Viability of Unsupervised Learning for Photo Imputation

Matt Thoburn, Jose Medina, Scott Weitzner

May 4, 2017

Abstract

We explored the viability of unsupervised learning algorithms for restoring photographs with missing pixels. We explored existing libraries for Python, but found them to be unsatisfactory. Because of this, we implemented our own variation on KNN, which has support in the scientific literature for missing data imputation. We found a surprising degree of success in restoring photos of moderate complexity with low to moderate levels of damage, and determined that when using KNN based imputation, lower values of K are better.

1 Introduction

For many people, photographs serve as a window into the past. Over time of course these keepsakes can become worn and torn, but can they be fixed? Currently one can restore photographs through specialized photograph restoration techniques. This generally involves expensive software, tremendous skill, and lots of time. Our work aims to make photo restoration attainable for anyone.

2 Previous Work

2.1 The Literature

In researching for this project we found numerous article abstracts concerning the imputation of missing data. Most of these are concerned with data preprocessing to improve the accuracy of clusters and classifiers, and not so much in terms of imputation for its own sake. A frequently discussed algorithm is K-Nearest Neighbors, which became the starting point for our own imputation.

2.2 Existing Algorithms

In searching for out-of-the-box solutions to our imputation goals, we first turned to one of the most popular Python libraries for Data Science: Scikit-Learn. It comes with a suite of preprocessing utilities, however, they are limited to naive mean, median, and mode which are functions that act on entire rows or columns. This was not nuanced enough for our purposes however, so we did not explore it further.

Figure 1: An example of what we wanted to recreate

(34,34,34)	(25,32,40)	(24,33,48)	(20,34,60)	(23,45,84)	(30,60,112)	(24,62,124)
(34,34,34)	(27,32,36)	(27,34,44)	(24,37,56)	(27,46,79)	(35,59,105)	(28,61,115)
(34,34,34)	(29,31,28)	(31,35,38)	(29,37,50)	(32,47,70)	(39,59,94)	(32,59,102)
(31,31,31)	(46,46,34)	(30,31,25)		(47,56,71)	(52,66,92)	(37,57,90)
(31,31,31)	(43,40,23)	(29,27,15)	(28,28,26)	(43,47,56)	(47,55,74)	(31,47,72)
(31,31,31)	(37,34,17)	(29,27,15)	(29,29,27)	(39,42,49)	(39,46,64)	(26,38,62)
(31,31,31)	(33,30,21)	(32,31,26)	(35,36,40)	(38,42,54)	(35,43,64)	(25,39,66)

Figure 2: An illustration of the imputation process.

We found a K-Nearest Neighbors implementation on github, but it would only search in a row or column for nearest neighbors. This was more promising, but still not nuanced enough. We wanted something that would consider a 360 degree neighborhood around a missing pixel.

3 Our algorithm: K-Radius Neighbors

Our algorithm represents photographs as a 2-Dimensional Matrix of RGBA tuples, where A is the alpha value, which is a measure of opacity, 0 being transparent, and 255 being opaque. We use the alpha to encode missing values by marking them with 0 opacity. We will discuss this further in preprocessing.

The algorithm scans through the matrix looking for missing tuples. Upon arriving at one, it gathers all valid tuples (in bounds and not missing) in a rectangle with a center at the missing point and side length 2 * R where R is a "radius" from the center. It then takes the R, G, and B values and attempts to calculate the mode for each and use those as the new pixel value. An illustration of this can be seen below in figure 2. If no mode can be found, it will instead use the mean, although this introduces some problems as we will discuss later. In rare instances when no pixels can be found in the search radius, the pixel is left blank.

It runs in complexity $O(W * H)$ where W,H are width and height of the image, respectively.

4 Preprocessing

4.1 Image Curation

During our search for a diverse dataset we looked through many online repositories. We wanted to find sets which contained a range of difficult pictures to restore as representative samples of real pictures. The photos were arranged by hand into three separate subcategories: easy, medium, and hard.

Our metric for determining difficulty included color variation and resolution as follows:

1. **Easy**: pictures with large solid colors and resolution of around 284x177.
2. **Medium**: pictures with blocks of similar colors across small regions with resolutions of 250x600.
3. **Hard:** included many color variations among small portions of the photos with resolution ranging from 600x250 to 1200x1600

4.2 Simulating Damage

In order to properly model damage we had to first produce "tears" in our photos. Manually tearing and repairing photos was too difficult due to our need to test multiple times on multiple tear patterns, so we wrote a tear script! This script starts from an edge of the photo and randomly "deletes" pixels in a jagged linear pattern to simulate a tear(actually marks them as transparent). We started with only a single tear, but later added in functionality that allowed us to "tear" certain percentages of the pixels away. A pixel is torn, or deleted, by changing the RGBA value to (0,0,0,0) where the "A" represents the alpha value which correlates to transparency. An example of this simulated tearing can be seen in figure 3.

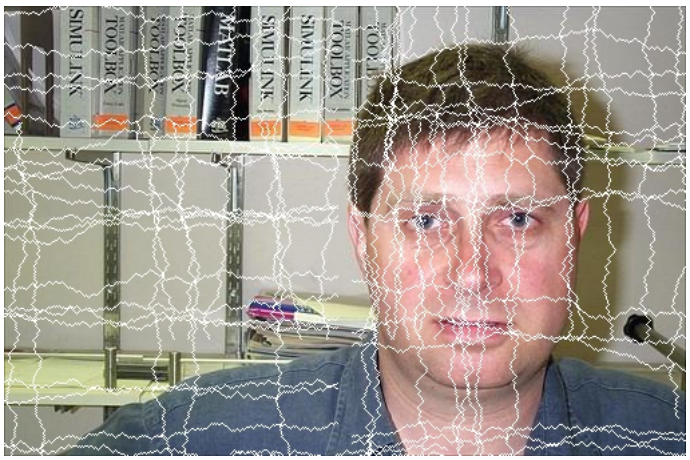

Figure 3: A torn photograph.

degradation	Easy	Medium	Hard
.01	.785	23.74	102.24
.1	.69	33.03	107.98
1	1.167	56.04	148.09

Table 1: Average Euclidean Distance Error (K = 3).

5 Results

5.1 Evaluation Metric

The quality of imputation is evaluated by the euclidean distance between the original pixel value in the training data before damaging and the imputed value, as we can treat the RGB tuples as points in $\mathbb{R}^3$. Thus, the error is defined as:

$E = \sqrt{(r_2 - r_1)^2 + (g_2 - g_1)^2 + (b_2 - b_1)^2}$

5.2 Initial Testing

Our initial testing was to get a better understanding of how our algorithm would perform under different conditions. We chose a sample image of easy, medium, and hard complexity and ran them ten times at degradation values of .01, 1, and 1 each. A K-value of 3 was arbitrarily selected, with the intention of exploring an ideal K-value at a later time. The results can be seen in Table 1 and Figure 3. It is apparent from the data that the complexity of the image has a much greater affect on imputation than image degradation. This is both promising and troublesome. On one hand, our algorithm can still impute at a very low error rate even for very damaged photos. On the other hand, the algorithm is limited to photos of low to medium complexity, and as complexity increases, degradation becomes more of a limiting factor.

One interesting issue with the algorithm, particularly in more complex images, is that in an attempt to impute missing values through the mean (when no mode could be agreed upon), the algorithm would introduce new colors into the image. This lead to black and white images having colored regions where the missing values were. Sometimes it is subtle enough to not be noticeable, but other times it can be quite pronounced. This can be seen in figure 4. It could potentially be up to the user whether or not to forgo mean imputation when no mean could be found.

5.3 Evaluating K Values

Having demonstrated the viability of our imputation algorithm under certain circumstances, the next task was to evaluate different K-values in an attempt to see what trends emerged. Selecting K-values is a matter of introducing trade offs; higher K-values allow a wider sampling, but also have the potential to introduce noise.

This concern about noise is reflected in the additional testing we did. We took five medium difficulty images, and tested them ten times for radii [1,9]. Radii was consistently correlated with error, and has led us to believe that a radius of 1 is the most effective. The results can be seen in figure 5, where error is plotted against radii for each image. With the exception of one image which

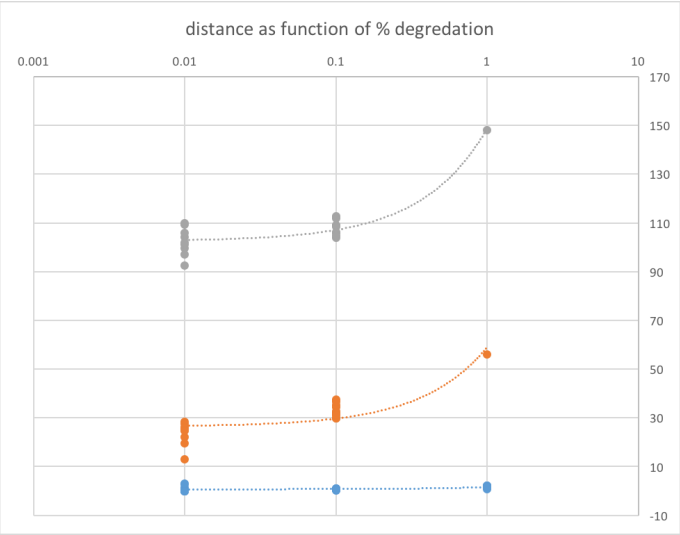

Figure 4: Euclidean Distance Error vs Degradation (K = 3).

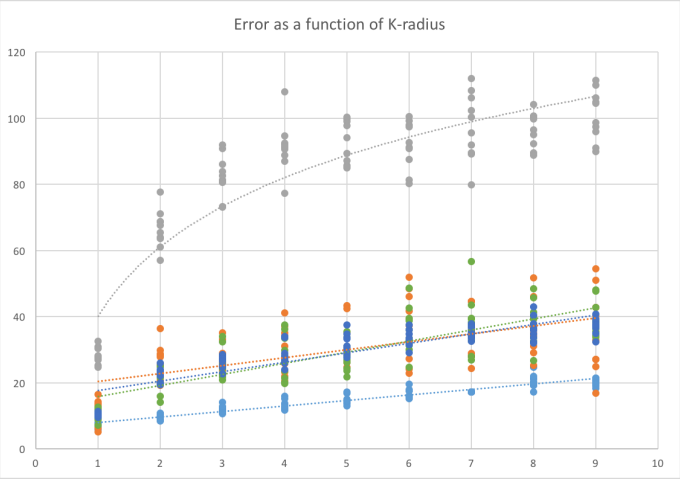

Figure 5: Evaluating different radii.

had higher than average error, the data was surprisingly consistent, showing a gradual increase in error as radii increased.

6 Conclusion

Overall, we are pleased with the results of such a simple algorithm. While it may not be feasible for heavily damaged photos or highly complicated photos, it is more than satisfactory under the right circumstances, and behaves consistently under a given set of circumstances.

7 Logistics

Matt wrote the bulk with the imputation function. Scott wrote the tear function. Jose assisted in debugging of both functions and curated the image dataset. Everyone contributed equally on the paper.

8 Future Thoughts

8.1 Different Error Metric?

Our current error metric includes using euclidean distance to calculate the "distance" of to pixels from their RGB values. It turns out that our algorithm works just fine for this project and our

output looks fine. However, this can be improved. A standard distance measurement for pixels includes using a CIE76 formula:

$\Delta E^*_{ab} = \sqrt{(L^*_2 - L^*_1)^2 + (a^*_2 - a^*_1)^2 + (b^*_2 - b^*_1)^2}$

where $\Delta E^*_{ab} \approx 2.3$ corresponds to a just noticeable difference. This formula takes into account the fact that the eye may not perceive slight changes in RGB values, and thus seeks to measure a more accurate representation of difference. This can be used to find a more appropriate color to impute.

8.2 More Nuanced Algorithms

In the future, it might be worth exploring more nuanced algorithms, as ours was very simple. For instance, it might be worth doing weighted radii, where closer concentric circles of pixels are considered more heavily than further pixels. Furthermore, the accuracy might see improvement if there was a way to detect shapes and boundaries or account for gradients in the colors to contextualize the missing pixels.

What would be interesting, but far beyond the scope of this course, would be to explore the viability of some kind of deep learning or neural network to to employ image recognition algorithms to determine what object the missing pixels belong to, and based on that object, make inferences about how the object should look once the missing pixels are restored.

8.3 Applications

This imputation is interesting in its own right, but it might serve as a useful preprocessing tool for image recognition software that encounters damaged images.