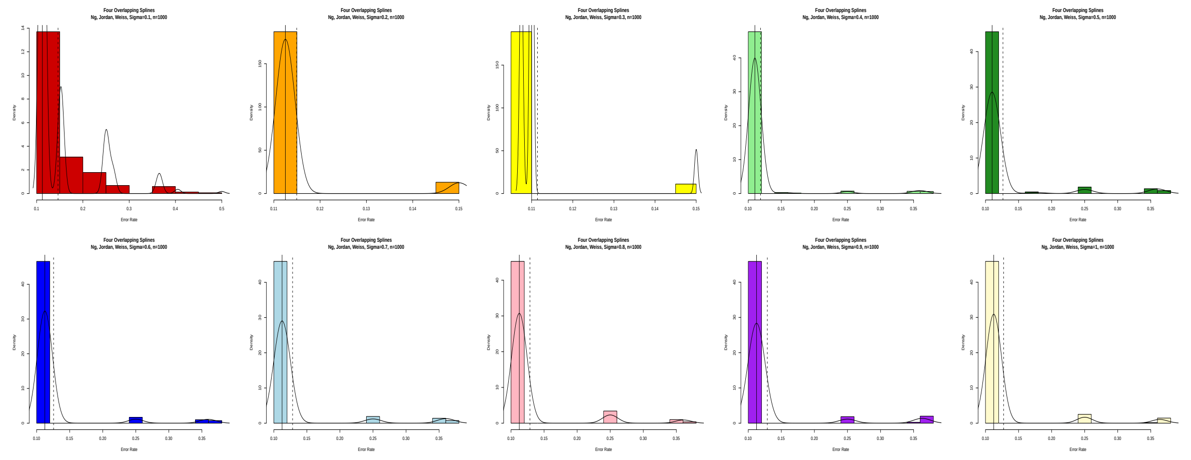


Figure 24: Ng/Jordan/Weiss error rates for four overlapping splines, σ values from [0.10, 1.00]

The stability of the NJW algorithm solutions on the four overlapping clusters is explored in Figure 24. Although slightly more erratic for smaller σ values, the clustering solution error rates appear to be quite consistent over the range of 0.10 to 1.00. In general, the largest proportion of solutions have an error rate of approximately 0.11. Two extremely small groups of clustering solutions with error rates of approximately 0.25 and 0.35 begin to emerge as well for slightly larger σ values. The overall mean and median error rates are both approximately 0.11.

We now move on to the Scott/Longuet-Higgins algorithm, and vary σ as we cluster the four well-separated Gaussian dataset (Figure 25). For $0.10 \leq \sigma \leq 0.30$ there appears to be three main clustering solution error rate groups: approximately 0.00, 0.25, and 0.45. The error rates with the highest frequency are 0.00 and 0.25, followed by the 0.45 group. As σ takes on values greater than 0.30, the highest error rate group of 0.45 no longer appears, and the 0.00 error rate becomes much more frequent than the 0.25 error rate. The mean and median error rates are located at approximately 0.10 and 0.00 in every histogram, respectively, except for the when $\sigma = 0.10$ (mean ≈ 0.16, median ≈ 0.25).

We continue by looking at the stability of the SLH algorithm on four overlapping clusters (Figure 26). Except for the first histogram where $\sigma = 0.10$ where error rates are all approximately 0.35 or 0.40, the error rates for the clustering solutions are all approximately the same for $0.20 \leq \sigma \leq 1.00$. The most frequent error rate is approximately 0.15, yet sometimes there is a clustering solution with a much higher error rate of approximately 0.35. The mean

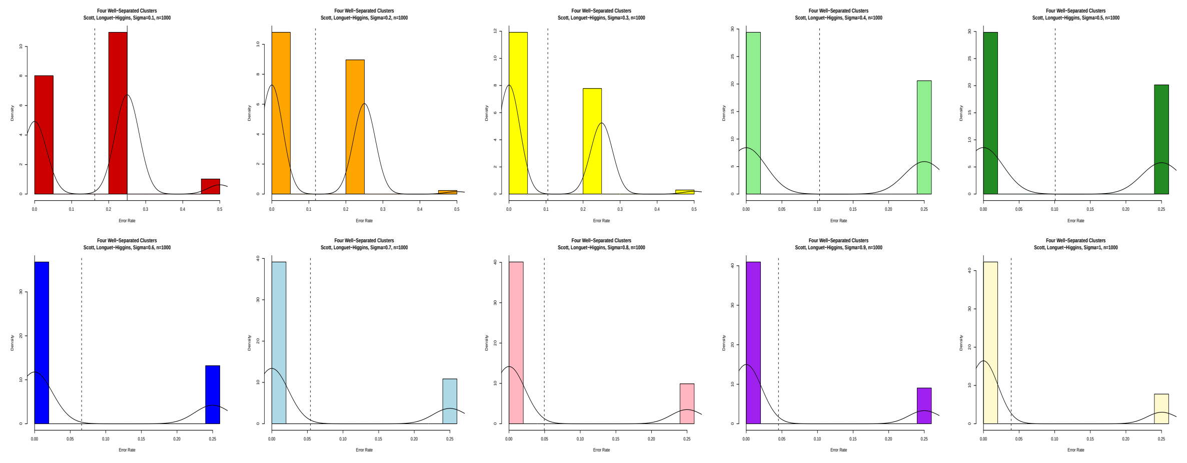


Figure 25: Scott/Longuet-Higgins error rates for four well-separated clusters, σ values from [0.10, 1.00]

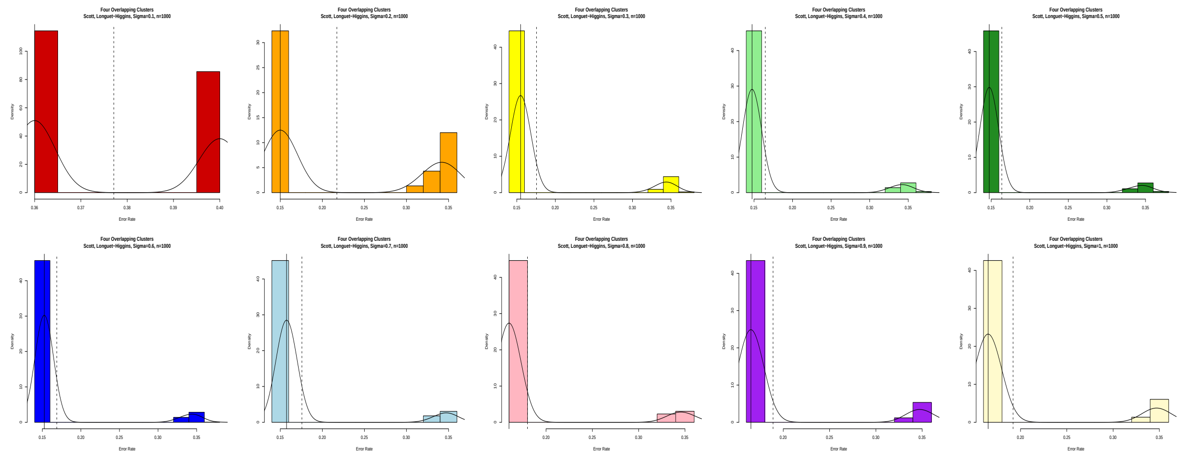


Figure 26: Scott/Longuet-Higgins error rates for four overlapping clusters, σ values from [0.10, 1.00]

and median of the overall clustering solution error rates are 0.175 and 0.150, respectively.

The SLH algorithm's stability of clustering solutions when segmenting the two well-separated splines is depicted in Figure 27. It is interesting to note that, once again, the clustering solution error rates are very constant within values of σ. Therefore, the mean and median are both the same for each of the separate histograms. The error rate is quite high at approximately 0.26 when $\sigma = 0.10$, yet drops to a minimum of about 0.11 when $\sigma = 0.20$. As σ continues to rise, the error rate slowly stabilizes at approximately 0.20.

Lastly, we inspect the stability of the SLH algorithm on the four overlapping splines (Figure 28). Error rates are very erratic for $\sigma = 0.10$, ranging from 0.10 to 0.50; however,

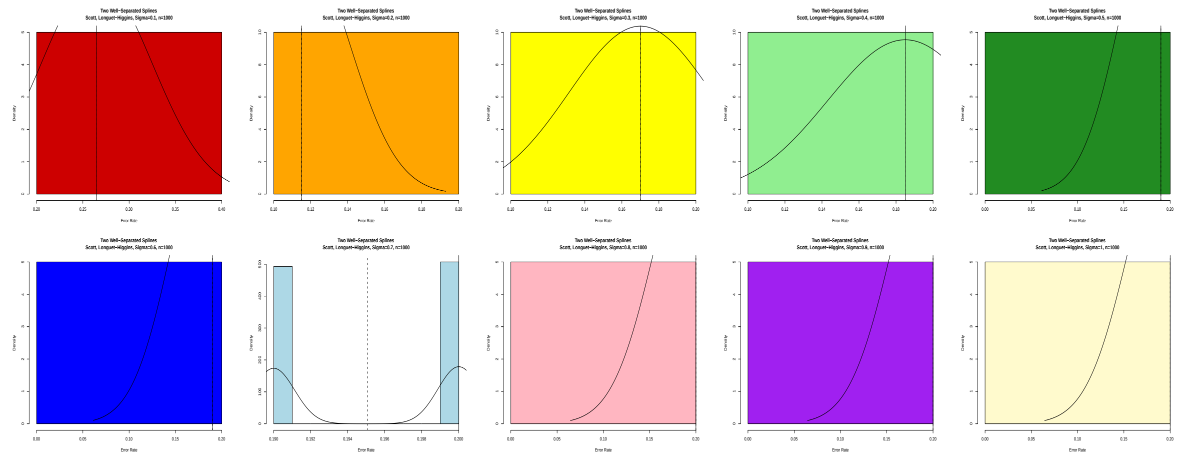


Figure 27: Scott/Longuet-Higgins error rates for two well-separated splines, σ values from [0.10, 1.00]

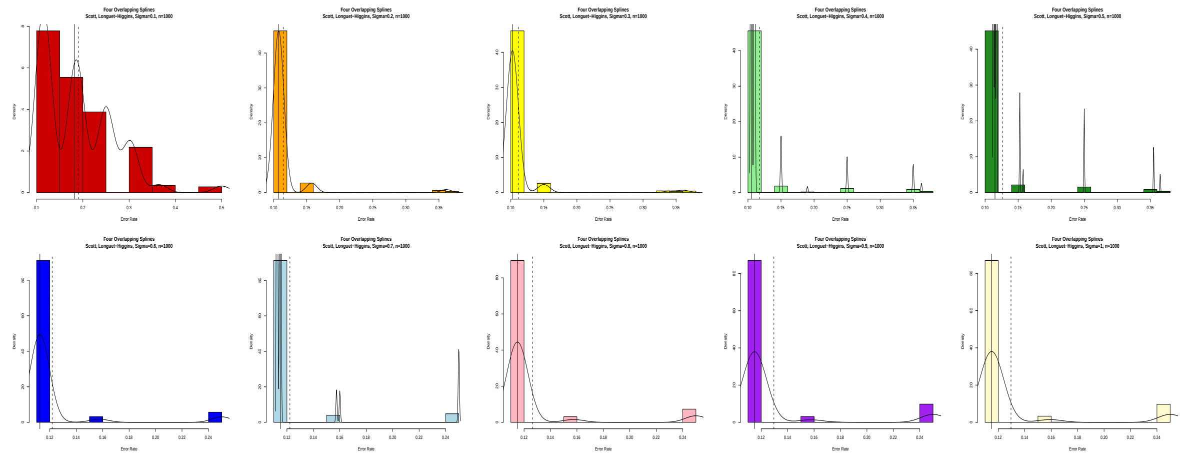


Figure 28: Scott/Longuet-Higgins error rates for four overlapping splines, σ values from [0.10, 1.00]

the error rates primarily fall within three main groups for values of $0.20 \leq \sigma \leq 1.00$: 0.10, 0.15, and 0.25. The vast majority of the error rates fall within the 0.10 group, and a comparably smaller proportion falls within the 0.15 and 0.25 groups. The mean error rate is approximately 0.125, whereas the median error rate is approximately 0.100.

Note that in Figure 29, all three algorithms appear to perform in almost indistinguishable fashions given different values of σ. We also see that across different σ values, the clustering solutions do not appear to change much. In addition to the similar misclassification rates noted above, we note that regardless of the dataset and algorithm, the clustering solution appears to become increasingly stable for σ values closer to 1. Thus, for the remainder of

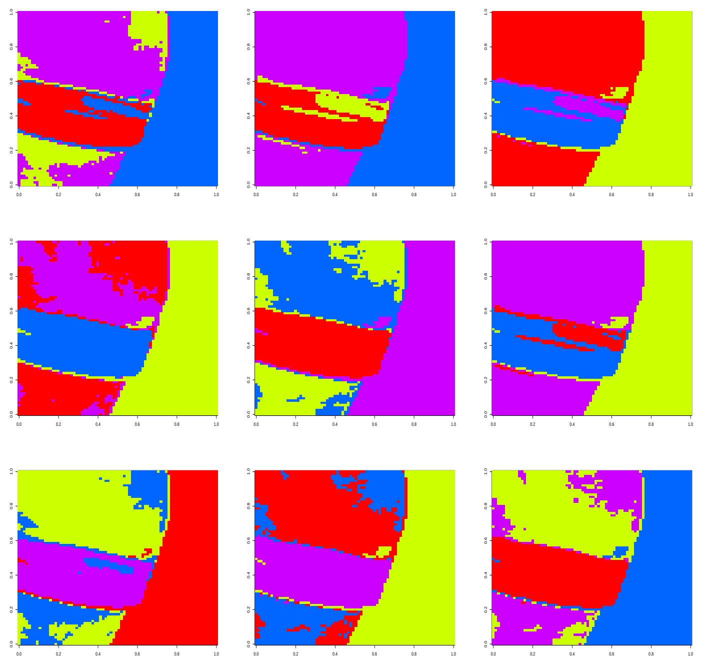


Figure 29: a., b., c. Perona Freeman; d., e., f. Ng/Jordan/Weiss; g., h., i. Scott/Longuet-Higgins; for all these rows, $\sigma = 0.25, 0.50, 0.75$

the analyses in this paper, the Perona/Freeman method of spectral clustering will solely be used (with $\sigma = 1.00$) since it requires the least amount of computation and still achieves almost exactly the same results as the other algorithms.

3 Image Segmentation: Choice of Features and Clusters

We now turn to segmenting images into groups of similar pixels. An image of size n by m has nm pixels, each of which can be measured by a set of features (or variables). We first explore the performance as related to several possible sets of variables.

3.1 Image Transformations

An image represented in RGB values breaks down into three components: red, green, and blue values. Each pixel in an image has three corresponding numbers that, when in correspondence with each other, create the one pixel color. It is easy to interpret the three individual RGB values, but not as practical in the real world sense of how humans visualize color. For that, we will consider the HSV representation.

The HSV image transformation breaks down into three components: hue, saturation, and value. Hue is what humans visualzie physical color to be in terms of light wavelength, saturation is the degree to which the hue appears within the color, and the value is the factor that measures brightness. Although the components of the a pixel's HSV value appear to be more foreign than the components of a pixel's RGB value, they make more sense in the physical world. We do not really see objects in terms of their red, green, and blue content separately, but instead by their hue, saturation, and value (Klein, 2011).

The HSV values are found by mapping the RGB values to cylindrical coordiantes. The reason we map to the conical representation is to give higher importance to the value parameter. We do so because as humans perceiving color, the value parameter matters the most. Notice that in the conical HSV diagram (Figure 30), as the value decreases, it is harder to distinguish between different hues and saturations. As humans have less light, it is harder to tell the difference between colors.

The sets of features/variables that we compare are listed below:

- RGB: Values are based upon the raw red, green, and blue content of each pixel in an image, and are bound between 0 and 1.
- RGBXY: Values are based upon the RGB pixel content, but also the physical x and y location of each pixel in an image. Values for RGB are bound between 0 and 1, whereas values for x and y are bound between 1 and the respective dimensions of the image.
- RGBXY Scaled: Values are based upon the RGBXY values; however, we divide the x and y location of each pixel by the maximum x and y dimensions of the image, respectively. All values are bound between 0 and 1.
- RGBXY All Scaled: Values are based upon the normal standardization of the RGBXY Scaled dataset and are bound between 0 and 1.
- HSV: Values are based upon the hue, saturation, and value content of each pixel in an image and are bound between 0 and 1. The hue corresponds to the dominant wavelength of the color we perceive, whereas the saturation corresponds to the dominance of hue in the color. Lastly, the value corresponds to how light or dark the color appears.
- HSVXY: Values are based upon the HSV pixel content, but also the physical x and y location of each pixel in an image. Values for HSV are bound between 0 and 1, whereas values for x and y are bound between 1 and the respective dimensions of the image.
- HSVXY Scaled: Values are based upon the HSVXY data; however, we divide the x and y location of each pixel by the maximum x and y dimensions of the image, respectively. All values are bound between 0 and 1.
- HSVXY All Scaled: Values are based upon the normal standardization of the HSVXY Scaled dataset and are bound between 0 and 1.

- HSVXY All Scaled Cone: Values are based upon mapping the HSVXY All Scaled dataset to a cone, and are bound between 0 and 1.

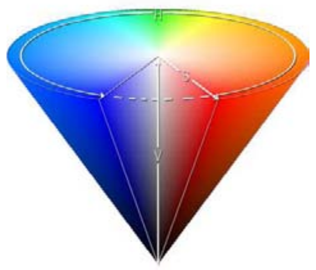


Figure 30: Conical HSV representation of color

As we are assessing the performance given different image representations, we keep both the distance measure and σ value within the calculation of the affinity matrix constant. Throughout all calculations, the distance measure was defined to be $\|x_i - x_j\|^2$, with a σ value of 1. We include solutions for a reasonable range of number of clusters, K = 2, 3, 4, and 5.

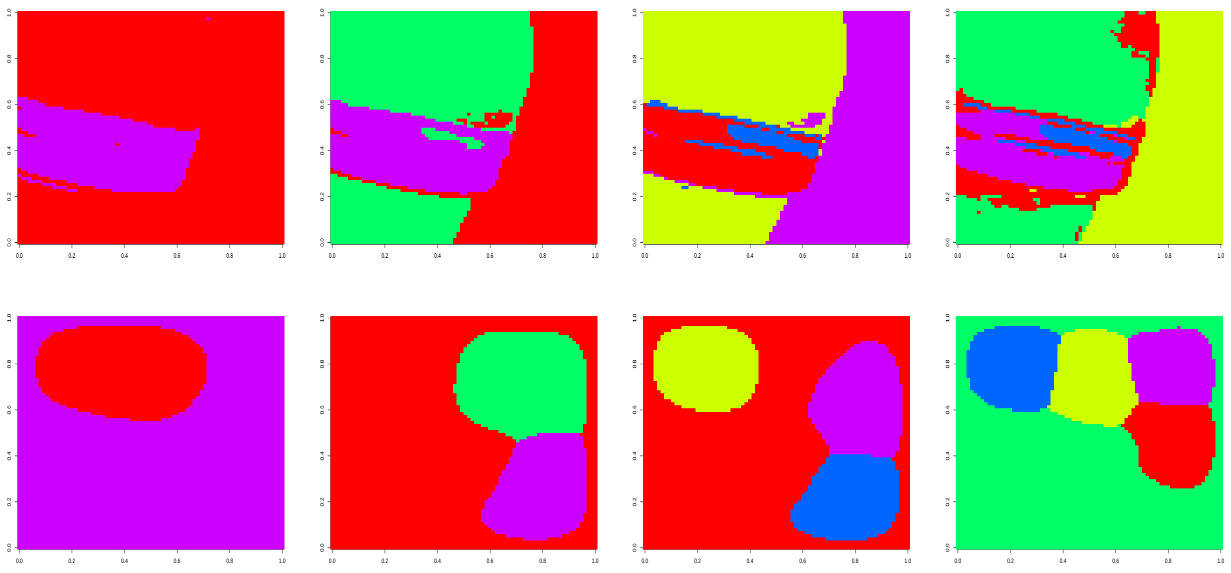


Figure 31: RGB segmentation, RGBXY segmentation; K=2, 3, 4, 5 for both

In Figure 31 we show the clustering solutions when using the RGB and RGBXY representation of the bracelet image. Using the RGB representation yields favorable solutions.

Clusters appear to be recovered based primarily upon their color (See Figure 2), which is what we would expect in this situation. As K increases, the solutions also reflect some type of shading within the image. On the other hand, the RGBXY representation does not appear suitable. The clustering solution primarily focuses on the physical location of the pixel, rather than the color content of the pixel. The returned clusters are somewhat nonsensical.

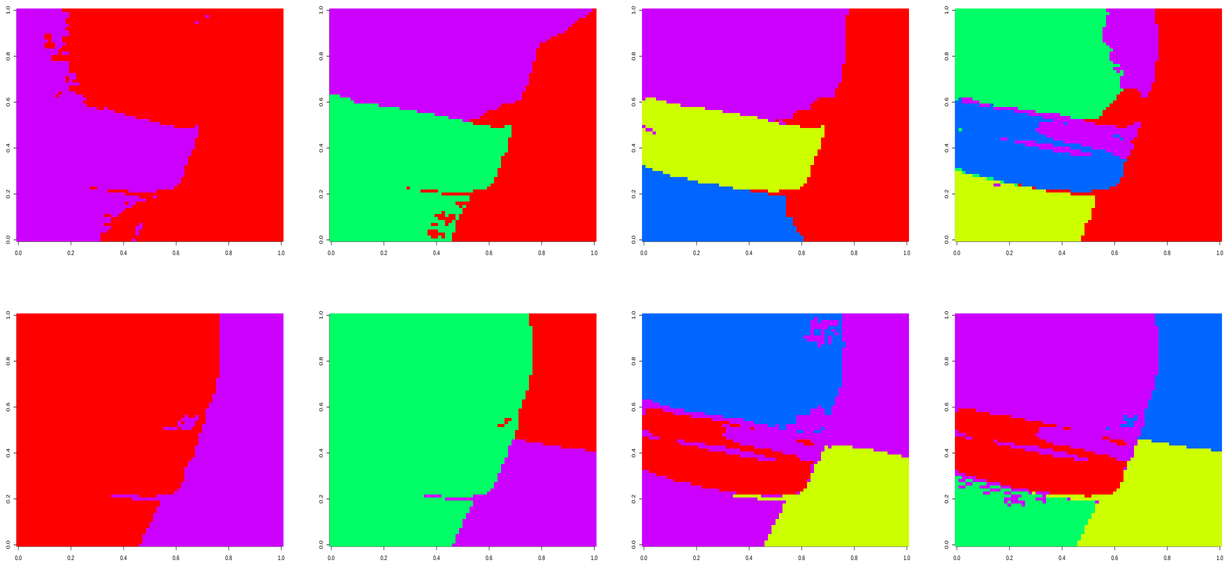


Figure 32: RGBXYscaled segmentation, RGBXYallscaled segmentation; K=2, 3, 4, 5 for both

Next we evaluate the RGBXYscaled and RGBXYallscaled versions of the picture (Figure 32). Although we begin to see some features of the original picture in the RGBXYscaled segmentations, the clusters do not always appear to make sense. Sometimes, the bracelet and wrist are incorrectly grouped together. Likewise, the wrist and background are sometimes grouped together. Therefore, clustering based upon this type of criteria may not be suitable either. As we inspect the RGBXYallscaled segmentations, we note that the bracelet image is recovered, but the clustering solutions are much more rigid about boundaries and do not pick up as many subtleties in shading in comparison to the solutions given with the RGB dataset. Depending on the desired basis for the boundaries, they might consider using this type of image data representation.

The HSV and HSVXY dataset clustering solutions are shown in Figure 33. We are able to recover the bracelet image with the HSV dataset just as before with the RGB; however, the clustering solutions seem to take more into account the gradual changes and shading within an image. When adding information pertaining to the x and y locations of the pixels for the HSVXY image data representation, the clustering solutions we see are very similar to the solutions of the RGBXY dataset. The physical pixel location information seems to outweigh the pixel color attributes and yields nonsensical results.

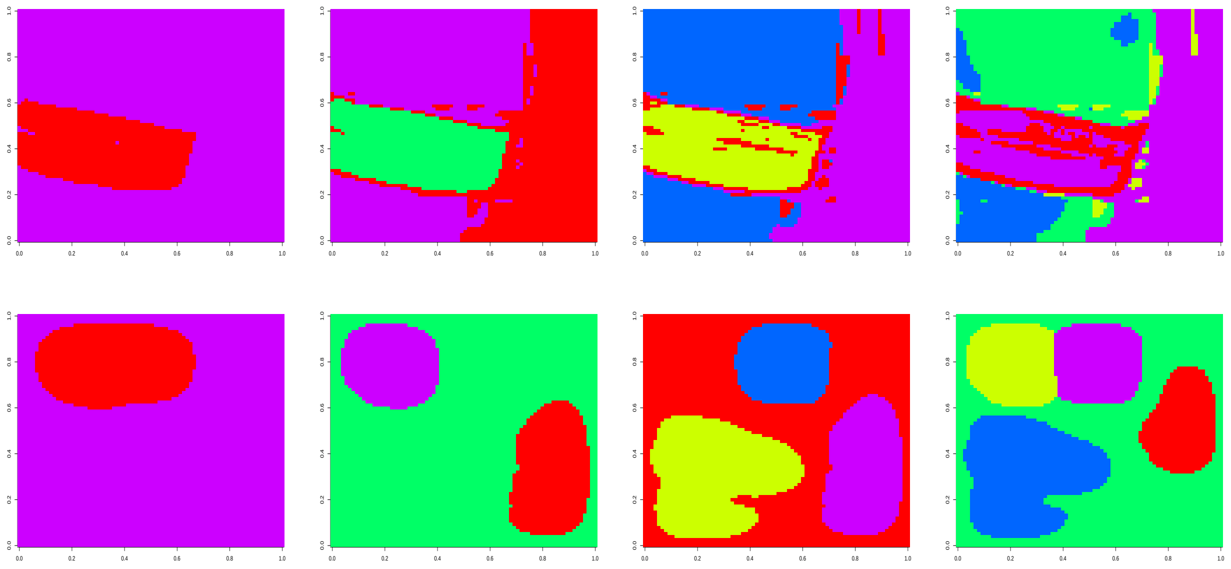


Figure 33: HSV segmentation, HSVXY segmentation; K=2, 3, 4, 5 for both

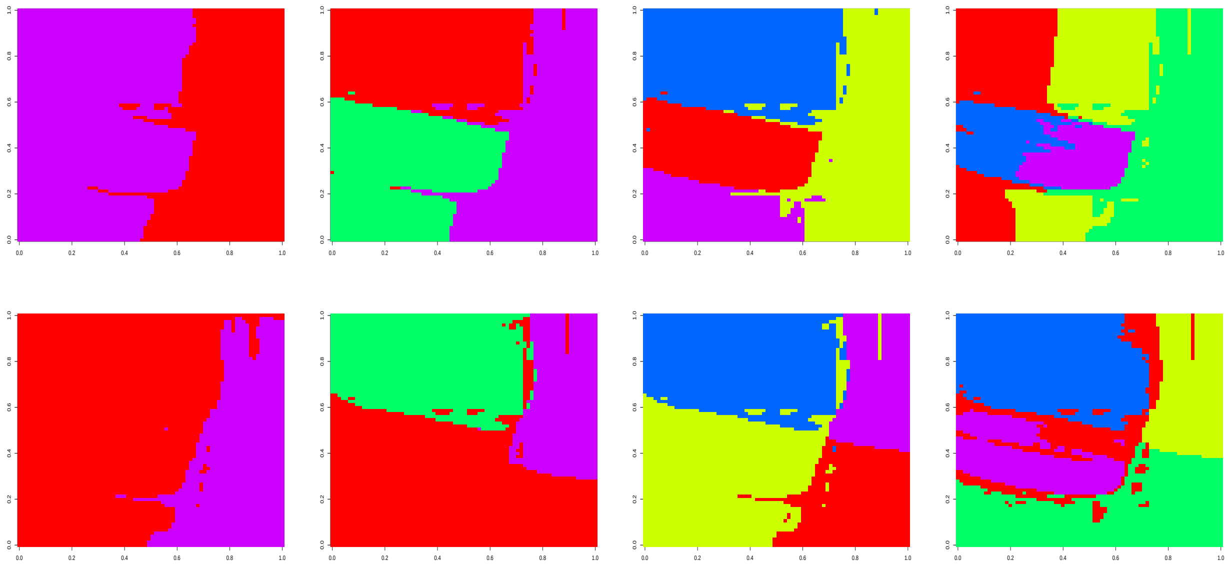


Figure 34: HSVXYscaled segmentation, HSVXYallscaled segmentation; K=2, 3, 4, 5 for both

Segmentation solutions for the HSVXYscaled and HSVXYallscaled datasets are shown in Figure 34. As before, the clustering solutions for the HSVXY dataset are a bit more rigid and take less account of shading than the RGB and HSV dataset solutions, which can be an advantage depending on the user's segmentation goals; however, true clusters seem to be split because of the pixel locations. For example, the upper half of the wrist and the lower half of the wrist are segmented into two distinct clusters when they probably should not be. Likewise, the bracelet is sometimes split into left and right halves. The outlines of the main clusters in the HSVXYallscaled image seem to be recovered slightly, yet the solutions in general do not appear useful in our context. The background and bracelet are sometimes clustered with the wrist, which is not accurate.

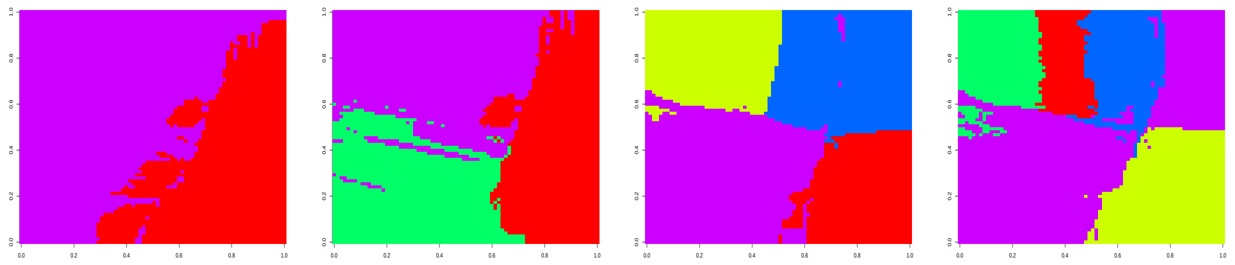


Figure 35: HSVXYallscaledcone clustering solution, K=2, 3, 4, 5

Lastly, we inspect the clustering solutions of the bracelet image under the HSVXYallscaled-cone values (Figure 35). Unfortunately, the main image clusters are not recovered. Therefore, the solutions are not useful in our context.

The RGB and HSV datasets appear to recover the image most completely, where RGB tends to focus a bit more on straight color values and HSV on shading aspects of the image. Both of these sets of variables could potentially be useful depending on the type of segmentation the user desires to implement. We note that both the RGBXY and HSVXY appear to be entirely overpowered by the x and y variables, effectively reducing the cluster assignments to projections of spheres in two dimensional space and disregarding any RGB or HSV values. Therefore, we believe those two will not be useful. The RGBXY Scaled, HSVXY Scaled, RGBXY All Scaled, HSVXY All Scaled, and HSVXY All Scaled Cone datasets are also very heavily influenced by x and y location, and do not yield accurate clustering solutions; however, there is some resemblance to the original image. For general segmentation, the inclusion of the x and y locations do not appear to benefit our analysis. Therefore, for the remainder of our analysis we choose to use only the RGB and HSV dataset representations of our images.

3.2 Choosing the Number of Clusters, K

As we have seen in the previous section, the K-means solutions are heavily dependent upon how many clusters we choose. For a very simple picture, one may be able to visually make an educated guess as to an estimate of what the true number of clusters should be; however, for more complex images, this is not a practical approach.

One criteria which we consider when attempting to choose the optimal K for a given dataset is the minimization of the within cluster sum of squared distance values. Of course, if the clusters are more condensed/similar, the within cluster sum of squared distance values should be minimized. We create the scree plots in Figure 36 below. The x-axis corresponds to K, whereas the y-axis corresponds to the within groups cum of squares.

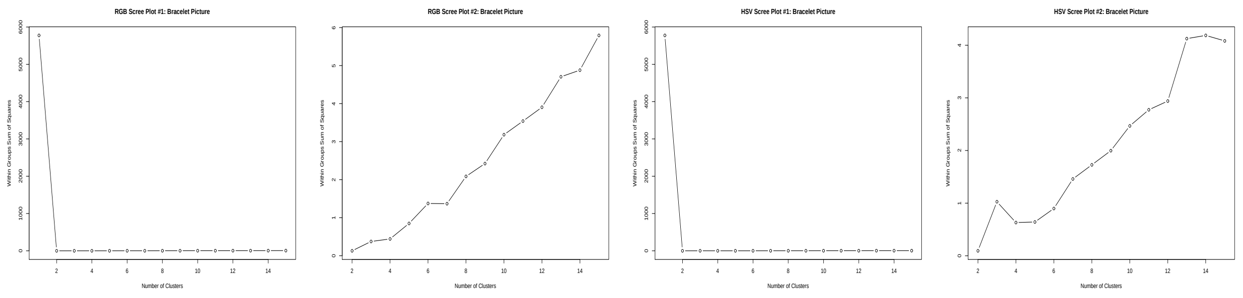


Figure 36: a., b. RGB scree plots, c., d., HSV scree plots

We note that within each plot there is a radical drop of the within cluster sum of squared Euclidean distances from K=1 cluster to K=2 clusters. The scale of the plots is greatly skewed, and nearly entirely masks the remaining information. Therefore, we include a second scree plot for each dataset that begins with K=2 cluster values.

Within each scree plot, there appears to be a slight increase of the within cluster sum of squared distances as K increases which may be indication that there could be a strong dependency of the segmentation on the initial starting values. There does not appear to be an area in which a low and stable sum of squared distances exists. On the other hand, we may not have inspected values of K enough, and should take into account that the jumps from $K \geq 2$ are very small considering the original scale. Unfortunately, we may not be able to determine an optimal K from a widely accepted and generally used method of simulation. We will have to explore other ways in which we can determine the best number of clusters to request from our K-means solutions given an image (see section 6). For the rest of our analyses, we will continue considering a suitable range of K from 2 to 5.

3.3 Image Segmentation with Removed Sections

Because we will ultimately attempt to impute cluster labels for missing pixels, we turn to assessing the performance of our image segmentation in the presence of missing pixels. Once again, we will analyze the wrist/bracelet picture using RGB and HSV and remove different sections of the image. The following methods of inducing missingness were explored:

- The removal of an "easy" rectangular box of pixels, where we expect that all missing pixels should be in the same cluster as each other and as the nearby pixels (i.e. the removal of adjacent pixels with approximately the same color among other pixels with approximately the same color).

- The removal of a "hard" rectangular box of pixels, where we expect that the missing pixels should not all be in the same cluster as each other, and nearby pixels are not necessarily in matching clusters (i.e. the removal of pixels with different colors and the border between the two groups).

- The removal of multiple boxes, some "easy" and some "hard."

- The removal of a random subset of all pixels.

After removing the pixels, the affinity matrix was found by using once again using the squared Euclidean distance with the σ value set at 1. The first K leading eigenvectors were then extracted, and K-means solutions with 2 to 5 clusters were estimated.

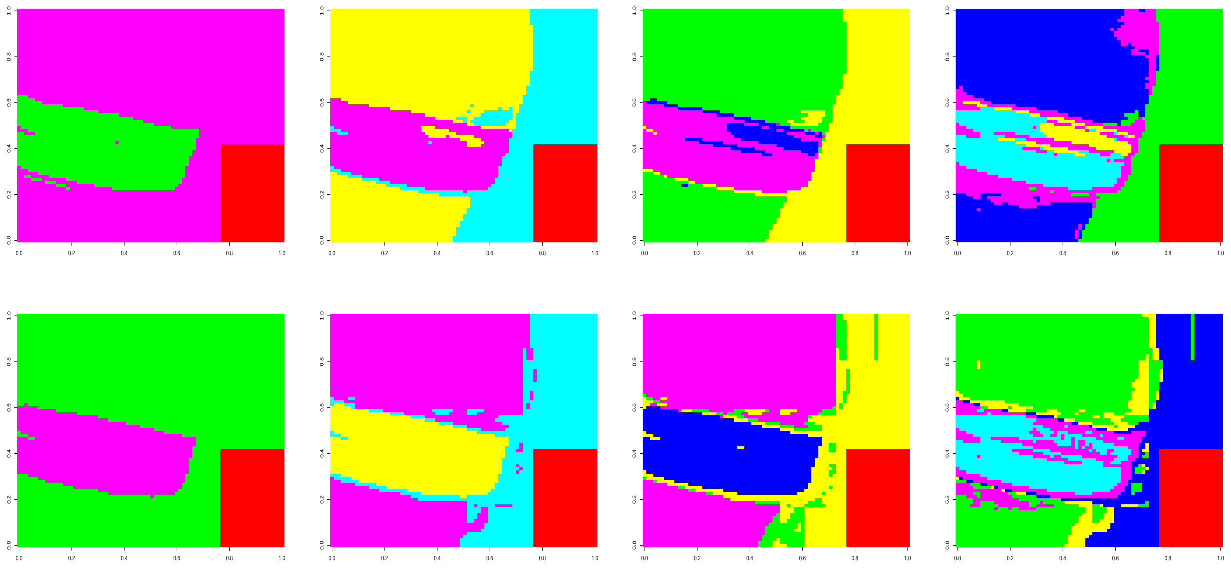


Figure 37: "Easy" box removal, a. - d. RGB, e. - h. HSV; both using K=2, 3, 4, 5

The removal of an "easy" box of pixels (Figure 37) served as a diagnostic for the sensitivity of our segmentation method. In all cases, other than the variability that comes along with the non-deterministic K-means solutions, this type of removal of pixels did not affect the final segmentations.

The removal of a "hard" box of pixels (Figure 38) was a bit more interesting. In most cases, this type of missingness did not affect the K-means solutions either; however, in some cases if the removal spanned primarily over a border region, sometimes the distinctions between groups would be lost. We expect this since most of the information lost has to do with the border which separates the groups in the first place.

Holding everything else constant, removing multiple boxes did not seem to have any effect on the outcome of the K-means solutions (Figure 39). The individual boxes appeared to

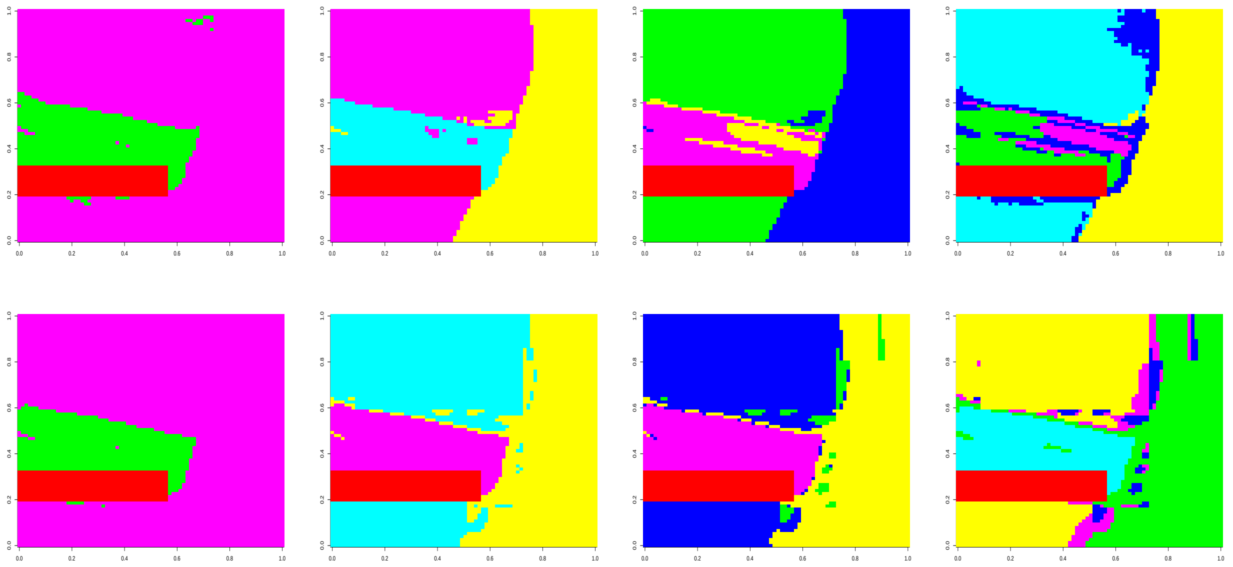


Figure 38: “Hard” box removal, a. - d. RGB, e. - h. HSV; both using K=2, 3, 4, and 5

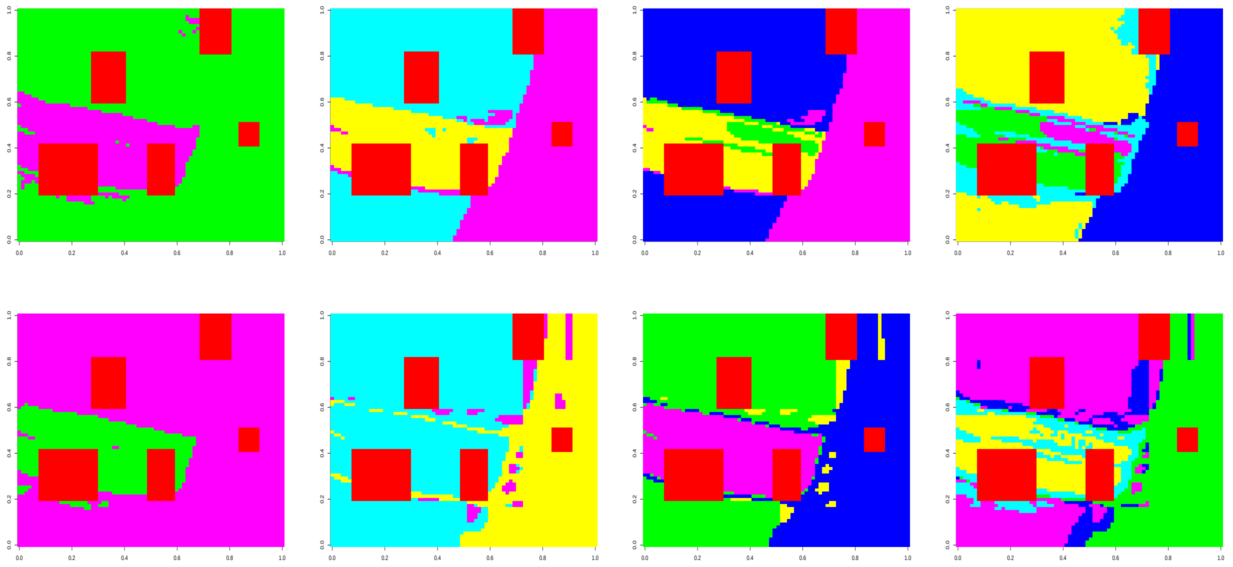


Figure 39: Many box removal, a. - d. RGB, e. - h. HSV; both using K=2, 3, 4, and 5

change the clustering solutions (if at all) as if the other missing boxes were not present. Therefore, we believe that different adjacent groups of missing pixels may be independent of one another when trying to cluster the remaining pixels.

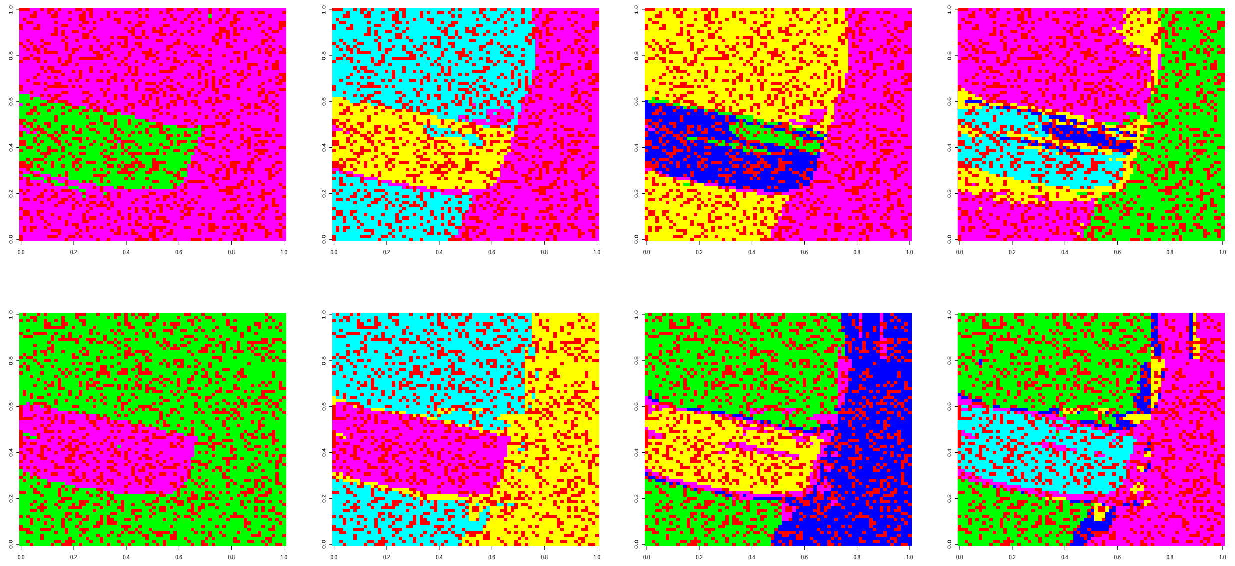


Figure 40: Random pixel removal, a. - d. RGB, e. - h. HSV; both using K=2, 3, 4, and 5

Finally, when removing a random subset of all pixels, the K-means solution did not seem to be affected (Figure 40).

4 Imputation Approaches

When pixels are close to each other, we believe that they have a higher chance of having similar attributes. Therefore, when imputing the cluster label of a missing pixel, we would like to primarily use the information of the surrounding pixels that are not missing; however, we also believe that pixels that are not directly adjacent to missing pixels might also contain relevant information for imputing the missing cluster labels. Therefore, we also will impute missing pixels by weighting available information: closer pixels will be upweighted (and thus influence the imputed cluster label more), whereas farther pixels will be downweighted (and thus influence the imputed cluster labels less).

4.1 Minimum Manhattan Distance Imputation Method

The first method of imputation considers only those non-missing adjacent pixels that are the minimum Manhattan distance away from the missing pixel. In contrast to Euclidean distance, the Manhattan distance between two points is defined to be the sum of the absolute differences of each respective coordinate. Suppose $x_1 = (x_{11}, x_{12}, x_{13}, ..., x_{1D})$ and $x_2 = (x_{21}, x_{22}, x_{23}, ..., x_{2D})$, then the Manhattan distance between those two points is calculated as:

$$Manhattan(x_1, x_2) = \sum_{d=1}^{D} |x_{1d} - x_{2d}|$$

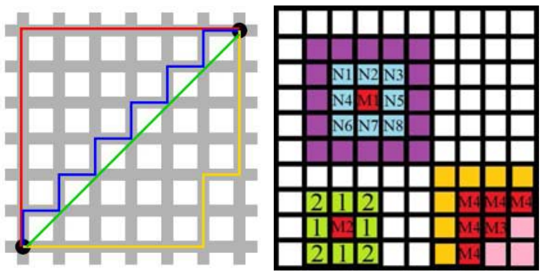


Figure 41: a. Manhattan distance vs. Euclidean distance, b. Nearest neighbors and missingness

Figure 41a depicts a graphical comparison between Euclidean distance (green) and Manhattan distance (red, yellow, and blue). Euclidean distance is often referred to as the "shortest distance" between two points (a simple straight line), whereas the Manhattan distance is the "city block" distance.

First, the minimum Manhattan distance imputation method finds all nearest neighbors of a missing pixel. In Figure 41b, the red pixel M1 represents a missing pixel, and the eight light blue pixels directly adjacent to M1 (including the four corners) are called the nearest neighbors of M1. Note that pixels on an edge or in a corner will have fewer adjacent neighbors. The Manhattan distances for each of the nearest neighbors is calculated. An example calculation of the Manhattan distance can be seen by the green pixels surrounding red pixel M2. Note that the Manhattan distances are either 1 or 2. During imputation, only non-missing adjacent pixels are considered. If there is only a single pixel with the minimum distance, then its cluster label is imputed for the missing pixel; however, if there are multiple adjacent, non-missing pixels with the same minimum distance from the missing pixel, we choose one by taking a random draw from a uniformly weighted multinomial distribution and assign the cluster label of the randomly selected neighbor to the missing pixel. Once assigned, the imputed label is then treated as the other cluster labels in future imputations as if it were a true value. The imputed label can now participate in imputing labels for other missing observations.

As an example, consider a missing pixel such as missing pixel M3. Using the minimum Manhattan distance method, the other surrounding missing pixels labeled M4 cannot contribute any information towards imputing M3. Only two of the three pink pixels have a chance of having their cluster label imputed for missing pixel M3. Since the pixels east and south of M3 both have a Manhattan distance of 1, they both have a $\frac{1}{2}$ chance of having their cluster label imputed, whereas the pixel south-east of M3 has a Manhattan distance of 2 and is not considered.

If a missing pixel is completely surrounded by missing pixels, then the algorithm temporarily skips that pixel and moves on to another missing pixel. The algorithm continues to iterate through all missing pixels, treating any imputed labels as cluster labels, until there are no more missing pixels left within the image (as eventually the previously skipped pixels will have at least one label imputed for one of its nearest neighbors). Note that this type of algorithm will work radially inward through large chunks of missingness, and have a high dependency on those few outside neighboring pixels.

4.2 Weighted Nearest Neighbors Imputation Method

The second method of imputation considers all pixels within a chosen distance of a missing pixel. We define these neighbors in terms of "rings." In Figure 41b, the first ring of nearest neighbors of pixel M1 is highlighted in light blue. The second ring of nearest neighbors for pixel M1 is highlighted in purple. Neighbors of a missing pixel in the first ring consist of the eight pixels that are directly adjacent to the missing pixel (including the four corners). The neighbors of a missing pixel in the second ring consist of those sixteen pixels that are directly adjacent to the first ring pixels (including the four corners) that are not members of the first rings. Again, notice that pixels on edges or corners will have incomplete rings. In general, the number of pixels in ring r alone is given by $8r$ (see section 4.2.1). Furthermore, given a number of rings r and a central pixel, we can determine which pixels comprise of the total set of $4r(r+1)$ (see section 4.2.2) nearest neighbors.

Within the weighted nearest neighbors imputation method, we assign weights to the different rings surrounding a missing observation. Using this imputation method, all available pixels within r rings have a chance to participate in the imputation of a missing pixel, rather than just those that correspond to the minimum Manhattan distance. As mentioned before, we assume that pixels closer by a missing pixel are probably more similar than pixels that are further away; however, that does not mean that pixels further away contain no imputation information. Therefore, we can weight the non-missing neighbor in a chosen number of surrounding rings such that closer rings have higher weights and farther rings have lower weights. We then can use these weights in multinomial draw in order to determine which cluster label to impute for the missing pixel.

Take for example the case in which we consider only the first three rings of nearest neighbors for a missing pixel. Suppose the first ring has a available observations, the second has b, and the third has c. There are $(a+b+c)$ pixels overall that have a possibility of having their cluster labels being chosen, so initially we assign each pixel a weight of $\frac{1}{(a+b+c)}$; however, we still must take into account the ring location of the pixels. Given weights w_1, w_2, and w_3 for the three rings, we calculate the final ring weights as follows:

$$\text{Ring 1 Weight: } \frac{1}{(a+b+c)} * \frac{w_1}{\frac{a}{(a+b+c)}}$$

$$\text{Ring 2 Weight: } \frac{1}{(a+b+c)} * \frac{w_2}{\frac{b}{(a+b+c)}}$$

$$\text{Ring 3 Weight: } \frac{1}{(a+b+c)} * \frac{w_3}{\frac{c}{(a+b+c)}}$$

In general, suppose there are r_{all} non-missing pixels within rings 1 though r, of which there are r_r non-missing pixels in ring r alone. Then the general formula for ring r weight is as follows:

$$\text{Ring } r \text{ Weight: } \frac{1}{r_{all}} * \frac{w_r}{\frac{r_r}{r_{all}}}$$

Note that by calculating ring weights in this manner, we are scaling the uniform overall pixel probabilities of being chosen by the specific ring weight w_r. Our multinomial probabilities also reduce to $\frac{w_r}{r_r}$ in this case. If we started with non-uniform weights for the pixels, e.g., then our rescaled weights would be $p_{ir} * \frac{w_r}{\frac{r_r}{r_{all}}}$, where p_{ir} is pixel i in ring r.

4.2.1 Determining the Number of Pixels in Ring r

Claim: The number of pixels surrounding an arbitrary center pixel in a ring r is equal to $8r$.

Proof:

- Count the 4 corner pixels
- Count the 4 pixels directly adjacent to the center pixel (2 vertically and 2 horizontally)
- Split the remaining image into 4 smaller squares of size r^2 (we know this is the size of the smaller sub-squares because we extend outwards from the center pixel by exactly r pixels)
  - Count $(r-1)$ pixels for each section between the corner pixels and the pixels directly adjacent to the center pixels. There will be 2 sets (1 vertically and 1 horizontally) in each of the 4 subsquares.
- Calculate: (4 corners) + (4 directly adjacent) + (4 subsquares)[(2 directions)$(r-1)$]
- $4 + 4 + 4[2(r-1)]$
- $8 + 8(r-1)$
- $8 + 8r - 8$
- $8r$

Q.e.d.

4.2.2 Determining the Total Number of Pixels in Rings 1, 2, ..., r

Claim: The number of pixels surrounding an arbitrary center pixel up through and including a ring r is equal to $4r(r+1)$.

Proof: By Induction on r

- Base Case:
  - For $r = 1$, we know the number of surrounding pixels for the first ring is $8r = 8(1) = 8$. Note that $4r(r+1) = 4(1)(2) = 8$
- Inductive Hypothesis
  - Suppose that for some arbitrary $k \geq 1$ that the number of cumulative pixels in the surrounding k rings of a central pixel is equal to $4k(k+1)$. We need to show that the number of cumulative pixels in the surrounding $k+1$ rings of the same central pixel is equal to $4(k+1)(k+2)$
- Inductive Step
  - $8(1) + 8(2) + ... + 8(k-1) + 8(k) = 4k(k+1)$
  - $8(1) + 8(2) + ... + 8(k-1) + 8(k) + 8(k+1) = 4k(k+1) + 8(k+1)$
  - $8(1) + 8(2) + ... + 8(k) + 8(k+1) = 4k^2 + 4k + 8k + 8$
  - $8(1) + 8(2) + ... + 8(k) + 8(k+1) = 4k^2 + 12k + 8$
  - $8(1) + 8(2) + ... + 8(k) + 8(k+1) = 4(k^2 + 3k + 2)$
  - $8(1) + 8(2) + ... + 8(k) + 8(k+1) = 4(k+1)(k+2)$

Q.e.d.

4.3 Results

We will analyze how these imputation approaches perform when attempting to segment the bracelet picture with induced missingness. We will use both RGB and HSV formats of the image data, and the range of K=2, 3, 4, and 5. The choice of the optimal number of rings is an open question whose answer can change depending on the type of image or missingness. For our analyses, we choose the number of rings r=5 as a reasonable choice for demonstration purposes. Likewise, we will choose the ring weights for the first through fifth rings to be $\frac{5}{15}$, $\frac{4}{15}$, $\frac{3}{15}$, $\frac{2}{15}$, $\frac{1}{15}$, respectively. By choosing these weights, we give complete pixels closer to the missing observations a higher probability of having their cluster label imputed.

We first compare the performances of the minimum Manhattan distance algorithm upon the "easy" missing box of pixels (Figure 42). In both the RGB and HSV representations among all cases of K, the images are restored and segmented very accurately. The minimum Manhattan method did not have any difficulty in placing all of the missing pixels in the same

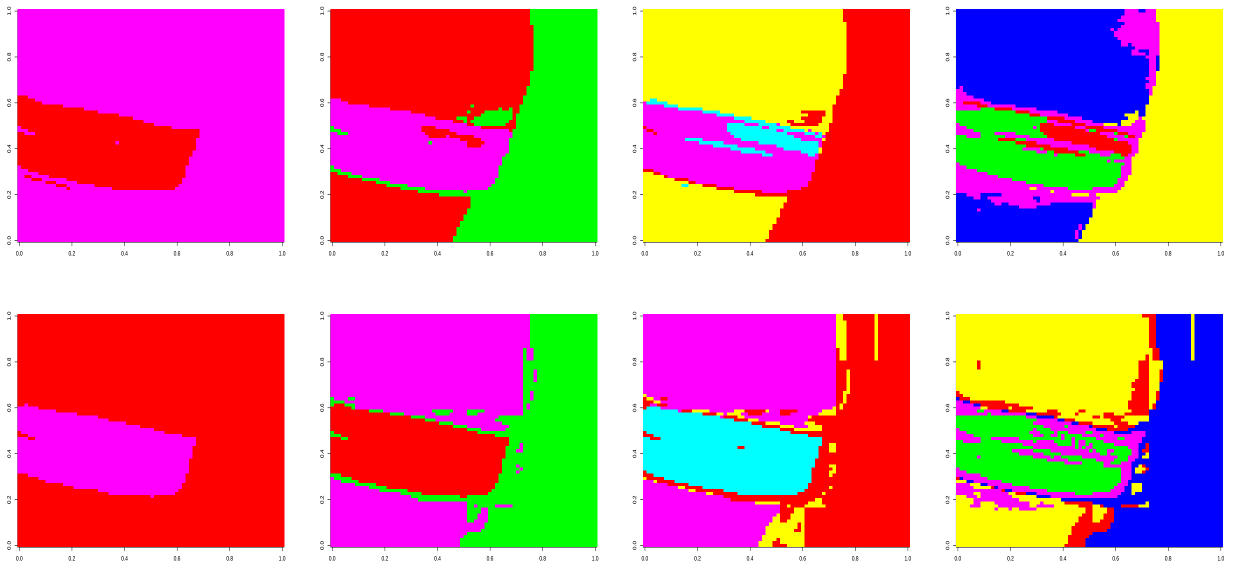


Figure 42: “Easy” box missingness imputation, minimum Manhattan method; a. - d. RGB values, e. - h. HSV values; both using K=2, 3, 4, and 5

group that corresponds to the sky, which we expected since the surrounding pixels are all of the same cluster.

We next try to impute over a more complex area of the image using the same method (Figure 43). In all solutions, it appears as if most of the missing observations are being segmented into the bracelet group, whereas some should truly be in the wrist group. These types of errors probably occur because the removed pixels are directly over a distinct cluster border between the wrist and bracelet. Therefore, although the algorithm can accurately determine that the missing pixels may belong to either the bracelet or wrist segment, there is some uncertainty as to which one out of the two segments they belong.

Next we inspect how the minimum Manhattan distance algorithm performs when attempting to impute values when there are many distinct areas of an image missing (Figure 44). Many of the missing boxes are recovered; however, there is one problematic box that extends across from the wrist to the bracelet. In many of the segmented solutions, the algorithm tended to either extend the bracelet cluster down into the wrist area, or extend the wrist cluster up into the bracelet area. Once again, these errors are somewhat expected since the nearby border information is not available.

We move on to explore how the weighted nearest ring neighbors method acts when imputing with the same images ($r = 5$). The first type of missingness considered is the “easy” box region of pixels (Figure 45). We expect all pixels removed to truly be part of the sky segment. Once again, the algorithm is able to recover each missing pixel's segment label quite well.

Next we want to see what may happen to the imputed segment assignments if the missing

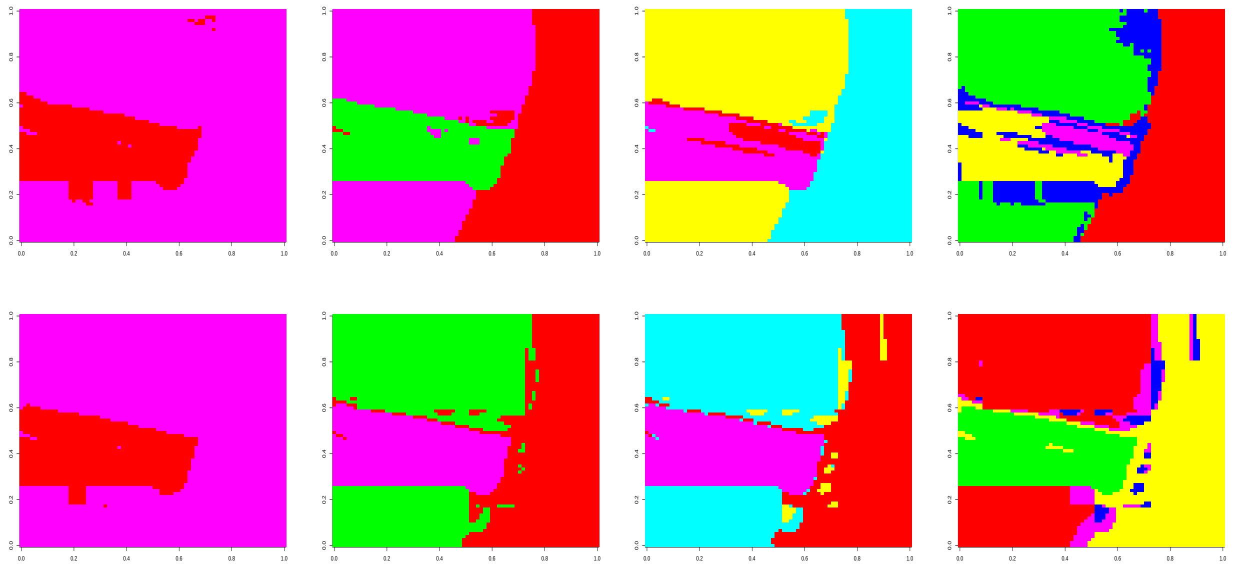


Figure 43: “Hard” box missingness imputation, minimum Manhattan method; a. - d. RGB values, e. - h. HSV values; both using K=2, 3, 4, and 5

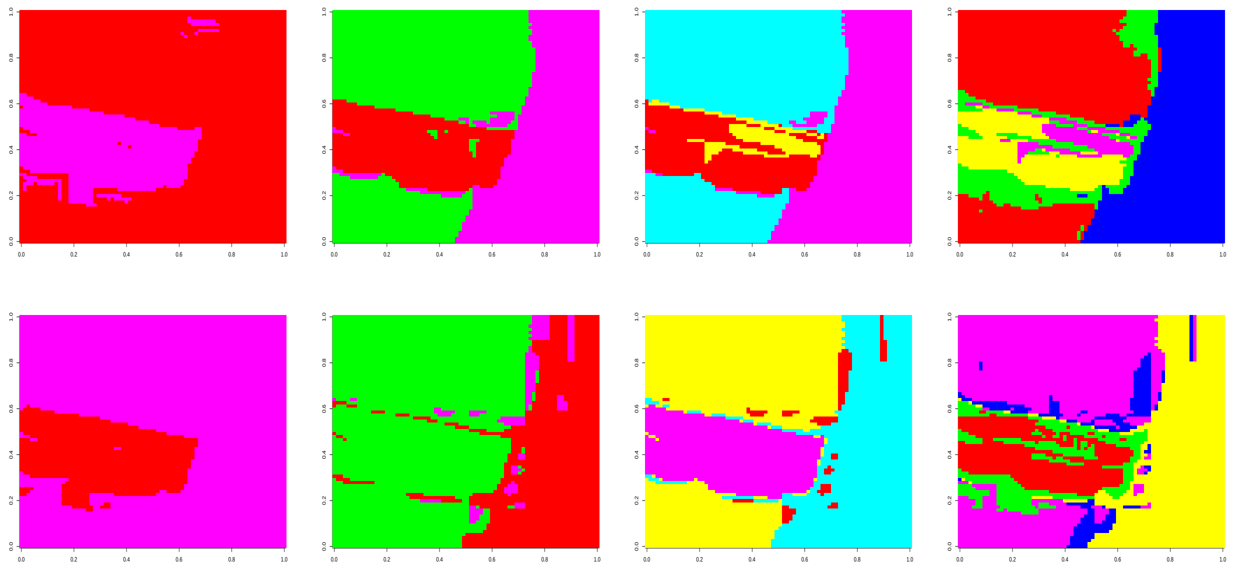


Figure 44: Many separate box missingness imputation, minimum Manhattan method; a. - d. RGB values, e. - h. HSV values; both using K=2, 3, 4, and 5

pixels cross a distinct border (Figure 46). The same type of errors as seen when applying the minimum Manhattan distance method also appear here. In all cases, the bracelet segment

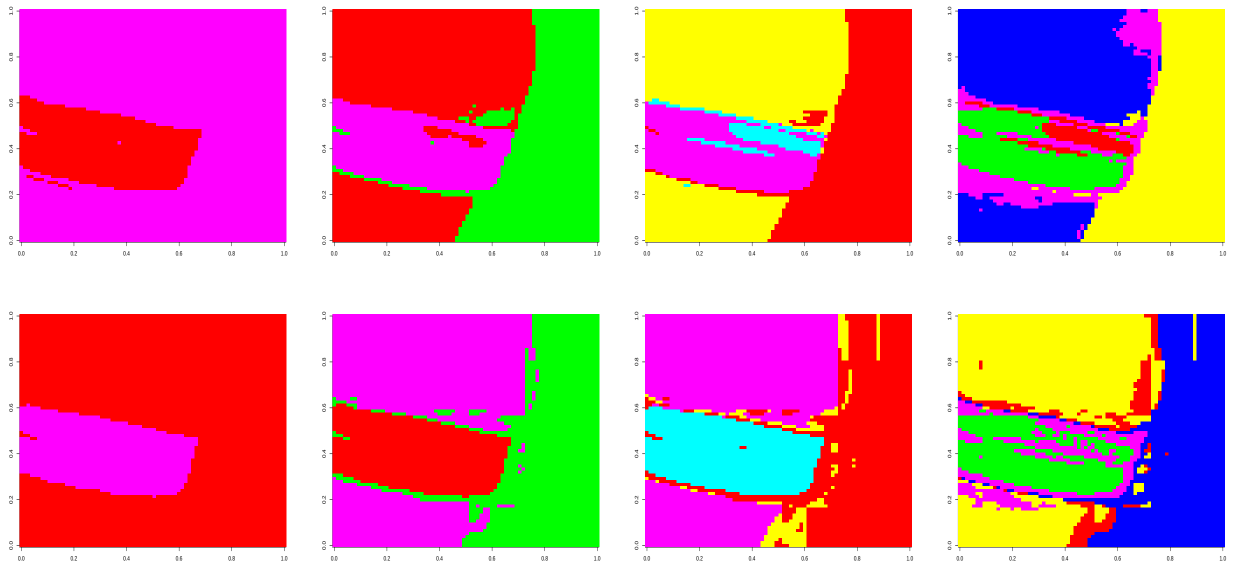


Figure 45: “Easy” box missingness imputation, weighted nearest neighbors method; a. - d. RGB values, e. - h. HSV values; both using K=2, 3, 4, and 5

was merely extended towards the wrist segment. This is problematic, since we know that a handful of observations should have been segmented into the wrist cluster instead.

Lastly, we implement the weighted nearest neighbors method to impute cluster labels for multiple missing boxes (Figure 47). The performance is very good, and nearly all of the missing observations are given imputed values that make sense. We see that there is sometimes a bit of uncertainty, especially where the wrist and bracelet meet; however, the borders are generally recovered.

In general, we find that the solutions of the imputation methods do not depend very much on the choice of data representation (RGB or HSV values) or the choice of K. Instead, the solutions seem to be greatly influenced by the location of the missing values. If the missingness is located within a region that does not contain much pixel variability, both the minimum Manhattan and weighted nearest neighbors methods of imputation perform very well. On the other hand, if the missingness is located in an area of the image where pixel variability is high (such as near a border of two distinct segments), both methods tend to make more errors. If there are smaller, non-dense areas of missing pixels, I would suggest using the minimum Manhattan method of imputation, since the imputed labels will not as easily be convoluted. On the other hand, if there are larger, more dense areas of missing pixels, I would suggest using the weighted nearest neighbors method of imputation, since the imputed labels will take into account a wider span of available pixel information.

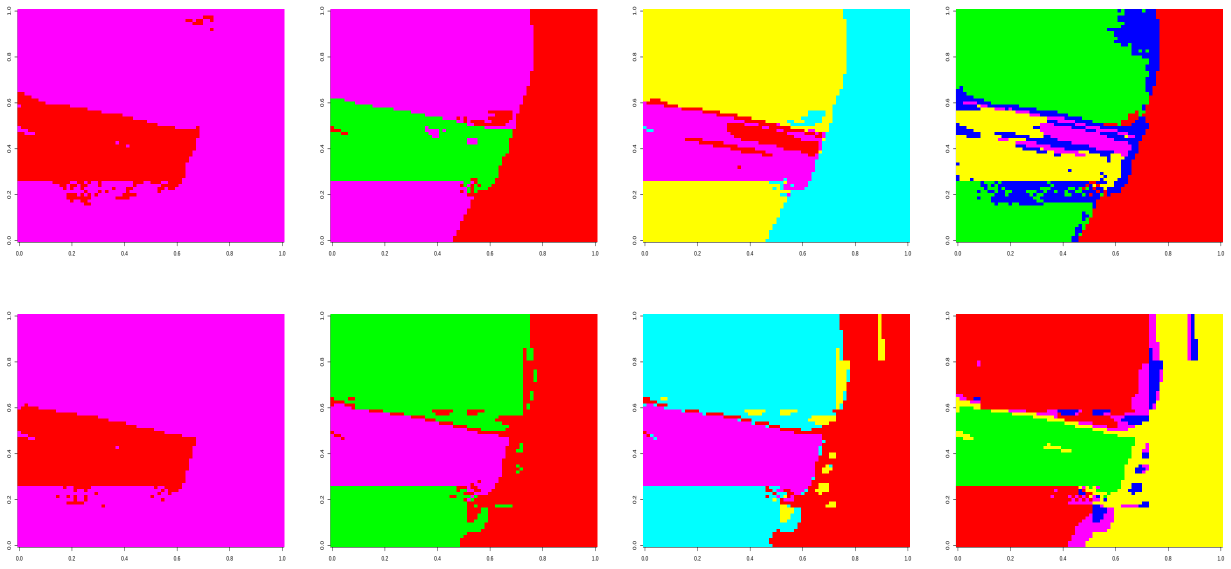


Figure 46: “Hard” box missingness imputation, weighted nearest neighbors method; a. - d. RGB values, e. - h. HSV values; both using K=2, 3, 4, and 5

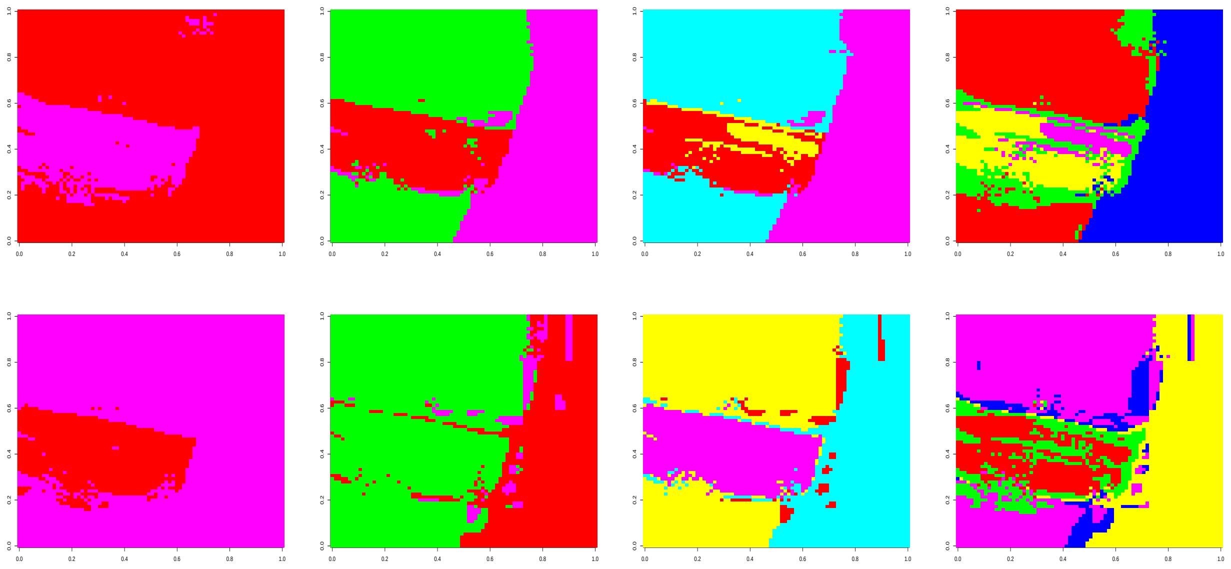


Figure 47: Many separate box missingness imputation, weighted nearest neighbors method; a. - d. RGB values, e. - h. HSV values; both using K=2, 3, 4, and 5

5 Experiments with Images

In this section we attempt to segment various types of images and impute missing cluster values. We will continue exploring the advantages and disadvantages of using the minimum

Manhattan and weighted nearest neighbors imputation methods; however, since we have found that the solution does not depend so much on the choice of data representation, we choose to solely use RGB values for the remainder of our analyses.

We begin by attempting to segment an image with a simple pattern: a checkerboard (Figure 48). Note that the true number of clusters is K=2. When trying to cluster the data into 3 or more groups, the K-means algorithm runs into trouble, as it must somehow allocate the extra groups. The solutions for $K \geq 3$ tend to have some random segmentation artifacts that only worsen as we try to impute. A flawless solution is seen when using the minimum Manhattan distance method after clustering with K=2. For the checkerboard image, the weighted nearest neighbors method tends to make many more mistakes than the minimum Manhattan distance method.

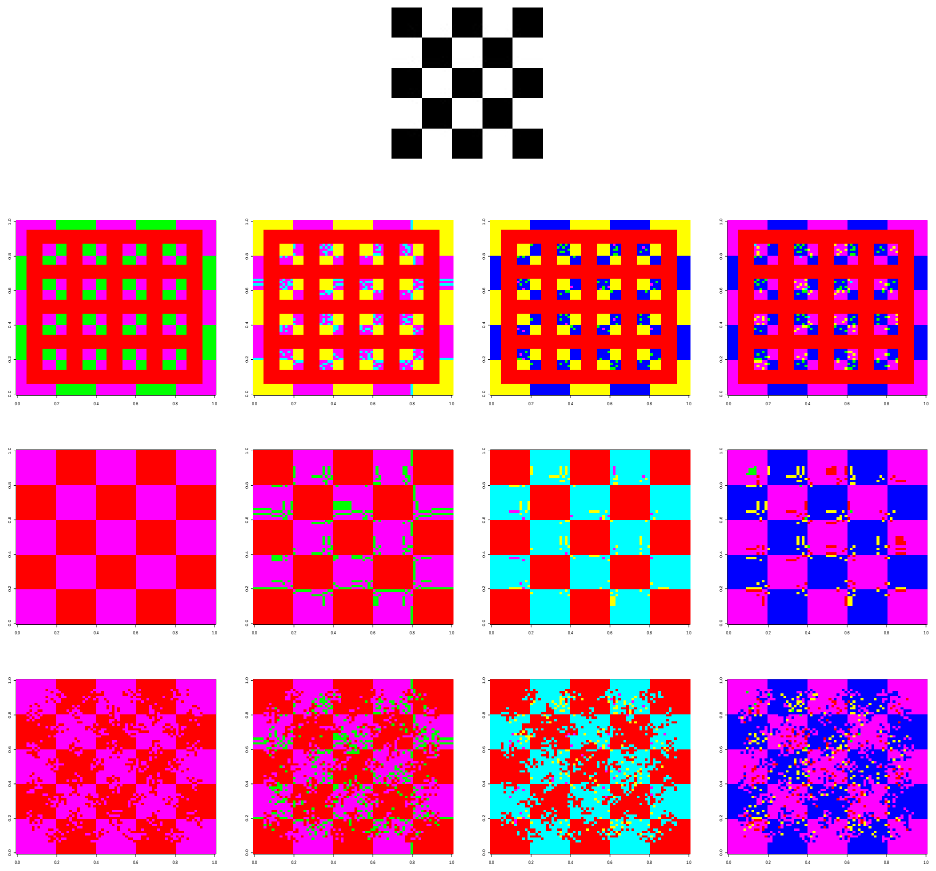


Figure 48: a. Checkerboard, b. - e. K-means solutions for K=2, 3, 4, and 5, f. - i. Minimum Manhattan imputation, j. - m. Weighted nearest neighbors imputation

The next image we run our algorithms upon is a similar checkerboard, but this time containing various colors (Figure 49). The K-means solutions tend to identify the main groups, yet seem to split the black border into two groups before making a distinction between the yellow and green squares. This is most likely due to some artifacts left in the image after converting it to a lower quality format. It is interesting to note that, given a particular K-means segentation solution, all of the minimum Manhattan imputations performed extremely well. Comparatively many more errors were made when using the weighted nearest neighbors imputation method once again.

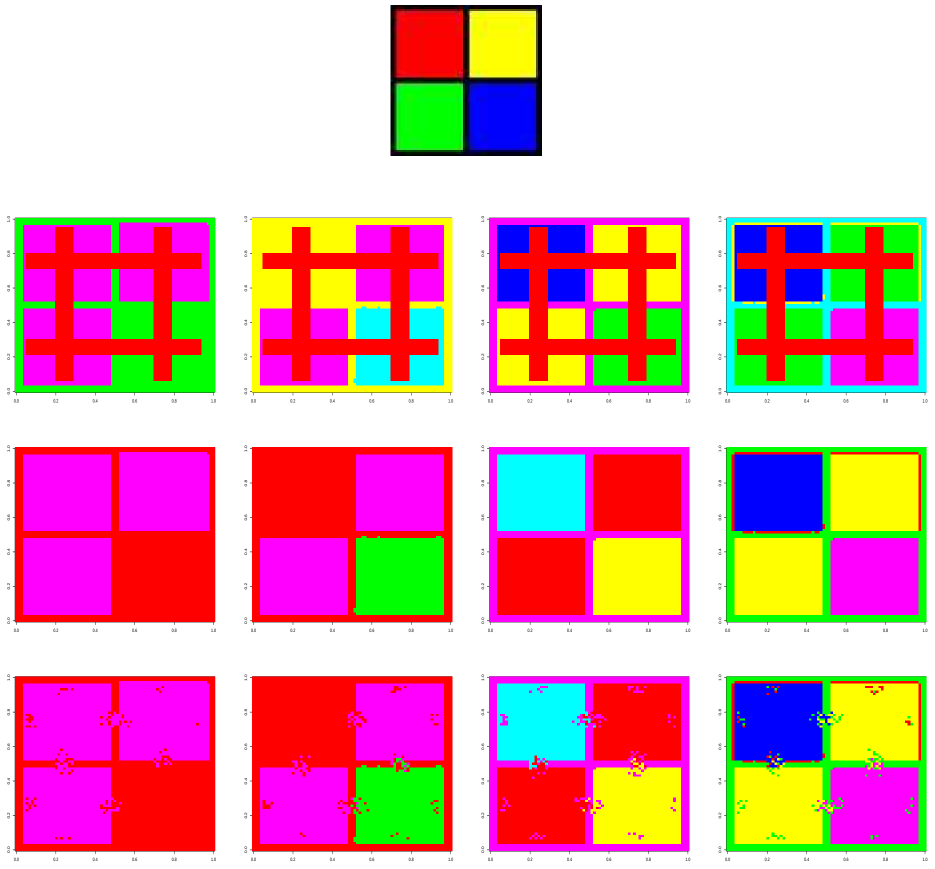


Figure 49: a. Colored checkerboard, b. - e. K-means solutions for K=2, 3, 4, and 5, f. - i. Minimum Manhattan imputation, j. - m. Weighted nearest neighbors imputation

Rather than just examining a simple pattern, we choose to next analyze an image with a somewhat complicated structure. The chosen image contains multiple colors along with text (Figure 50). Once again, the K-means algorithm does not have too much difficulty in identifying different groups of observatins, especially when trying to distinguish the background white with the colored text. Unfortunately, some minor shading that we do not care for is being picked up near the edge of the text. When imputing we note that, although not flawless, the minimum Manhattan imputation gives a clearer solution. Problems that arise for this method are that gaps in letters tend to be closed up entirely if they start out missing. For example, consider the final word *GREEN* and the spaces in the top of the *G*, the hole in the R, and the gap in the spokes of the two E letters. On the other hand, the weighted nearest neighbors method just tries to smooth over the entire missing region and produces a much fuzzier solution which may not be as readable to some individuals.

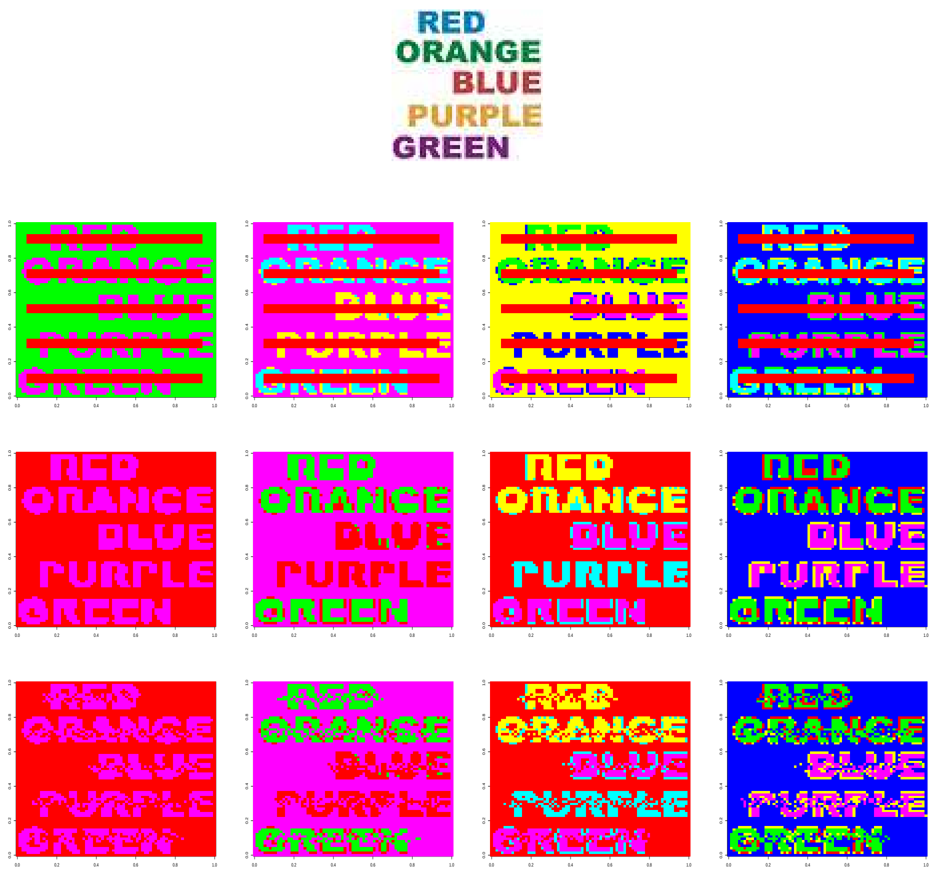


Figure 50: a. Multi-colored words, b. - e. K-means solutions for K=2, 3, 4, and 5, f. - i. Minimum Manhattan imputation, j. - m. Weighted nearest neighbors imputation

In Figure 51, we display the results of imputing with an image of a grapefruit. This image has some complicated shading and color structure, but a pretty consistent pattern of triangular wedges. We would like to see if both the color and shape are recovered. The missing box near the middle of the image serves primarily as a diagnostic for the color retrieval, since the color appears most constant in the region. The slender horizontal missing rectangle near the bottom of the image serves as a diagnostic for the wedge shape pattern retrieval, since it extends across the boundaries of a few different wedges. In all of the solution images, it is quit difficult to tell that there were imputed missing segmentation values; however, this is probably due to the fact that the image is inherently complex and it is quite difficult to tell where true segments should be placed. In some respects, this can be seen as an advantage since mistakes may not be as visually detrimental to the clustering solution.

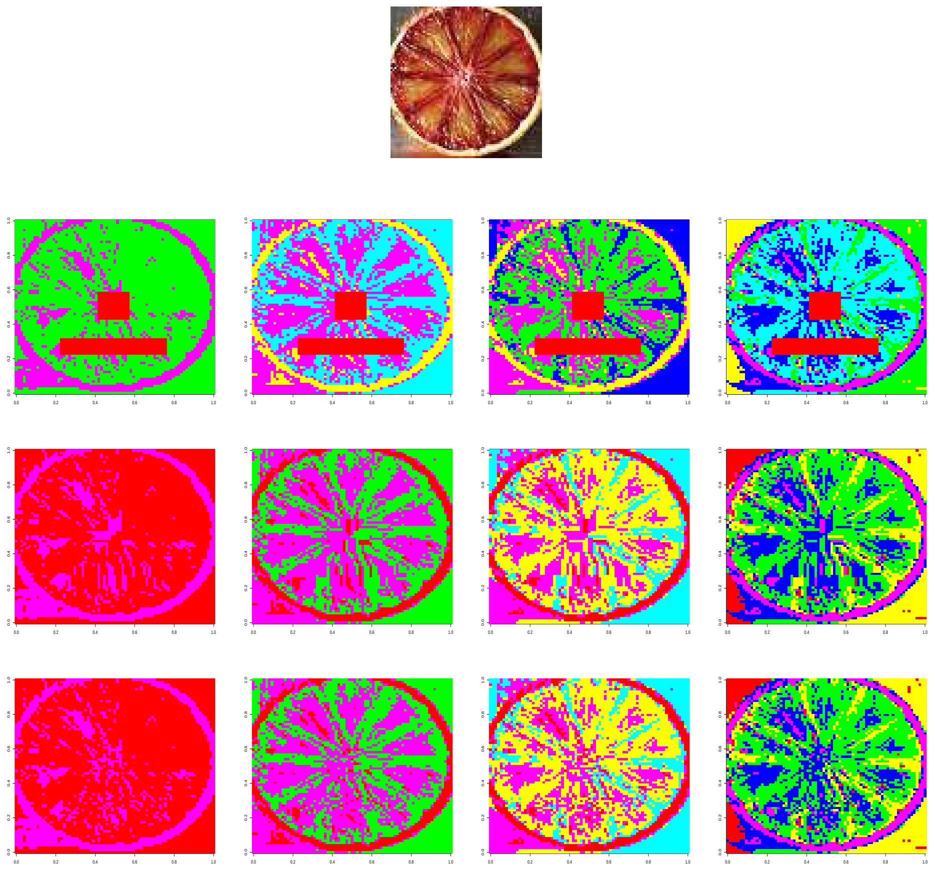
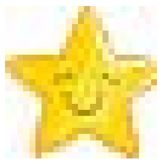


Figure 51: a. Grapefruit, b. - e. K-means solutions for K=2, 3, 4, and 5, f. - i. Minimum Manhattan imputation, j. - m. Weighted nearest neighbors imputation

The next example shows that sometimes a certain type of missingness is irreparable no matter the method of imputation. The image of Figure 52 is of a smiling cartoon star. We choose to delete pixels of approximately half of the star's face, including an entire eye. When using K-means to segment the image, the various leftover features such as the background, star outline and body, smile, and eye are identified quite well; however, when we move on to impute the missing eye all of our methods fail. All solutions for both algorithms seem to simply patch up the area with primarily the cluster that corresponds to the star's body. There is almost no indication that the star's eye was ever initially there. We somewhat expected this kind of solution, since the remaining pixel values held nearly no information about the missing eye. This example illustrates that no matter how sophisticated a method of imputation may be, if the missingness retracts an entire subsection of an image, there is no way to restore what was once there.

Since we have already seen how our algorithms perform when mainly dealing with overt patterns in an image, the next image we choose to segment and impute with is a color wheel (Figure 53). Whereas our analyses images such as the checkerboard focused primarily upon



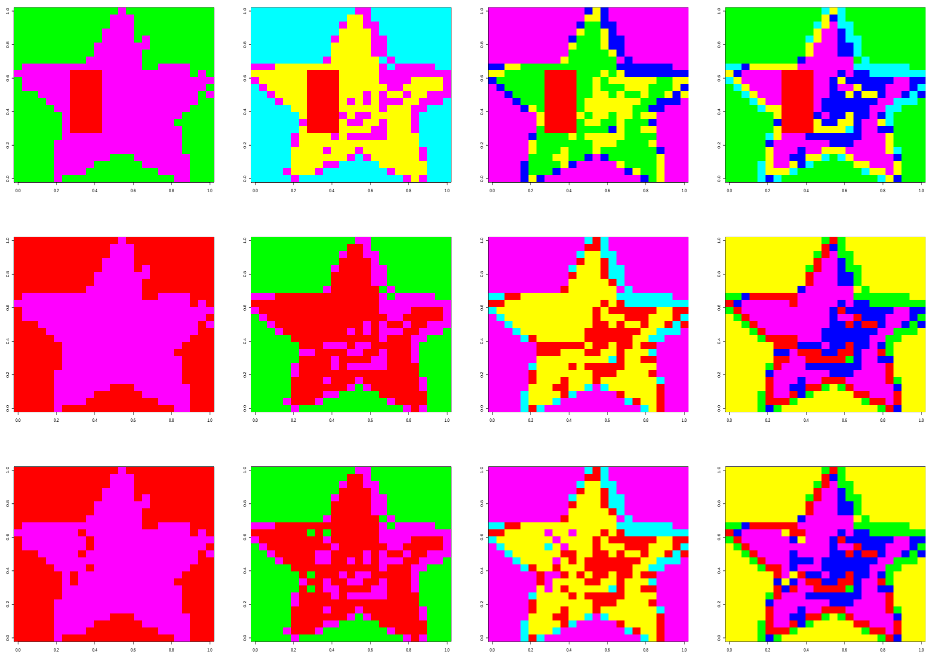


Figure 52: a. Smiling star, b. - e. K-means solutions for K=2, 3, 4, and 5, f. - i. Minimum Manhattan imputation, j. - m. Weighted nearest neighbors imputation

the structure of the image, we will now focus our attention on the color aspect of this new image since it is the predominant feature. First, we note that as we tend towards the center of the images, colors appear to be increasingly washed out and white, which reflects the white background of the image. This would explain why after removing a central section of the image our imputation methods tend to consider the center of the image as background. Since there are so many colors, the K-means algorithm roughly splits the color wheel into equal sections. The minimum Manhattan method tends to produce solutions with quickly changing boundaries, whereas the solutions of the weighted nearest neighbors method has more amorphous segmentations, but appear to look like hazier versions of the minimum Manhattan solutions.

The next image of a shirt and necklace contains some interesting shapes and group boundaries which we would like to try to segment (Figure 54). Once again, since the image is more complex, higher values of K seem to recover structure in the image more accurately. After clustering specifically remove pixels that cross both the circular pendant and the straight chain to see what happens when imputing. We note that the minimum Manhattan distance

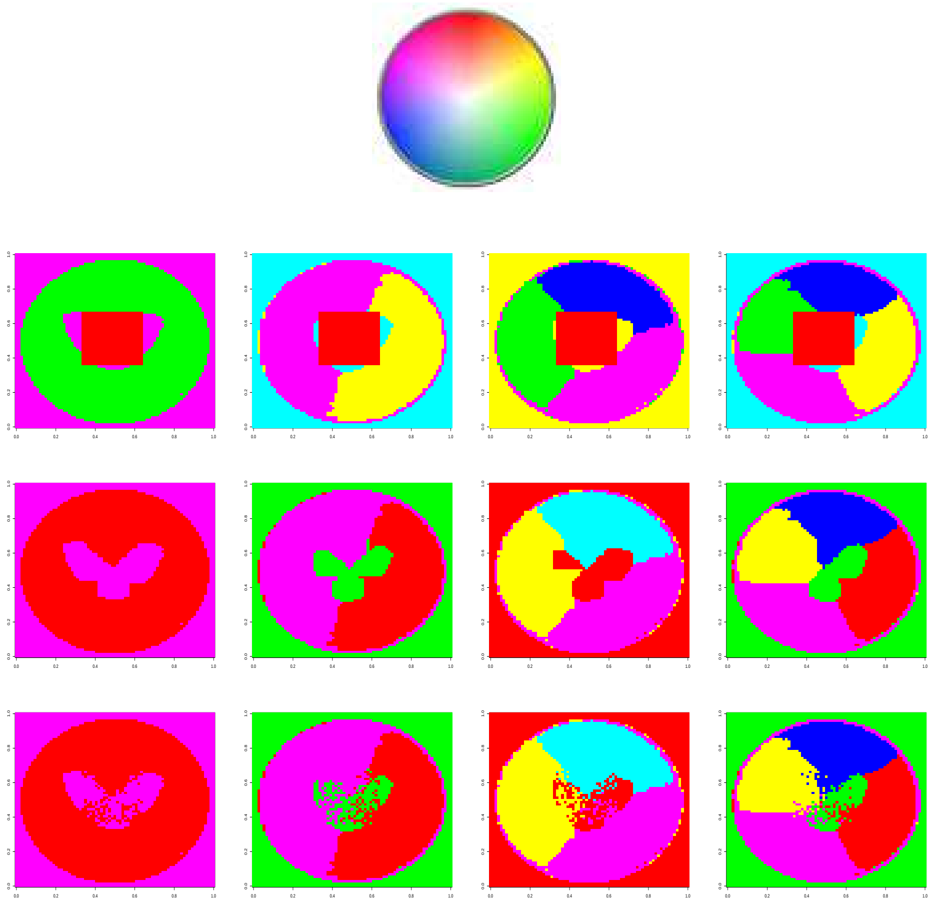


Figure 53: a. Color wheel, b. - e. K-means solutions for K=2, 3, 4, and 5, f. - i. Minimum Manhattan imputation, j. - m. Weighted nearest neighbors imputation

method usually is able to link back together the broken straight chain, whereas the weighted nearest neighbors method is not as successful. Furthermore, both methods are able to somewhat extend the circular pendant's shape downward, but not able to connect the bottom half and recover the original ring shape.

For our final image, we consider a profile image of the same dancer of the bracelet/wrist image, Travis Wall (Figure 55). Notice that the color scheme is a bit more simplistic, yet there is heavy shading throughout the image. We specifically close up on the facial area of the image and induce missingness near the boundary between his nose and the background and the boundary between his eyebrow and forehead. It is clear that as K increases, the shading within the image is more readily recognized. In both the minimum Manhattan and weighted nearest neighbors solutions, the imputation for the nose is not terribly bad. Both shading and shape information appear to be successfully recovered. On the other hand, for all solutions the eyebrow area segmentation is not imputed as well. In many of the solutions, it seems as if there is simply a rectangular segment placed upon the image that does not

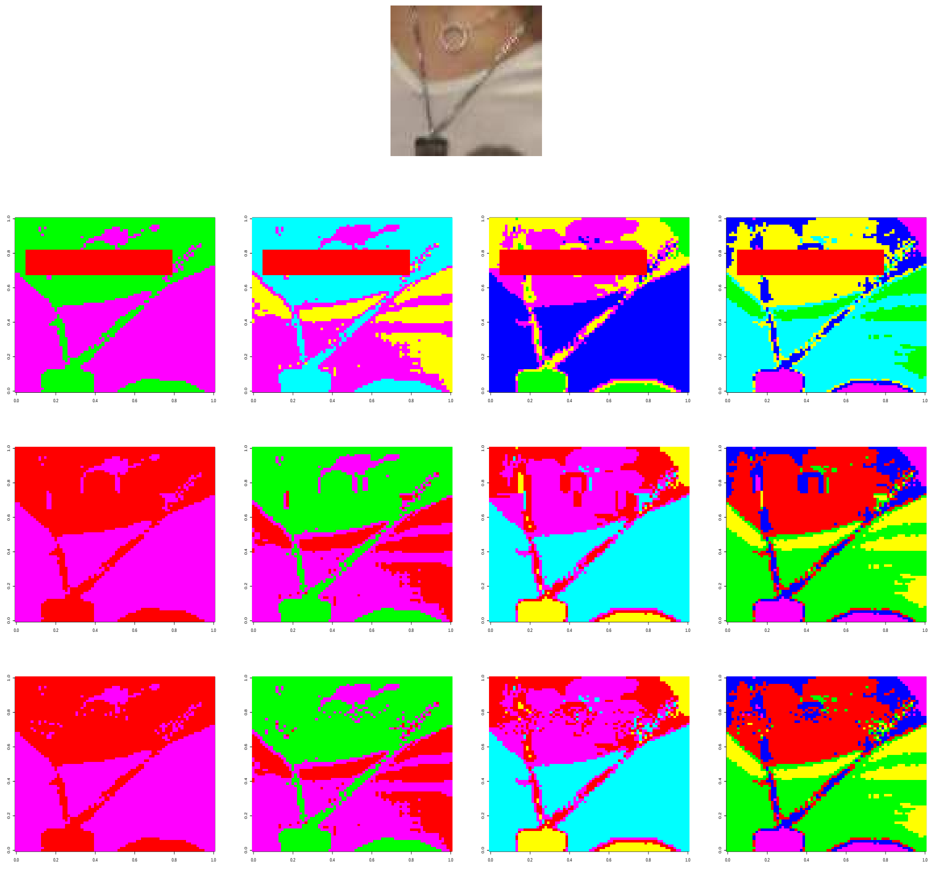


Figure 54: a. Necklaces and shirt, b. - e. K-means solutions for K=2, 3, 4, and 5, f. - i. Minimum Manhattan imputation, j. - m. Weighted nearest neighbors imputation

belong. Neither shading nor shape information are successfully imputed in this area of the image.

6 Conclusions

The problem of missingness is a widespread statistical issue with no perfect solution. In this thesis, we attempted to explore ways in which we may be able to impute missing information in an effort to segment images into like groups. We began with a review of the Perona/Freeman, Ng/Jordan/Weiss, and Scott/Longuet-Higgins spectral clustering algorithms on various artificial datasets and found that they all perform in a comparable manner. Next, we explored the stability of our K-means clustering solutions by permuting through various values of our tuning parameter σ and recording the error rates. Higher values of σ were found to generally be better for consistency. Our next goal was to determine the suitable way in which to represent our image data. After exploring a plethora of different options,

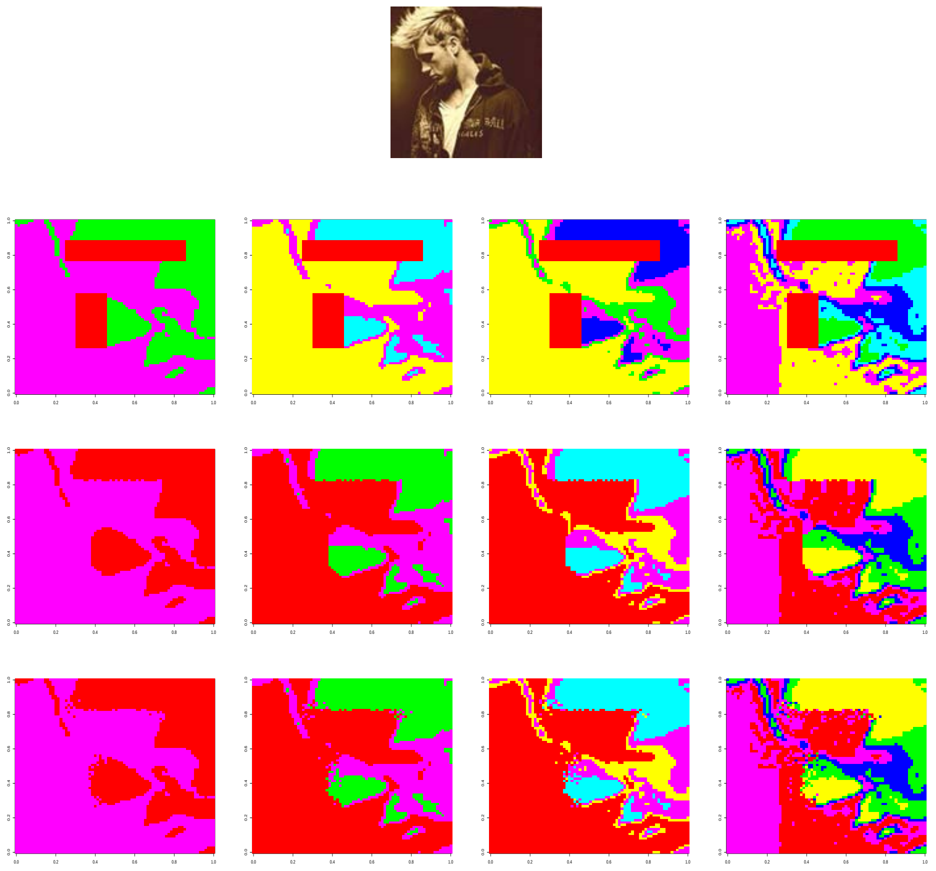


Figure 55: a. Travis Wall profile, b. - e. K-means solutions for K=2, 3, 4, and 5, f. - i. Minimum Manhattan imputation, j. - m. Weighted nearest neighbors imputation

the RGB and HSV values seemed most appropriate, although similar. Finally we began the imputation phase, and found that neither the minimum Manhattan distance nor the weighted nearest neighbors methods are flawless. Both work in some situations, and fail in others. For example, in images that have a clear pattern or structure (such as the checkerboard images), the minimum Manhattan distance method outperforms the weighted nearest neighbors method by far; however, both methods' outcomes are similar given an increase in image complexity.

Although our analysis was a bit extensive, there are still a few open questions and topics for future work. For example, research still needs to be conducted pertaining to the optimal choice of K and σ. As seen within this paper, segmentation solutions vary greatly over certain ranges of both tuning parameters. Furthermore, the usage of other measures of distance besides squared Euclidean may be unknowingly useful at this point.

One attempt at an ideal solution could implement a user interface where a human can subjectively outline, identify, and therefore suggest possible similar segments. An approach of this manner would help reduce the dependence on the non-deterministic nature of K-means since the user would be helping the algorithm begin with initial clusters. Furthermore, an interface that implements user input could also yield various different solutions of the same image, depending on exactly what the user would like to highlight or identify.

References

"Algorithm AS 136: A K-Means Clustering Algorithm." 28.1 (2010): 100-08. Rpt. in By J. A. Hartigan and M. A. Wong. Blackwell for the Royal Statistical Society. Journal of the Royal Statistical Society. Web. 29 Apr. 2011. <http://www.irconan.co.uk/2346830.pdf>.

Altmayer, Lawrence. "Hot-Deck Imputation: A Simple DATA Step Approach." Web. 29 Apr. 2011. <http://analytics.ncsu.edu/sesug/1999/075.pdf>.

Banks, Bruce. Tarnished Artwork. Digital image. NASA. 4 Jan. 2005. Web. 4 Apr. 2011. <http://www.nasa.gov/centers/glenn/business/AtomicOxRestoration.html>.

Checkerboard. Digital image. Blistered Thumbs. Web. 11 Apr. 2011. <http://www.blisteredthumbs.net/avatars/500px-checkerboard_pattern.svg.png>.

Color Wheel. Digital image. Senocular. Web. 4 Apr. 2011. <http://www.senocular.com/fireworks/files/colorwheel.png>.

Fraley, Chris, and Adrian E. Raftery. MCLUST: Software for Model-Based Cluster Analysis. Thesis. 1998. Journal of Classification. Print.

Frevele, Jamie. Facial Recognition Scan. Digital image. Geekosystem. 16 Dec. 2010. Web. 4 Apr. 2011. <http://www.geekosystem.com/facebook-face-recognition/>.

Harvey, Andrew. HSV Cone. Digital image. Andrew Harvey's Blog. 22 Aug. 2009. Web. 21 Apr. 2011. <http://andrewharvey4.files.wordpress.com/2009/08/450px-hsv_color_cone.png>.

Kaulitzki, Sebastian. Tumor X-Ray. Digital image. 123RF. July 2006. Web. 4 Apr. 2011. <http://www.123rf.com/photo_5960423_x-ray-human-head-with-cerebral-tumor.html>.

Klein, Andrew D. "Semi-Automated Collection of Pitch Location and." Thesis. Carnegie Mellon University, 2011. Print.

Melia, Marina, and Jiambo Shi. "A Random Walks View of Spectral Segmentation." Thesis. Carnegie Mellon University & University of Washington. AI Statistics (2001). Print.

Multi-colored Words. Digital image. Blifaloo. Web. 11 Apr. 2011. <http://www.blifaloo.com/images/illusions/color_words.gif>.

Ng, Andrew Y., Michael I. Jordan, and Yair Weiss. "On Spectral Clustering: Analysis and an Algorithm." Thesis. University of California, Berkeley, 2001. Advances in Neural Information Processing Systems (2001): 849-56. Print.

Orange Grapefruit. Digital image. Jurylaw. Web. 4 Apr. 2011. <http://jurylaw.typepad.com/photos/uncategorized /2007/05/02/orange_flickr_423230640_fa99fee3e0_.jpg>.

Patterson, Steve. Pixelated Apple. Digital image. Photoshop Essentials. 2006. Web. 4 Apr. 2011. <http://www.photoshopessentials.com/basics/selections/why-make-selections/>.

Perona, P., and W. T. Freeman. "A Factorization Approach to Grouping." Lecture. European Conference on Computer Vision. Mitsubishi Electric Research Laboratories. Web. 20 Sept. 2010.

Santos, Rafael. Colored Blocks. Digital image. Java Image Processing Cookbook. Web. 11 Apr. 2011. <http://www.lac.inpe.br/JIPCookbook/Resources/Datasets/4colorpattern_orig.png>.

Sapic, Julija. Egyptian Hieroglyphics. Digital image. 123RF. Web. 4 Apr. 2011. <http://www.123rf.com/photo_7494002_egyptian-hieroglyphics.html>.

Scott, G. and Longuet-Higgins, H. An algorithm for associating the features of two patterns. In Proc. Royal Society London, volume B244, pages 21-26, 1991.

Shi, Jianbo, and Jitendra Malik. "Normalized Cuts and Image Segmentation." IEEE Transactions on Pattern Analysis and Machine Intelligence 22.8 (2000): 888-905. Print.

Smiling Cartoon Star. Digital image. Wordpress. Web. 4 Apr. 2011. <http://iuliaradu.files.wordpress.com/2010/01/star-smiley-face-download1.gif?w=300&h=287>.

"Taxicab Geometry." Wikipedia, the Free Encyclopedia. 2 Apr. 2011. Web. 02 May 2011. <http://en.wikipedia.org/wiki/Taxicab_geometry>.

Torn Rug. Digital image. Rug Repair and Rug Cleaning in Atlanta. Web. 4 Apr. 2011. <http://www.oldrugsonline.com/>.

Wall, Travis. Travis Wall, Fire Island Dance Festival. 2010. Photograph. Fire Island, New York. Facebook. Summer 2010. Web. 1 Aug. 2010. <http://www.facebook.com/traviswall#!/photo.php?fbid=1556897395631>.

Wall, Travis. Travis Wall, NUVO Necklace. 2010. Photograph. NUVO Tour. Travis Wall TV. 2009. Web. 9 Oct. 2010. <www.traviswall.tv>.

Weiss, Yair. "Segmentation Using Eigenvectors: a Unifying View." Thesis. University of California, Berkeley. Print.

R Code Appendix

```
#####Creates random 2-dimensional data from specified normal distributions
#####Takes sizes of groups, matrix of means, and matrix of standard deviations
#####Returns data and true labels
generate.data = function(sizes, means, sds) {
n.groups=length(sizes)
data=NULL
true=NULL

for (i in 1:n.groups) {
current.data=cbind(rnorm(sizes[i],means[i,1],sds[i,1]),
rnorm(sizes[i],means[i,2],sds[i,2]))
data=rbind(data, current.data)
true=c(true, rep(i,sizes[i]))
}

return(list(data=data, true=true))
}


#####Creates affinity matrix A using Euclidean distance
#####Takes data and sigma
#####Returns affinity matrix A
A.matrix = function(data, sigma) {
numerator=(-1*as.matrix(dist(data)^2))
denominator=(2*(sigma^2))
A=exp(numerator/denominator)
return(A)
}


#####Creates a matrix with pixel x & y locations appended
#####Takes data, pixel.height of image, and pixel.width of image
#####Returns edited matrix data.new
Append.Loc = function(data, pixel.height, pixel.width) {
y=sort(rep(seq(1:pixel.width), pixel.height))
x=rep(seq(1:pixel.width), pixel.height)
data.new=cbind(data, x, y)
return(data.new)
}


#####Creates a picture rotated 90 degrees clockwise
#####Takes a picture vector, height of image, and width of image
#####Returns rotated picture
Rotate.Picture = function(picture, height, width) {
rotated.picture=matrix(NA, height, width)
for(i in 1:height) {
rotated.picture[, width-(i-1)]=picture[i,]
}
```

```
return(rotated.picture)
}


#####Creates diagonal degree matrix D and D^-.5
#####Takes affinity matrix A
#####Returns diagonal degree matrix D and D^-.5
D.matrix = function(A) {
D=D.5=matrix(0, nrow(A), ncol(A))
diag(D)=apply(A,1,sum)
diag(D.5)=1/sqrt(diag(D))
return(list(D=D, D.5=D.5))
}


#####Creates transition matrix L
#####Affinity matrix A & diagonal matrix D.5
#####Returns transition matrix L
L.matrix = function(A, D.5) {
L=D.5%*%A%*%D.5
return(L)
}


#####Creates maximum eigenvector matrix
#####Takes affinity matrix A and number of clusters K
#####Returns maximum eigenvector matrix X
X.matrix = function(A, K) {
eigenvalues=eigen(A)$values
eigenvectors=eigen(A)$vectors

first=which(eigenvalues==max(eigenvalues))
X=cbind(eigenvectors[,first])
eigenvalues=eigenvalues[-first]
eigenvectors=eigenvectors[,-first]

      for (i in 1:(K-1)) {
current=which(eigenvalues==max(eigenvalues))
X=cbind(X, eigenvectors[,current])
eigenvalues=eigenvalues[-current]
eigenvectors=eigenvectors[,-current]
}
return(X)
}


#####Creates eigenvector matrix sorted by maximum
#####Takes affinity matrix A and number of clusters K
#####Returns eigenvector matrix max.eigenvectors sorted by maximum
sort.eigenvectors = function(A, K) {
eigenvalues=eigen(A)$values
```

```
eigenvectors=eigen(A)$vectors
max.eigenvectors=eigenvectors[,rev(order(eigenvalues))][,1:K]
return(max.eigenvectors)
}


#####Creates normalized length eigenvector matrix Y
#####Takes maximum eigenvector matrix X
#####Returns normalized length eigenvector matrix Y
Y.matrix = function(X, K) {
initial.sum=0

for (i in 1:K) {
initial.sum=initial.sum+(X[1,i]^2)
}

initial.sum=sqrt(initial.sum)
Y=rbind(X[1,]/initial.sum)

      for (i in 1:(nrow(X)-1)) {
sum=0

            for (j in 1:K) {
sum=sum+(X[i+1,j]^2)
}

       sum=sqrt(sum)
Y=rbind(Y,X[i+1,]/sum)
      }
return(Y)
}


#####Creates an assignment comparison table
#####Takes a clustering assignment and true labels
#####Returns a comparison table
comparison.table = function(cluster.assignments, true) {
comparison=table(true, cluster.assignments)
return(comparison)
}


#####Performs Perona/Freeman algorithm
#####Takes data, number of clusters K, and sigma
#####Returns eigenvalues, eigenvestors, and kmeans
Perona.Freeman = function(data, K, sigma) {
A=A.matrix(data, sigma)
eigenvalues=eigen(A)$values
eigenvectors=eigen(A)$vectors
kmeans=kmeans(eigenvectors[,1:K], K)
return(list(eigenvalues=eigenvalues, eigenvectors=eigenvectors,
kmeans=kmeans))
}


#####Performs Ng/Jordan/Weiss algorithm
#####Takes affinity matrix A and number of clusters K
#####Returns Y and kmeans
Ng.Jordan.Weiss = function(data, K, sigma) {
A=A.matrix(data, sigma)
D.5=D.matrix(A)$D.5
L=L.matrix(A, D.5)
X=X.matrix(L, K)
Y=Y.matrix(X,K)
km=kmeans(Y, K)
return(list(Y=Y, km=km))
}


#####Performs Scott/Longuet-Higgins algorithm
#####Takes data, number of clusters K, and sigma
#####Returns normalized length eigenvector matrix Y, Q=Y%*%t(Y), and kmeans
Scott.Longuet.Higgins = function(data, K, sigma) {
A=A.matrix(data, sigma)
X=X.matrix(A, K)
Y=Y.matrix(X, K)
Q=Y%*%t(Y)
km=kmeans(Q, K)
return(list(Y=Y, Q=Q, km=km))
}


#####Creates a dataset with negative ones in the specified region
#####Takes image data of three columns, the dimensions of the image, and
#####dimensions of where the negative ones should appear
#####Returns modified three column dataset
negative.box = function(threecoldata, pic.row, pic.col, row.start, row.end,
column.start, column.end) {

red.matrix=matrix(threecoldata[,1], nrow=pic.row, ncol=pic.col)
green.matrix=matrix(threecoldata[,2], nrow=pic.row, ncol=pic.col)
blue.matrix=matrix(threecoldata[,3], nrow=pic.row, ncol=pic.col)

red.matrix[row.start:row.end, column.start:column.end]=-1
green.matrix[row.start:row.end, column.start:column.end]=-1
blue.matrix[row.start:row.end, column.start:column.end]=-1

new.data=matrix(c(red.matrix, green.matrix, blue.matrix), ncol=3)
return(new.data)
}
```

```
#####Creates a dataset with only positive values (assuming rows are all
#####positive or all negative)
#####Takes dataset with potentially negative values that should be deleted
#####Returns positive dataset with correct corresponding rows and the rows
#####that had negative values
delete.negative = function(negative.matrix) {
rownames(negative.matrix)=seq(1:nrow(negative.matrix))
negative.rows=which(negative.matrix[,1]==-1)
positive.matrix=negative.matrix[-negative.rows,]
return(list(negative.rows=negative.rows, positive.matrix=positive.matrix))
}


#####Example of testing the stability of the sigma values
#####Tests Perona/Freeman sigmas
col.vec=c("red3", "orange", "yellow", "lightgreen", "forestgreen", "blue",
"lightblue", "lightpink", "purple", "lemonchiffon")

data=four.good[,1:2]
true=four.good$true
for (i in seq(.1:1, by=.1)) {
A=A.matrix(data, i)
eigenvalues=eigen(A)$values
eigenvectors=eigen(A)$vectors
error.vec=0
for (j in 1:1000) {
km=kmeans(eigenvectors[,1:4], 4)
error=classError(km$clust, true)
error.vec[j]=error$errorRate
cat(paste("Just completed iteration", j, "on Perona Freeman sigma", i, "\n"))
flush.console()
}
pstitle=paste("4GoodPFSigma", i, ".ps", sep="")
postscript(pstitle)
hist(error.vec, xlab="Error Rate", main=paste("Four Well-Separated Clusters\nPerona,
Freeman, Sigma=", i, ", n=", j, sep=""), prob=TRUE, col=col.vec[i*10])
lines(density(error.vec), lwd=2)
abline(v=mean(error.vec), lwd=2, lty=2)
abline(v=median(error.vec), lwd=2)
dev.off()
}


#####Example of creation of various datasets
Travis=read.jpeg("Travis.jpg")
red.Travis=as.vector(Travis[,,1])
green.Travis=as.vector(Travis[,,2])
blue.Travis=as.vector(Travis[,,3])
col.Travis=cbind(red.Travis, green.Travis, blue.Travis)

bracelet=imagematrix(Travis[290:365,1530:1605,])
```

```
red.bracelet=as.vector(bracelet[,,1])
green.bracelet=as.vector(bracelet[,,2])
blue.bracelet=as.vector(bracelet[,,3])

RGB.bracelet=cbind(red.bracelet, green.bracelet, blue.bracelet)

RGBXY.bracelet=Append.Loc(RGB.bracelet, 76, 76)

RGBXYscaled.bracelet=cbind(RGBXY.bracelet[,1:3], RGBXY.bracelet[,4]/max(RGBXY.bracelet[,4]),
RGBXY.bracelet[,5]/max(RGBXY.bracelet[,5]))

RGBXYallscaled.bracelet=stdize(RGBXYscaled.bracelet)

HSV.bracelet=t(rgb2hsv(t(RGB.bracelet)))

HSVXY.bracelet=Append.Loc(HSV.bracelet, 76, 76)

HSVXYscaled.bracelet=cbind(HSVXY.bracelet[,1:3], HSVXY.bracelet[,4]/max(HSVXY.bracelet[,4]),
HSVXY.bracelet[,5]/max(HSVXY.bracelet[,5]))

HSVXYallscaled.bracelet=stdize(HSVXYscaled.bracelet)

HSVXYallscaledcone.bracelet=cbind(HSVXYallscaled.bracelet[,2]*HSVXYallscaled.bracelet[,3]
*cos(2*pi*HSVXYallscaled.bracelet[,1]), HSVXYallscaled.bracelet[,2]*
HSVXYallscaled.bracelet[,3]*sin(2*pi*HSVXYallscaled.bracelet[,1]),
HSVXYallscaled.bracelet[,3], HSVXYallscaled.bracelet[,4],
HSVXYallscaled.bracelet[,5])


#####Example of creating scree plosts
wss=(nrow(A.RGB.bracelet.eigenvectors)-1)*sum(apply(A.RGB.bracelet.eigenvectors, 2, var))
for (i in 2:15) {
wss[i]=sum(kmeans(A.RGB.bracelet.eigenvectors[,1:i], i)$withinss)
}
plot(1:15, wss, type="b", xlab="Number of Clusters", ylab="Within Groups Sum of Squares",
main="RGB Scree Plot #1: Bracelet Picture")
plot(2:15, wss[2:15], type="b", xlab="Number of Clusters", ylab="Within Groups Sum of Squares",
main="RGB Scree Plot #2: Bracelet Picture")


#####Example of ‘‘easy’’ box removal and segmentation
negative.bracelet.easy=negative.box(RGB.bracelet, 76, 76, 45, 76, 59, 76)
delete.negative.bracelet.easy=delete.negative(negative.bracelet.easy)
positive.bracelet.easy=delete.negative.bracelet.easy$positive.matrix
negative.rows.bracelet.easy=delete.negative.bracelet.easy$negative.rows

A.bracelet.new.easy=A.RGB.bracelet[-negative.rows.bracelet.easy,-negative.rows.bracelet.easy]
eigen.bracelet.new.easy=eigen(A.bracelet.new.easy)

new.labels2.easy=rep(0, nrow(negative.bracelet.easy))
kmeans.positive2.easy=kmeans(eigen.bracelet.new.easy$vectors[,1:2], 2)
```

```
new.labels2.easy[as.numeric(rownames(positive.bracelet.easy))]=kmeans.positive2.easy$clust
image(Rotate.Picture(matrix(new.labels2.easy, nrow=76, ncol=76), 76, 76), col=rainbow(6))

new.labels3.easy=rep(0, nrow(negative.bracelet.easy))
kmeans.positive3.easy=kmeans(eigen.bracelet.new.easy$vectors[,1:3], 3)
new.labels3.easy[as.numeric(rownames(positive.bracelet.easy))]=kmeans.positive3.easy$clust
image(Rotate.Picture(matrix(new.labels3.easy, nrow=76, ncol=76), 76, 76), col=rainbow(6))

new.labels4.easy=rep(0, nrow(negative.bracelet.easy))
kmeans.positive4.easy=kmeans(eigen.bracelet.new.easy$vectors[,1:4], 4)
new.labels4.easy[as.numeric(rownames(positive.bracelet.easy))]=kmeans.positive4.easy$clust
image(Rotate.Picture(matrix(new.labels4.easy, nrow=76, ncol=76), 76, 76), col=rainbow(6))

new.labels5.easy=rep(0, nrow(negative.bracelet.easy))
kmeans.positive5.easy=kmeans(eigen.bracelet.new.easy$vectors[,1:5], 5)
new.labels5.easy[as.numeric(rownames(positive.bracelet.easy))]=kmeans.positive5.easy$clust
image(Rotate.Picture(matrix(new.labels5.easy, nrow=76, ncol=76), 76, 76), col=rainbow(6))


#####Example of ‘‘hard’’ box removal and segmentation
negative.bracelet.hard=negative.box(RGB.bracelet, 76, 76, 52, 61, 1, 43)
delete.negative.bracelet.hard=delete.negative(negative.bracelet.hard)
positive.bracelet.hard=delete.negative.bracelet.hard$positive.matrix
negative.rows.bracelet.hard=delete.negative.bracelet.hard$negative.rows

A.bracelet.new.hard=A.RGB.bracelet[-negative.rows.bracelet.hard,-negative.rows.bracelet.hard]
eigen.bracelet.new.hard=eigen(A.bracelet.new.hard)

new.labels2.hard=rep(0, nrow(negative.bracelet.hard))
kmeans.positive2.hard=kmeans(eigen.bracelet.new.hard$vectors[,1:2], 2)
new.labels2.hard[as.numeric(rownames(positive.bracelet.hard))]=kmeans.positive2.hard$clust
image(Rotate.Picture(matrix(new.labels2.hard, nrow=76, ncol=76), 76, 76), col=rainbow(6))

new.labels3.hard=rep(0, nrow(negative.bracelet.hard))
kmeans.positive3.hard=kmeans(eigen.bracelet.new.hard$vectors[,1:3], 3)
new.labels3.hard[as.numeric(rownames(positive.bracelet.hard))]=kmeans.positive3.hard$clust
image(Rotate.Picture(matrix(new.labels3.hard, nrow=76, ncol=76), 76, 76), col=rainbow(6))

new.labels4.hard=rep(0, nrow(negative.bracelet.hard))
kmeans.positive4.hard=kmeans(eigen.bracelet.new.hard$vectors[,1:4], 4)
new.labels4.hard[as.numeric(rownames(positive.bracelet.hard))]=kmeans.positive4.hard$clust
image(Rotate.Picture(matrix(new.labels4.hard, nrow=76, ncol=76), 76, 76), col=rainbow(6))

new.labels5.hard=rep(0, nrow(negative.bracelet.hard))
kmeans.positive5.hard=kmeans(eigen.bracelet.new.hard$vectors[,1:5], 5)
new.labels5.hard[as.numeric(rownames(positive.bracelet.hard))]=kmeans.positive5.hard$clust
image(Rotate.Picture(matrix(new.labels5.hard, nrow=76, ncol=76), 76, 76), col=rainbow(6))


#####Example of many box removal and segmentation
negative.bracelet.1=negative.box(RGB.bracelet, 76, 76, 1, 15, 53, 61)
negative.bracelet.2=negative.box(negative.bracelet.1, 76, 76, 45, 61, 38, 45)
negative.bracelet.3=negative.box(negative.bracelet.2, 76, 76, 45, 61, 7, 23)
negative.bracelet.4=negative.box(negative.bracelet.3, 76, 76, 15, 31, 22, 31)
negative.bracelet.5=negative.box(negative.bracelet.4, 76, 76, 38, 45, 64, 69)
negative.bracelet.many=negative.bracelet.5

delete.negative.bracelet.many=delete.negative(negative.bracelet.many)
positive.bracelet.many=delete.negative.bracelet.many$positive.matrix
negative.rows.bracelet.many=delete.negative.bracelet.many$negative.rows

A.bracelet.new.many=A.RGB.bracelet[-negative.rows.bracelet.many,-negative.rows.bracelet.many]
eigen.bracelet.new.many=eigen(A.bracelet.new.many)

new.labels2.many=rep(0, nrow(negative.bracelet.many))
kmeans.positive2.many=kmeans(eigen.bracelet.new.many$vectors[,1:2], 2)
new.labels2.many[as.numeric(rownames(positive.bracelet.many))]=kmeans.positive2.many$clust
image(Rotate.Picture(matrix(new.labels2.many, nrow=76, ncol=76), 76, 76), col=rainbow(6))
dev.off()

new.labels3.many=rep(0, nrow(negative.bracelet.many))
kmeans.positive3.many=kmeans(eigen.bracelet.new.many$vectors[,1:3], 3)
new.labels3.many[as.numeric(rownames(positive.bracelet.many))]=kmeans.positive3.many$clust
image(Rotate.Picture(matrix(new.labels3.many, nrow=76, ncol=76), 76, 76), col=rainbow(6))

new.labels4.many=rep(0, nrow(negative.bracelet.many))
kmeans.positive4.many=kmeans(eigen.bracelet.new.many$vectors[,1:4], 4)
new.labels4.many[as.numeric(rownames(positive.bracelet.many))]=kmeans.positive4.many$clust
image(Rotate.Picture(matrix(new.labels4.many, nrow=76, ncol=76), 76, 76), col=rainbow(6))

new.labels5.many=rep(0, nrow(negative.bracelet.many))
kmeans.positive5.many=kmeans(eigen.bracelet.new.many$vectors[,1:5], 5)
new.labels5.many[as.numeric(rownames(positive.bracelet.many))]=kmeans.positive5.many$clust
image(Rotate.Picture(matrix(new.labels5.many, nrow=76, ncol=76), 76, 76), col=rainbow(6))


#####Example of random observation removal and segmentation
make.negative=sample(1:5776, 1444)

negative.bracelet.random=RGB.bracelet
negative.bracelet.random[make.negative,]=-1

delete.negative.bracelet.random=delete.negative(negative.bracelet.random)
positive.bracelet.random=delete.negative.bracelet.random$positive.matrix
negative.rows.bracelet.random=delete.negative.bracelet.random$negative.rows

A.bracelet.new.random=A.RGB.bracelet[-negative.rows.bracelet.random,-negative.rows.bracelet.random]
eigen.bracelet.new.random=eigen(A.bracelet.new.random)

new.labels2.random=rep(0, nrow(negative.bracelet.random))
kmeans.positive2.random=kmeans(eigen.bracelet.new.random$vectors[,1:2], 2)
new.labels2.random[as.numeric(rownames(positive.bracelet.random))]=kmeans.positive2.random$clust
```

```
image(Rotate.Picture(matrix(new.labels2.random, nrow=76, ncol=76), 76, 76), col=rainbow(6))

new.labels3.random=rep(0, nrow(negative.bracelet.random))
kmeans.positive3.random=kmeans(eigen.bracelet.new.random$vectors[,1:3], 3)
new.labels3.random[as.numeric(rownames(positive.bracelet.random))]=kmeans.positive3.random$clust
image(Rotate.Picture(matrix(new.labels3.random, nrow=76, ncol=76), 76, 76), col=rainbow(6))

new.labels4.random=rep(0, nrow(negative.bracelet.random))
kmeans.positive4.random=kmeans(eigen.bracelet.new.random$vectors[,1:4], 4)
new.labels4.random[as.numeric(rownames(positive.bracelet.random))]=kmeans.positive4.random$clust
image(Rotate.Picture(matrix(new.labels4.random, nrow=76, ncol=76), 76, 76), col=rainbow(6))

new.labels5.random=rep(0, nrow(negative.bracelet.random))
kmeans.positive5.random=kmeans(eigen.bracelet.new.random$vectors[,1:5], 5)
new.labels5.random[as.numeric(rownames(positive.bracelet.random))]=kmeans.positive5.random$clust
image(Rotate.Picture(matrix(new.labels5.random, nrow=76, ncol=76), 76, 76), col=rainbow(6))


#####Finds the nearest neighbors of a pixel given the pixel location and image dimensions
pixel.nn = function(pixel.row, pixel.col, picture.row, picture.col) {

#####Inner Region#####
if (pixel.row!=1 && pixel.col!=1 && pixel.row!=picture.row
&& pixel.col!=picture.col) {
nnx=c(pixel.row-1, pixel.row-1, pixel.row-1, pixel.row,
pixel.row, pixel.row+1, pixel.row+1, pixel.row+1)
nny=c(pixel.col-1, pixel.col, pixel.col+1, pixel.col-1,
pixel.col+1, pixel.col-1, pixel.col, pixel.col+1)
}

#####Top Row#####
if (pixel.row==1 && pixel.col!=1 && pixel.col!=picture.col) {
nnx=c(pixel.row, pixel.row+1, pixel.row+1, pixel.row+1, pixel.row)
nny=c(pixel.col-1, pixel.col-1, pixel.col, pixel.col+1, pixel.col+1)
}

#####Left Column#####
if (pixel.row!=1 && pixel.row!=picture.row && pixel.col==1) {
nnx=c(pixel.row-1, pixel.row-1, pixel.row, pixel.row+1, pixel.row+1)
nny=c(pixel.col, pixel.col+1, pixel.col+1, pixel.col+1, pixel.col)
}

#####Right Column#####
if (pixel.row!=1 && pixel.row!=picture.row && pixel.col==picture.col) {
nnx=c(pixel.row-1, pixel.row-1, pixel.row, pixel.row+1, pixel.row+1)
nny=c(pixel.col, pixel.col-1, pixel.col-1, pixel.col-1, pixel.col)
}

#####Bottom Row#####
if (pixel.row==picture.row && pixel.col!=1 && pixel.col!=picture.col) {
nnx=c(pixel.row, pixel.row-1, pixel.row-1, pixel.row-1, pixel.row)
nny=c(pixel.col-1, pixel.col-1, pixel.col, pixel.col+1, pixel.col+1)
}

#####Top Left Corner#####
if (pixel.row==1 && pixel.col==1) {
nnx=c(pixel.row, pixel.row+1, pixel.row+1)
nny=c(pixel.col+1, pixel.col+1, pixel.col)
}

#####Top Right Corner#####
if (pixel.row==1 && pixel.col==picture.col) {
nnx=c(pixel.row, pixel.row+1, pixel.row+1)
nny=c(pixel.col-1, pixel.col-1, pixel.col)
}

#####Bottom Left Corner#####
if (pixel.row==picture.row && pixel.col==1) {
nnx=c(pixel.row-1, pixel.row-1, pixel.row)
nny=c(pixel.col, pixel.col+1, pixel.col+1)
}

#####Bottom Right Corner#####
if (pixel.row==picture.row && pixel.col==picture.col) {
nnx=c(pixel.row-1, pixel.row-1, pixel.row)
nny=c(pixel.col, pixel.col-1, pixel.col-1)
}

nn=cbind(nnx, nny)
return(nn)
}


#####Calculates manhattan distances given pixel location and nearest neighbors
pixel.Manhattan.dist = function(pixel.row, pixel.col, nn) {
row.dist=abs(nn[,1]-pixel.row)
col.dist=abs(nn[,2]-pixel.col)
Manhattan.dist=row.dist+col.dist

return(list(row.dist=row.dist, col.dist=col.dist, Manhattan.dist=Manhattan.dist))
}


#####Removes appropriate missing observations given nearest neighbors
find.row<-function(x,nn){

return(which(x[1]==nn[,1]&x[2]==nn[,2]))

}


#####Imputes pixel cluster labels using Manhattan distance and inherited values from previously
```

```
#####implemented functions
impute.Manhattan = function(nn.new, ring.index,ring.wt,labels.vec, pixel.row, pixel.col,
picture.row, picture.col,method=method) {

##minimum Manhattan distance method;
##only pixels tied for closest get to participate - could be in outside
##rings if inside ring is missing
if(method=="min"){

Manhattan.dist=pixel.Manhattan.dist(pixel.row, pixel.col, nn.new)$Manhattan.dist
min.Manhattan=min(Manhattan.dist)
min.Manhattan.loc=which(Manhattan.dist==min.Manhattan)
min.Manhattan.num=length(min.Manhattan.loc)

##tied pixels
if(min.Manhattan.num>1){
prob.vec=rep(1/min.Manhattan.num, min.Manhattan.num)
random.draw=which(rmultinom(1, 1, prob.vec)==1)
best.nn=nn.new[min.Manhattan.loc[random.draw],]
}
##unique closest
else {
best.nn=nn.new[min.Manhattan.loc,]
}

}
##all nn pixels participate with ring weight probably descending
if(method=="all"){
prob.vec=rep(1/nrow(nn.new),nrow(nn.new))
for(j in 1:length(ring.wt)){
prob.vec[ring.index==j]<-prob.vec[ring.index==j]*
(ring.wt[j]/sum(prob.vec[ring.index==j]))
}
random.draw=which(rmultinom(1, 1, prob.vec)==1)
best.nn=nn.new[random.draw,]
}

impute=matrix(labels.vec, nrow=picture.row, ncol=picture.col)[best.nn[1], best.nn[2]]

return(impute)
}


##Impute missing pixels function
impute = function(missing, original.labels, picture.row, picture.col,n.rings,
ring.wt,method=method) {

cantimpute=NULL
imputed.labels=original.labels

for (i in 1:length(missing)) {
```

```
pixel.row=missing[i]-(floor(missing[i]/picture.row))*picture.row
pixel.col=floor(missing[i]/picture.row)+1

##find near neighbors in rings
nn=ring.nn(pixel.row, pixel.col, picture.row, picture.col,n.rings)
ring.index<-nn[,3]
nn<-nn[,1:2]

##remove near neighbors that are missing pixels
nn.matrix=missing.matrix=matrix(0, picture.row, picture.col)
nn.matrix[nn]=1
missing.matrix[missing]=1

remove=which(missing.matrix==1&nn.matrix==1, arr.ind=TRUE)
remove.loc<-apply(remove,1,find.row,nn)
nn=nn[-remove.loc,]
ring.index<-ring.index[-remove.loc]

##rescaling ring.wt if necessary
if(n.rings!=length(unique(ring.index))){  ##lost at least one full ring to NAs
updated.ring.wt<-ring.wt[unique(ring.index)]/
sum(ring.wt[unique(ring.index)])  #re-scaling

}
else updated.ring.wt<-ring.wt

new.label=impute.Manhattan(nn,ring.index,updated.ring.wt,original.labels,
pixel.row, pixel.col, picture.row, picture.col,method=method)

if (new.label!=0) {
imputed.labels[missing[i]]=new.label
}
else {
cantimpute=c(cantimpute, missing[i])
}
}


return(list(imputed.labels=imputed.labels, cantimpute=cantimpute))
}


##Impute using newly imputed labels function
impute.usingnew = function(missing, original.labels, picture.row,
picture.col,n.rings,ring.wt=NULL,method="min") {

if(is.null(ring.wt)) ring.wt<-rep(1/n.rings,n.rings)  ##default set of weights (terrible)
impute.step=impute(missing, original.labels, picture.row,
picture.col,n.rings,ring.wt,method=method)
imputed.labels=impute.step$imputed.labels
cantimpute=impute.step$cantimpute
```

```
while(length(cantimpute)>0) {
impute.step=impute(cantimpute, imputed.labels, picture.row,
picture.col,n.rings,ring.wt,method=method)
imputed.labels=impute.step$imputed.labels
cantimpute=impute.step$cantimpute
}

return(imputed.labels)
}





#####Finds all nearest neighbors within a certain number of rings
ring.nn = function (pixel.row, pixel.col, picture.row, picture.col, nrings) {

nn=NULL

for (i in 1:nrings) {
nnx=c((pixel.row-i):(pixel.row+i), rep(pixel.row+i, 2*i),
rev((pixel.row-i):(pixel.row+i))[-1], rep(pixel.row-i, 2*i)[-1])
nny=c(rep(pixel.col-i, ((2*i)+1)), ((pixel.col-i):(pixel.col+i))[-1],
rep(pixel.col+i, (2*i)-1), rev(((pixel.col-i):(pixel.col+i))[-1]))
nn.new=cbind(nnx, nny, rep(i,8*i))
nn=rbind(nn, nn.new)
}

x.remove=which(nn[,1]<=0 | nn[,1]>picture.col)
y.remove=which(nn[,2]<=0 | nn[,2]>picture.row)
remove=unique(c(x.remove, y.remove))

if(length(remove)!=0) {
nn=nn[-remove,]
}

return(nn)
}


#####Sample imputation code
wheel=read.jpeg("colorwheel.small.jpg")
red.wheel=as.vector(wheel[,,1])
green.wheel=as.vector(wheel[,,2])
blue.wheel=as.vector(wheel[,,3])
col.wheel=cbind(red.wheel, green.wheel, blue.wheel)

negative.wheel=negative.box(col.wheel, 75, 75, 26, 48, 26, 48)
delete.negative.wheel=delete.negative(negative.wheel)
```

```
positive.wheel=delete.negative.wheel$positive.matrix
negative.rows=delete.negative.wheel$negative.rows

A.newwheel=A.matrix(positive.wheel, 1)
eigen.newwheel=eigen(A.newwheel)

new.labels2=rep(0, nrow(negative.wheel))
kmeans.positive2=kmeans(positive.wheel, 2)
new.labels2[as.numeric(rownames(positive.wheel))]=kmeans.positive2$clust
image(Rotate.Picture(matrix(new.labels2, nrow=75, ncol=75), 75, 75), col=rainbow(6))

new.labels3=rep(0, nrow(negative.wheel))
kmeans.positive3=kmeans(positive.wheel, 3)
new.labels3[as.numeric(rownames(positive.wheel))]=kmeans.positive3$clust
image(Rotate.Picture(matrix(new.labels3, nrow=75, ncol=75), 75, 75), col=rainbow(6))

new.labels4=rep(0, nrow(negative.wheel))
kmeans.positive4=kmeans(positive.wheel, 4)
new.labels4[as.numeric(rownames(positive.wheel))]=kmeans.positive4$clust
image(Rotate.Picture(matrix(new.labels4, nrow=75, ncol=75), 75, 75), col=rainbow(6))

new.labels5=rep(0, nrow(negative.wheel))
kmeans.positive5=kmeans(positive.wheel, 5)
new.labels5[as.numeric(rownames(positive.wheel))]=kmeans.positive5$clust
image(Rotate.Picture(matrix(new.labels5, nrow=75, ncol=75), 75, 75), col=rainbow(6))

RGBwheel2min5=impute.usingnew(negative.rows, new.labels2,
75, 75, 5, c(5/15, 4/15, 3/15, 2/15, 1/15), method="min")
RGBwheel3min5=impute.usingnew(negative.rows, new.labels3,
75, 75, 5, c(5/15, 4/15, 3/15, 2/15, 1/15), method="min")
RGBwheel4min5=impute.usingnew(negative.rows, new.labels4,
75, 75, 5, c(5/15, 4/15, 3/15, 2/15, 1/15), method="min")
RGBwheel5min5=impute.usingnew(negative.rows, new.labels5,
75, 75, 5, c(5/15, 4/15, 3/15, 2/15, 1/15), method="min")

image(Rotate.Picture(matrix(RGBwheel2min5, nrow=75, ncol=75), 75, 75), col=rainbow(6))
image(Rotate.Picture(matrix(RGBwheel3min5, nrow=75, ncol=75), 75, 75), col=rainbow(6))
image(Rotate.Picture(matrix(RGBwheel4min5, nrow=75, ncol=75), 75, 75), col=rainbow(6))
image(Rotate.Picture(matrix(RGBwheel5min5, nrow=75, ncol=75), 75, 75), col=rainbow(6))

RGBwheel2all5=impute.usingnew(negative.rows, new.labels2,
75, 75, 5, c(5/15, 4/15, 3/15, 2/15, 1/15), method="all")
RGBwheel3all5=impute.usingnew(negative.rows, new.labels3,
75, 75, 5, c(5/15, 4/15, 3/15, 2/15, 1/15), method="all")
RGBwheel4all5=impute.usingnew(negative.rows, new.labels4,
75, 75, 5, c(5/15, 4/15, 3/15, 2/15, 1/15), method="all")
RGBwheel5all5=impute.usingnew(negative.rows, new.labels5,
75, 75, 5, c(5/15, 4/15, 3/15, 2/15, 1/15), method="all")

image(Rotate.Picture(matrix(RGBwheel2all5, nrow=75, ncol=75), 75, 75), col=rainbow(6))
image(Rotate.Picture(matrix(RGBwheel3all5, nrow=75, ncol=75), 75, 75), col=rainbow(6))
image(Rotate.Picture(matrix(RGBwheel4all5, nrow=75, ncol=75), 75, 75), col=rainbow(6))
image(Rotate.Picture(matrix(RGBwheel5all5, nrow=75, ncol=75), 75, 75), col=rainbow(6))
```