

K-NN Implementation

– Character Recognition from Images

Abstract

The goal of the experiment is to use K-Nearest Neighbor for character recognition. Images for characters have already been segmented and labeled by type(character).

Distance metrics, neighbor amount, weighted or unweighted K-NN, and train set sizes can all affect the learning results. We experimented K-NN with Euclidean, Manhattan, Chebyshev and Minkowski distances. The best accuracy so far is 0.733 by running distance-weighted K-NN on sharpened images, with neighbor amount set to 1 and the distance metric set to Euclidean/Minkowski. For data preprocessing, PCA dimensionality reduction and image processing could sufficiently affect the learning time spent, and the prediction accuracy.

The accuracy scores from cross validation on the train set are generally above 0.94, there is a huge difference (about 0.2) between scores on the validation set and scores on the train set. This is caused by overfitting. That is, the test set has data that never appears in the train set. Why overfitting happens is that we only compare images by pixel values, rather than extracted pattern features, our model by theory should not behave well considering font sizes, font shearing and so on.

For expansion, we also test distance-weighted K-Nearest Neighbor Regression with Gaussian kernel as the base function. Best result is K-nearest neighbors regression with Gaussian sigma = 1/16, neighbor amount k=25. The R2 score under this situation reaches about 0.65. So, the model's interpretability is not good enough. This might be improved by changing the base function for getting weights from distance; Or, this data model is not suitable to be processed by K-nearest neighbors regression.

1 Introduction

The goal is to classify patterns from images corresponding to alphanumeric or other characters.

Our experiment analyzes and manipulates the images from raw data, then outputs each image as a flattened array storing pixel values. Extra image processing is executed in order to get an enhanced image or to extract the useful information from it. K-NN is run on different image sets to determine the best image processing way on our data set.

We analyze how distance metrics, weighted or unweighted distances, and size of the train set can affect K-NN behavior. An expansion experiment with K-Nearest Neighbor Regression is also included.

2 Programming Modules

Numpy and Pandas are used for data processing. Matplotlib is used for analyzing extracted features and training result visualization. For image processing, we invoke the PIL module.

3 Data Analyze

Our source images are mostly at size (20, 20), so 400 pixels to store value information for each sample point. After reading in images, we convert them to grayscale to make sure that for later training, values on each dimension are within the same range.

By reducing the image size, we could lower the calculation weight and time spent for calculating distance in Knn. However, image crisp(edges) cannot be maintained after size reduction.

We could add a filter on top of the raw images or on the downscaled images, to enhance key structures and to remove noises.

3-1. Image Resize for Lighter Calculation

The images are downscaled from (20, 20) to (10, 10). We test nearest, bilinear, bicubic and Lanczos filters on top of the resized images.



The first column are raw images, second are resized images without any filters. From column 3-6, they are resized images with nearest, bilinear, bicubic and Lanczos filters.

Even with filters, processed images are comparatively blurred. Thus, we will try edge sharpening and noise reducing first on raw images, then resize.

3-2. Image Sharpening and Noise Reduction

The PIL library has a built-in filter “ImageFilter.SHARPEN” for this task. After downscaling images, we apply a filter to enhance edges more to remove noises.



Columns(left to right):

[1] Raw images [2] Raw images with sharpen filter [3] Resize(downscale)

[4-7] Resize with nearest, bilinear, bicubic and Lanczos filters and edge enhancement.

After comparing and contrasting, column [2] (sharpened images), and column [5] (sharpened + resize + bilinear filter + enhanced edge) show promising results..

Although edges with the nearest filter may be more crisp(column[4]), however, small details(we are not sure are key structures for recognition) are lost.

3-3. Train Set Construction with Processed Images

Based on the analysis in Section 3.2, we construct three image sets for later testing:

(1) with raw images; (2) with raw images with sharpened filter; (3) resized($\frac{1}{4}$ of original size) on sharpened raw images with Bilinear filter and enhanced edges.

```
def data_from_processed_image(data_path, sharpen=False, resize=False):
    x = []
    y = []

    labels = os.listdir(data_path)
    # Labels originally looks like: ['.DS_Store', '0', '1', '10', '11', '12', '13',...]
    labels.pop(0)

    if resize:
        sharpen = True

    # Now access each labelled folder and read images within it as np.array
    for label in labels:
        label_data_path = os.path.join(data_path, label)
        image_file_names = os.listdir(label_data_path)
        for file_name in image_file_names:
            img = Image.open(os.path.join(label_data_path, file_name))
            img = ImageOps.invert(img)
            # make sure image is grayscale
            img = ImageOps.grayscale(img)
            # image sharpening
            if sharpen:
                img = img.filter(ImageFilter.SHARPEN)
            if resize:
                img = img.resize((10, 10), Image.BILINEAR)

            pixels = np.array(img).flatten()

            # add pixels and associated labels to collection
            x.append(pixels)
            y.append(label)

    return x, y
```









































3-4. Dimensionality Reduction

In general, "the more features we have then the more data we need to train a good model. Expanding on this, if you have a fixed amount of training data (which is often the case) your model's accuracy will decrease for every feature you have"[1].

Normally images have a lot of pixels to retain their clarity, but that significantly increases its size and slows down the performance of the system when it has to process multiple images[2]. A raw image in our data set has 400 dimensions, with resizing, the amount gets cut to 100.

To overcome this situation we can use the dimensionality reduction technique which comes under Unsupervised Machine Learning. A commonly used method is PCA, it sufficiently reduces dimensions, or in other words compresses the image using PCA in python[2].

Below diagram shows the images and recovered images from PCA Dimensionality Reduction.

Sharpened (20*20)	Sharpened + PCA (20*20)	Resized (10*10)	Resized + PCA (10*10)
			
			
			
			
			
			
			
			
			
			

Dimensionality reduction on sharpened (but not resized) images gives a promising compression result. Complicated images are blurred a bit, but still recognizable.

As for downscaled images, simple patterns such as "5", "D", still maintain essential structures. However, the compressed images for complicated images are almost unrecognizable, as the image is already blurred due to resize.

We will test K-NN on 5 different data sets:

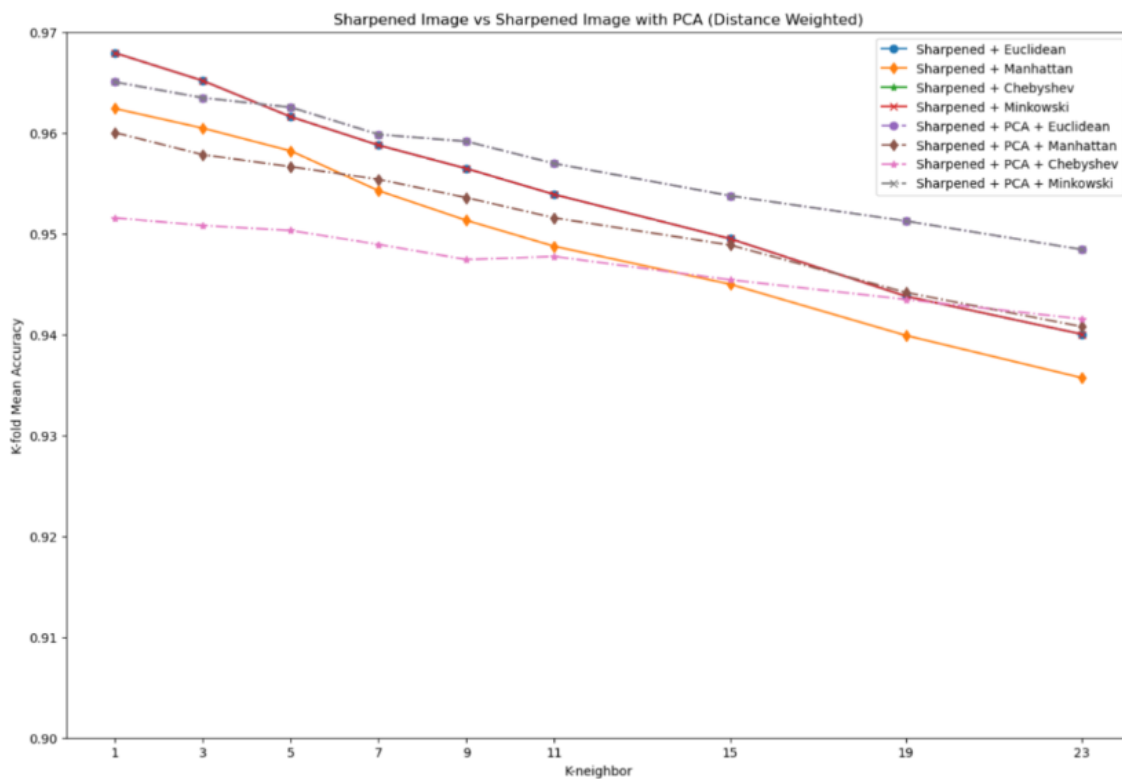
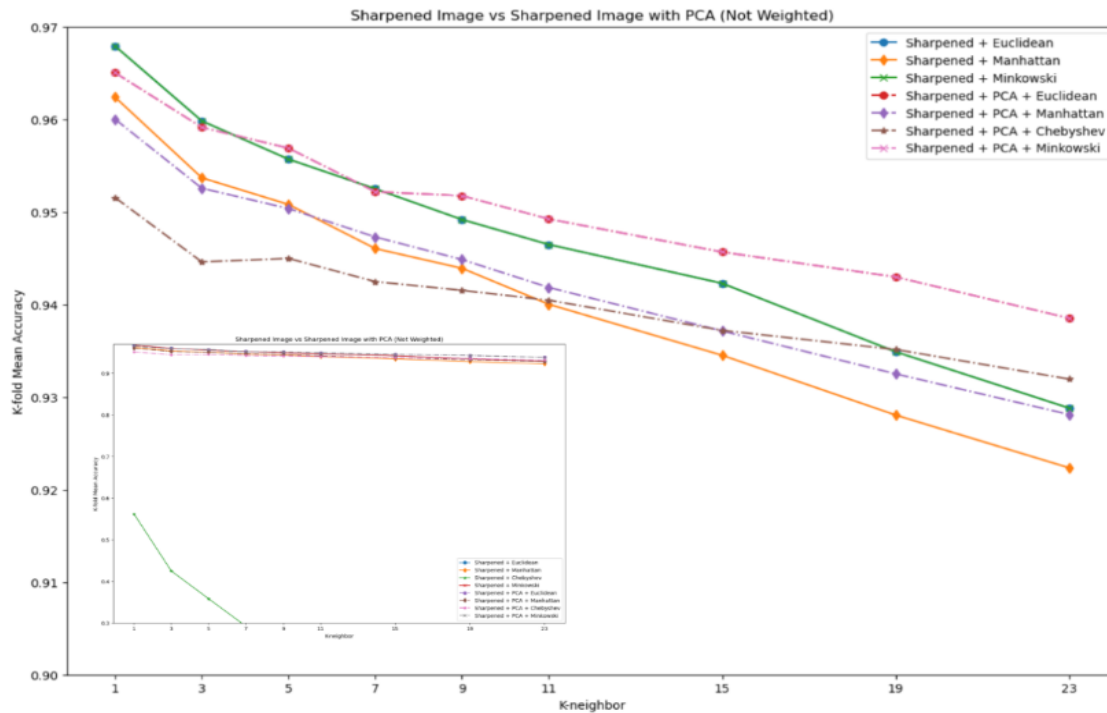
- (1) with raw images; (2) with raw images with sharpened filter; (3) PCA on (2);
- (4) resized($\frac{1}{4}$ of original size) on sharpened raw images with Bilinear filter and enhanced edges;
- (5) PCA on (4).

An initial guess is, resized images with PCA(5) may have the least training cost but lowest accuracy.

4 K-NN on Train Set and Validation Set

Neighbor amount, distance algorithms and whether K-NN is weighted are involved in parameter tuning.

The validation set is splitted from the train set. We will evaluate if observations from the validation set also happen on the test set.



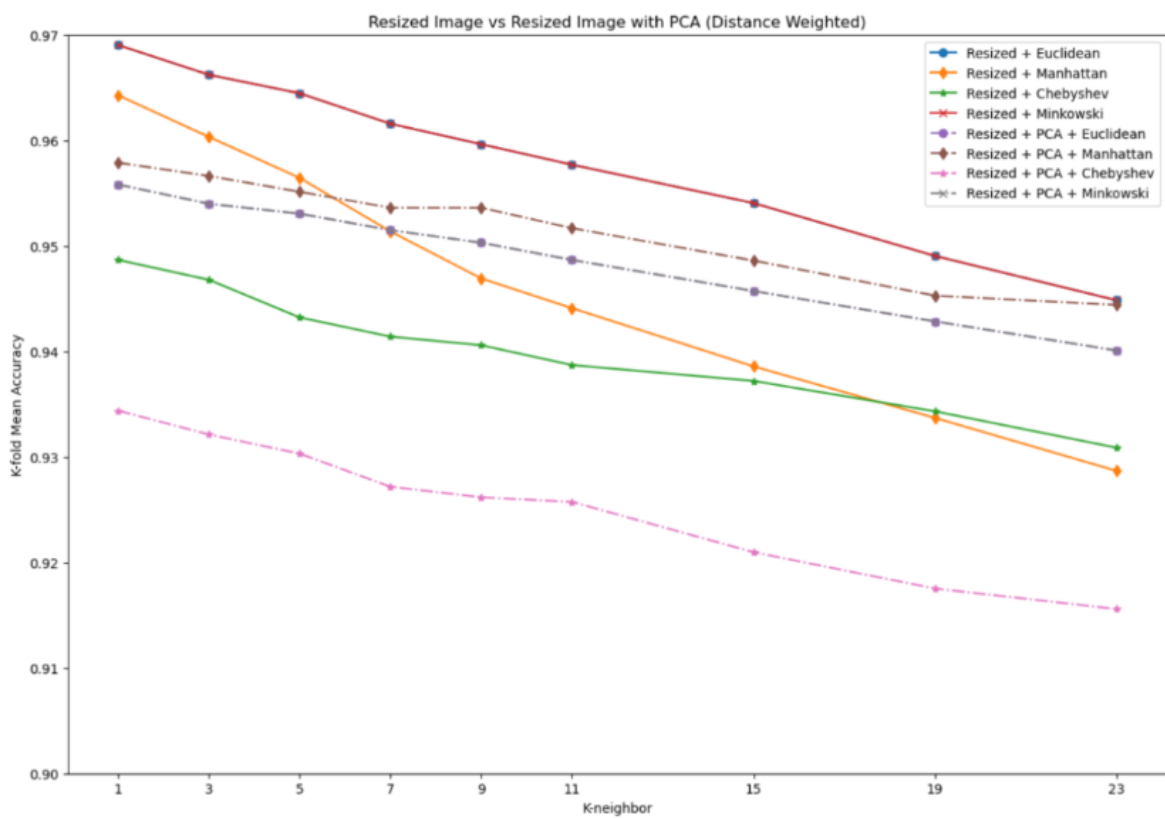
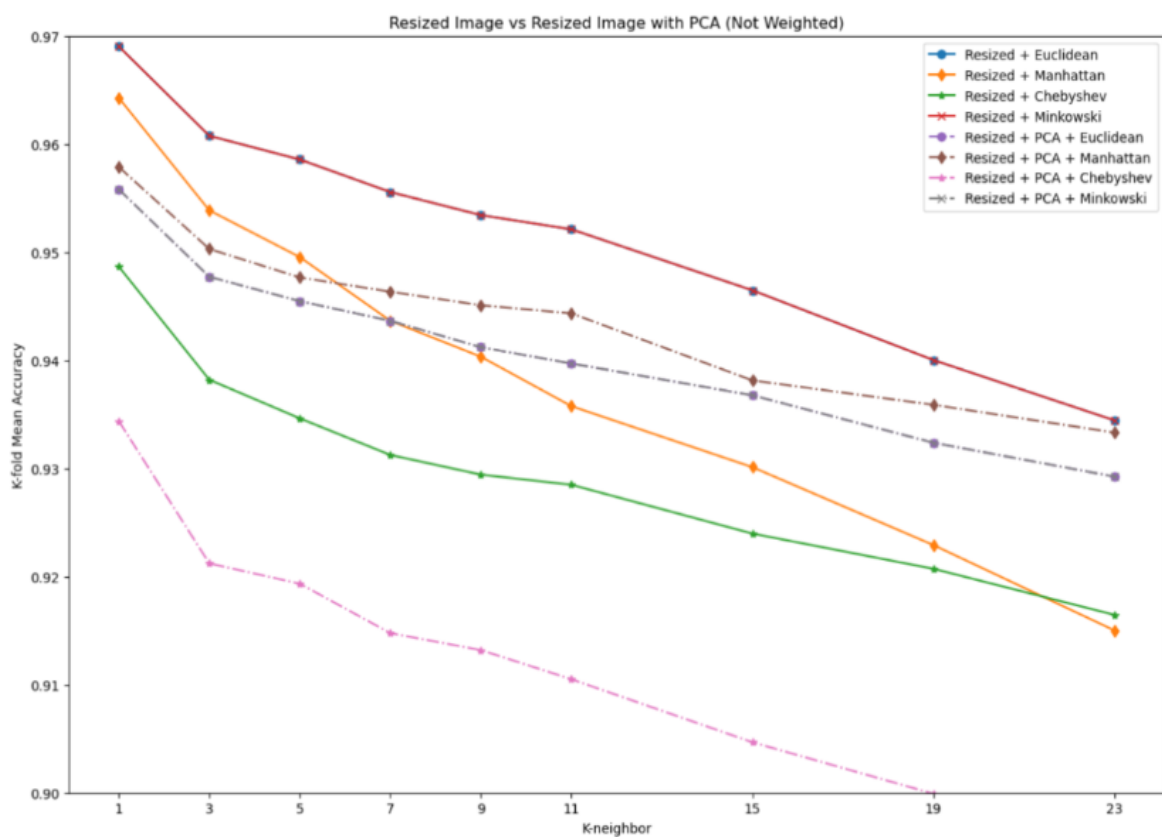


Image 1(top left): Non-Weighted

Performances on sharpened images with Euclidean and Minkowski are the same. Performances of Euclidean and Minkowski on sharpened images with PCA compression are the same. Also, Euclidean and Minkowski perform the best. Manhattan on sharpened images is at first slightly better than on compressed images. As the k grows up to 3, performance on the sharpened image set gets worse than on compression ones. In the smaller image, the worst behavior(green line) is from Chebyshev.

Image 2(top right): Weighted

Compared to Image1, weighted performances are better than unweighted. To explain this conclusion, the highest scores(in Image2) are a bit higher than in Image1, and the lowest scores in Image2 are much higher. Besides, the overall downward trend for accuracy scores with weighted calculation is much smoother.

Minkowski and euclidean still behave the same. Weighted Chebyshev on PCA is much better than non-weighted. However, it is still the worst performance.

An interesting fact is Chebyshev and Euclidean overlaps.

Chebyshev gives out the relatively poorest results. However, weighted Chebyshev on PCA-compressed sharpened images is more stable, by comparing the slopes of the accuracy curves from Image1 and Image2.

Image 3(bottom left): Non-Weighted

Chebyshev still produces the lowest accuracy, and its downward trend is sharp.

Except Manhattan, behaviors of the same distance metrics on resized images are better than on resized ones with PCA compression. This is perhaps because the compression does not sufficiently maintain enough image structures.

Manhattan on resized images is at first better than on compressed resized images. When k reaches to 5, accuracies on compressed images get slightly better. Manhattan is the sum distance between two data points in a grid-like path. Although compression on PCA has caused key structure loss(Section 2.4), by increasing the neighbors amount, more "correct" sample points participate in the vote, as differences from all dimensions add up, and, distances between "similar but incorrect ones" get evened out by "correct" ones.

Image 4(bottom right): Weighted

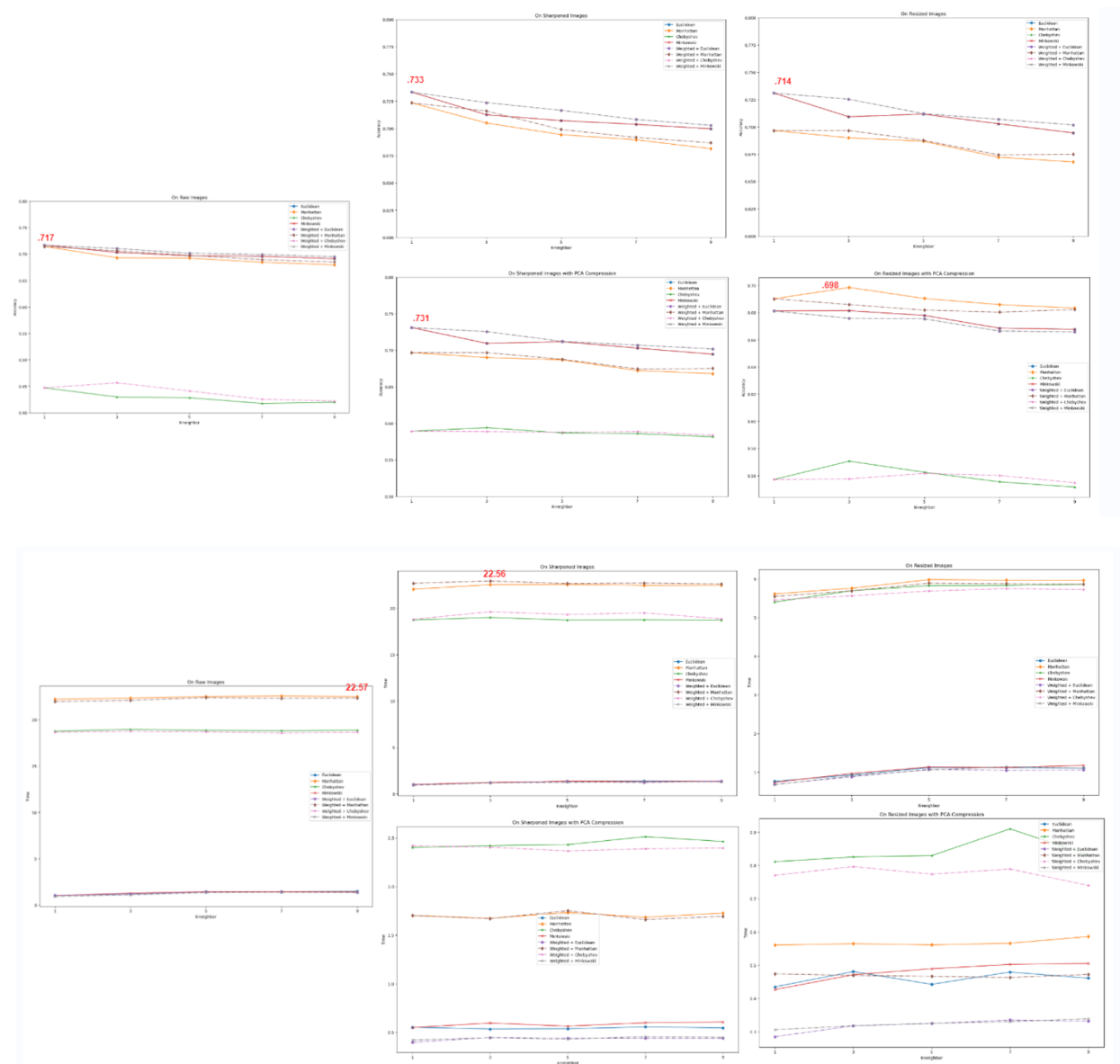
All curves have similar trends and rank as non-weighted curves(in Image3), though the accuracy is higher than without weighted.

It is interesting that non-weighted performances of Euclidean and Minkowski on the same process image set are always the same. This rule also stands for weighted performances. We will test if this remains the same on the test set. We will also investigate if Chebyshev still behaves the worst.

The accuracy scores from cross validation are above 0.95. At this stage, it is hard to say if the model is well-trained or if overfitting happens.

5 K-NN on Train Set and Test Set

In Section 4, accuracy score curves all have downward trends as the neighbor amount increases. Therefore, in this section, we could test limited neighbor amount values.



Overall, accuracy scores generally follow a downward trend as the neighbor amount increases. Time spent(training cost) increases while the neighbor amount grows up. Weighted Knn has better accuracy than non-weighted ones.

Under the same weighted method, Euclidean and Minkowski still have the same accuracy, and Minkowski takes slightly a bit more time to run. Chebyshev is the most expensive distance metric and has the lowest accuracy on this data set. This is perhaps because Chebyshev never provides a sharp bound on the probability[3].

Without any resizing or dimensionality reduction, that is 400-pixel values for each sample point, time spent with Minkowski and Euclidean is similar.

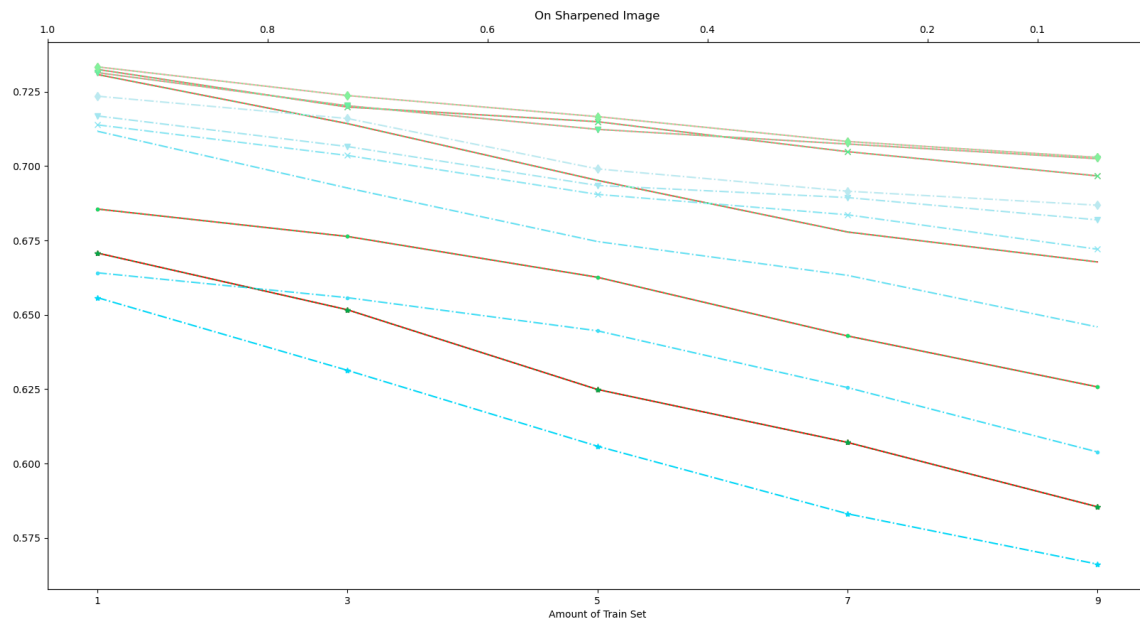
Performances on the test set prove the guess in Section 3.4. Resized images with PCA compression may have the fastest speed (time spent all below 1 sec) but lowest accuracy(peak is below 0.7). Details that form the key structures for identifying the image pattern are lost.

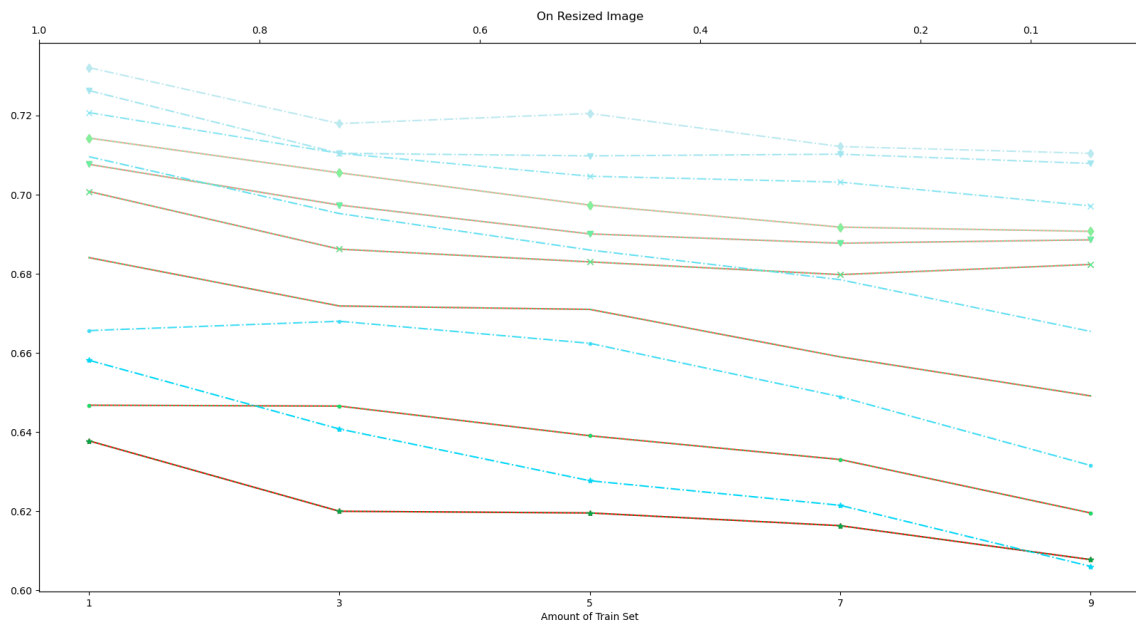
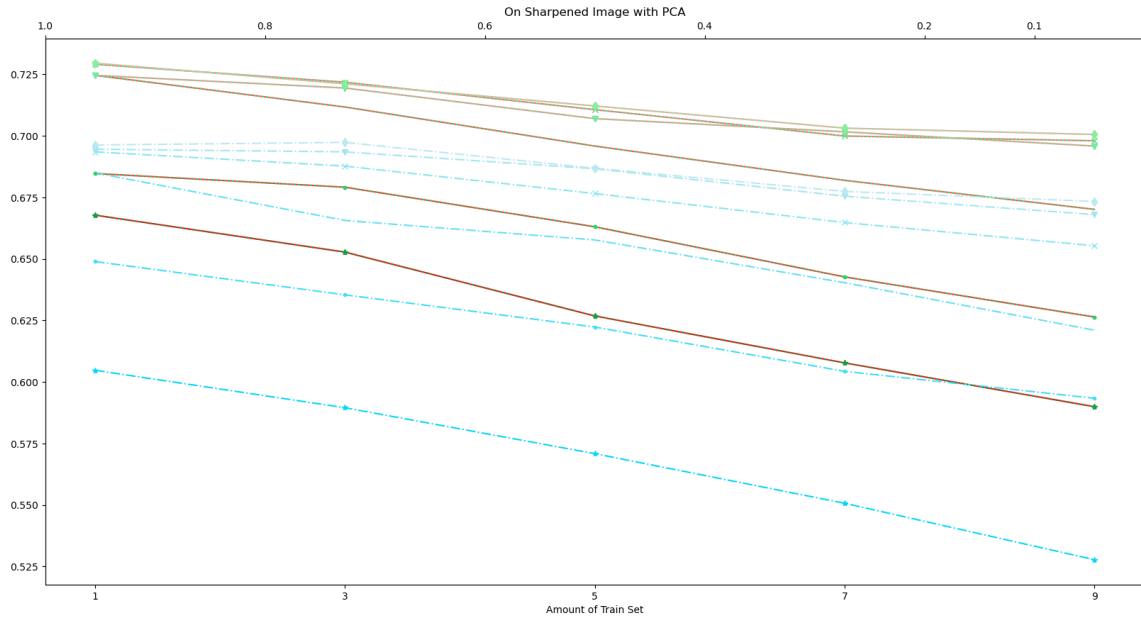
Best accuracy score reaches 0.7333. In general, the cheapest and best combination is to run distance weighted K-NN with Minkowski distance on sharpened images.

Although Manhattan distance is usually preferred over the more common Euclidean distance when there is high dimensionality in the data[4], our graph suggests the other way around. This is potentially because Manhattan is usually used to deal with linear regression or data with linearity.

6. Impact From Train Set Amount

We will take sharpened images, sharpened images with PCA and resized images for testing. Knn weighted by distance will be used in this section, as in the previous section, weighted performs better than non-weighted ones.





The upper X axis is the amount of data we take from the original train set to form the new train set.

Red lines: Euclidean; Blue lines: Manhattan; Green lines: Minkowski.

Accuracy grows as the train set becomes larger. Accuracy on the same data set drops as the neighbor amount increases.

The highest accuracy on sharpened images reaches 0.733 on sharpened images, with PCA compression the accuracy slightly decreases to 0.729.

Regardless of the train set amount, Euclidean and Minkowski distance still behave the same.

On sharpened images (with and without PCA), Minkowski and Euclidean still are the best performing algorithm; Exception happens on the resized images, on which Manhattan distance carries out the best accuracy. Manhattan works better if (1) every dimension is of equal importance, (2) linear relationship exists between dimensions. Although situation(1) may be the true case for our data, (2) is unclear. This might be the reason why it does not behave as well as the other distance methods. Also due to this, without resizing the images to much smaller-dimensional samples, we can presume that "the more features we have then the more data we need to train a good model. Expanding on this, if you have a fixed amount of training data (which is often the case) your model's accuracy will decrease for every feature you have"[2]. That is perhaps why Manhattan behaves better on resized images, as we limit the features we have.

Chebyshev still behaves the worst(not showing on the graph). Online forum mentioned that Chebyshev is extremely useful when measuring the distance between images. However, it is not what the data suggests here. One of the reasons may be that Chebyshev is used to detect similarity between images on extracted features, rather than pixels.

7. K-Nearest Neighbor Regression

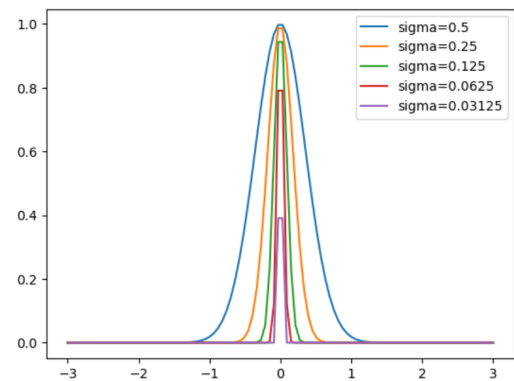
Locally weighted linear regression is the nonparametric regression methods that combine k-nearest neighbor based machine learning. It is referred to as locally weighted because for a query point the function is approximated on the basis of data near that and weighted because the contribution is weighted by its distance from the query point[5].

We use Gaussian function as the kernel to interpolate distances for contribution. Gaussian is the probability density function of the normal distribution. An advantage of Gaussian is that the Central limit theorem states that when we add a large number of independent random variables, irrespective of the original distribution of these variables, their normalized sum tends towards a Gaussian distribution[6].

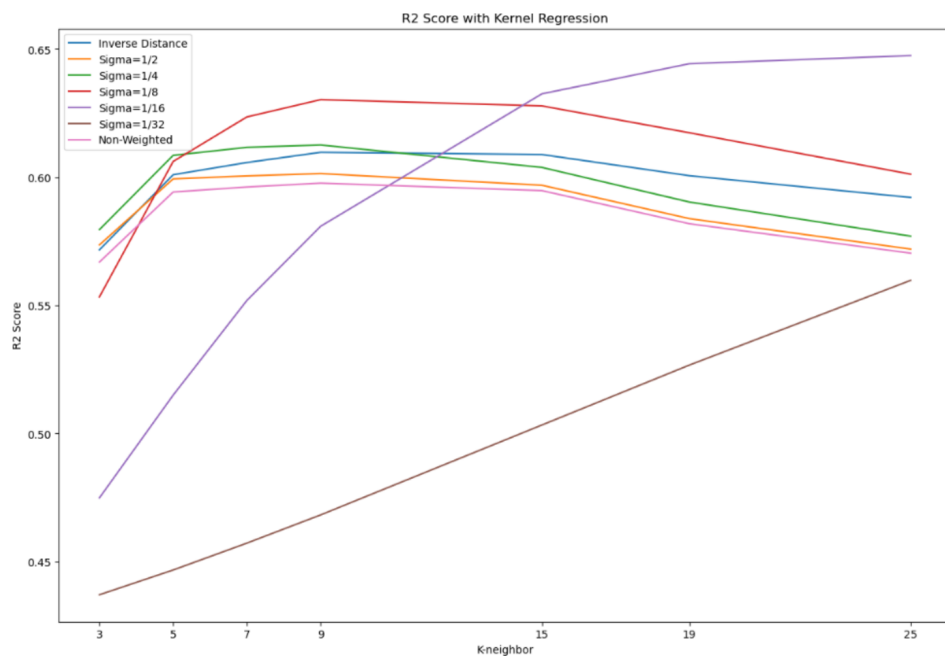
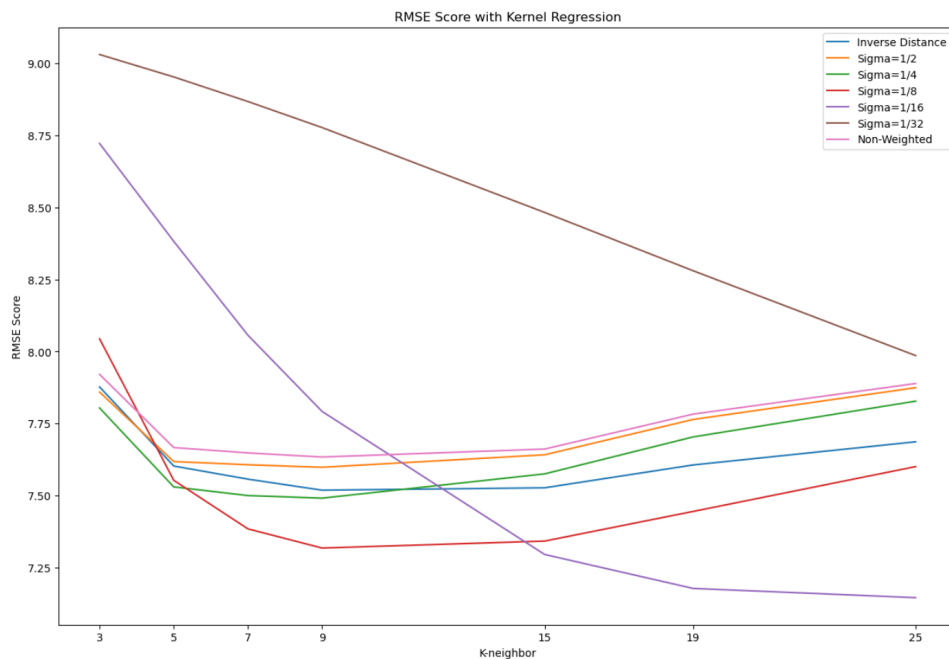
```
def gaussian(dist):
    # if user attempts to classify a point that was zero distance from one
    # or more training points, those training points are weighted as 1.0
    # and the other points as 0.0
    sigma = 1/32
    dist = normalize(dist)
    if dist.dtype is np.dtype(object):
        for point_dist_i, point_dist in enumerate(dist):
            # check if point_dist is iterable
            # (ex: RadiusNeighborClassifier.predict may set an element of
            # dist to 1e-6 to represent an 'outlier')
            if hasattr(point_dist, "__contains__") and 0.0 in point_dist:
                dist[point_dist_i] = point_dist == 0.0
            else:
                dist[point_dist_i] = np.exp(-np.square(point_dist)/np.square(sigma))
    else:
        with np.errstate(divide="ignore"):
            dist = np.exp(-np.square(dist)/np.square(sigma))
        inf_mask = np.isinf(dist)
        inf_row = np.any(inf_mask, axis=1)
        dist[inf_row] = inf_mask[inf_row]
    return dist
```

The parameter sigma in the upper code is the standard deviation, also known as the Gaussian RMS width. It is a measure of how spread out numbers are. The higher the sigma value (between 0-1) is, the more spread out the distribution (probability density) is going to be. We will test on how RMS affects the regression results.

Experiments for this section run on sharpened images without PCA compression. As this set has carried out the best accuracies in Section 5 and 6.



RMSE and R2 scores are used to evaluate the quality of the regression model.



RMSE and R2 scores have negative correlation. High R2 and low RMSE are what a good model should have.

Best regression result is K-nearest neighbors regression with Gaussian sigma = 1/16, neighbor amount k=25. The R2 score under this situation reaches about 0.65. So, the model's interpretability is not good enough. This might be improved by changing the base function for getting weights from distance; Or, this data model is not suitable to be processed by K-nearest neighbors regression.

Non-weighted(uniform) K-nearest neighbors regression has the bottom 3 R2 scores. It is fair to say that distance-based weighted kernel regression performs better than non-weighted.

When sigma gets close to 1, the weights are more expanded, which suggests that more information is aggregated from farther samples. That is the reason R2 scores grow up while sigma decreases. When sigma reaches 1/32, there is a sharp drop on model behaviors. This is where overfitting happens.

When the neighbor amount is small (k below 7), high Gaussian sigma has better R2 scores. This is because we have limited samples to interpolate, and to avoid underfitting, we need to aggregate as much information as we can.

8. Summary

We experimented K-NN with Euclidean, Manhattan, Chebyshev and Minkowski distances. The best accuracy so far is 0.733 by running distance-weighted K-NN on sharpened images, with neighbor amount set to 1 and distance set to Euclidean/Minkowski.

PCA dimensionality reduction could sufficiently reduce the learning time spent, however, at the same time, decrease the prediction accuracy.

K-NN performances can also be affected by the train set amount. On images that haven't been downsampled, the more samples we have in the train set, the better the accuracy we can get.

On the same image data set, Euclidean and Minkowski behave the same. Manhattan behaves better on downsampled images Chebyshev has the worst performances. One of the reasons may be that Chebyshev is used to detect similarity between images on extracted features, rather than directly comparing pixels.

For expansion, we also test distance-weighted K-Nearest Neighbor Regression with Gaussian kernel as the base function. Best result is K-nearest neighbors regression with Gaussian sigma = 1/16, neighbor amount k=25. The R2 score under this situation reaches about 0.65. So, the model's interpretability is not good enough. This might be improved by changing the base function for getting weights from distance; Or, this data model is not suitable to be processed by K-nearest neighbors regression.

Overall, neither K-NN or K-Nearest Neighbor Regression can have accuracy scores more than 0.75, so we could not say we build a model with good precision and interpretability over the data we have. The issue is that the task is in fact a pattern recognition. Only comparing pixel values(colors) does not consider pattern size and transforming(e.g shearing due to perspective) . To improve the model, we should extract pattern features instead.

9. References

- [1] Preprocessing images with dimensionality reduction
<https://www.kaggle.com/code/hamishdickson/preprocessing-images-with-dimensionality-reduction>
- [2] A quick guide to color image compression using PCA in python
<https://towardsdatascience.com/dimensionality-reduction-of-a-color-photo-splitting-into-rgb-channels-using-pca-algorithm-in-python-ba01580a1118>
- [3] What-are-pros-and-cons-of-using-Chebyshev-theorem
<https://www.quora.com/What-are-pros-and-cons-of-using-Chebyshev-theorem%5D>
- [4] Different Types of Distance Metrics used in Machine Learning
https://medium.com/@kunal_gohrani/different-types-of-distance-metrics-used-in-machine-learning-e9928c5e26c7
- [5] Locally weighted linear Regression using Python
<https://www.geeksforgeeks.org/locally-weighted-linear-regression-using-python/>
- [6] Why Data Scientists love Gaussian?
<https://towardsdatascience.com/why-data-scientists-love-gaussian-6e7a7b726859>