CS 440 Assignment 2
Face and Digit Classification

Yuyang Chen(yc791), Zhaohan Yan(zy134),Simiao Fan(sf578)

May 06, 2019

1 Overview

Acknowledgement: This project is based on the one created by Dan Klein and John DeNero that was given as part of the programming assignments of Berkeley's CS188 course. In this project, we designed three classifiers: a naive Bayes classifier, a perceptron classifier and a classifier of our choice. We tested our classifiers on two image data sets: a set of scanned handwritten digit images and a set of face images in which edges have already been detected. Even with simple features, our classifiers will be able to do quite well on these tasks when given enough training data. Optical character recognition (OCR) is the task of extracting text from image sources. The first data set on which we will run our classifiers is a collection of handwritten numerical digits (0-9). This is a very commercially useful technology, similar to the technique used by the US post office to route mail by zip codes. There are systems that can perform with over 99% classification accuracy (see LeNet-5 for an example system in action). Face detection is the task of localizing faces within video or still images. The faces can be at any location and vary in size. There are many applications for face detection, including human computer interaction and surveillance. we will attempt a simplified face detection task in which our system is presented with an image that has been pre-processed by an edge detection algorithm. The task is to determine whether the edge image is a face or not.

Perceptron: Perceptron is a type of linear classifier (binary) that helps to analyze the given information data. It utilizes a single layer neural network for learning and classification. Reading one individual sample at a time, it updates its knowledge of the training data by adjusting the weights iteratively to find a good network. It performs fewer assumptions about data and is regularly more precise than the Naive Bayes classifier.

Naive Bayes: Naive Bayes classifiers are a family of probabilistic classifiers that are based on Bayes' theorem. These algorithms work by combining the probabilities that an instance belongs to a class based on the value of a set of features. In this case we will be testing if images belong to the class of the digits 0 to 9 based on the state of the pixels in the images.

K-Nearest Neighbor(K-NN): In pattern recognition, the k-nearest neighbors algorithm (k-NN) is a non-parametric method used for classification. k-NN is a type of instance-based learning, or lazy learning, where the function is only approximated locally and all computation is deferred until classification. The k-NN algorithm is among the simplest of all machine learning algorithms. In this project I will use K-NN to tell weather digit is from 0-9 and whether a provided data is face or not.

2 Perceptron

My implementation of perceptron is based on the information given in the slides from Lecture 11, developed as a module in perceptron.py. When initializing the module, it generates a matrix with the size of (label number ∗ (feature number + 1)) so that during the training method, it will be simpler and faster to reach the existing data.

The input of the training function will flatten the matrix to a 1D array as a list of feature values. As for classifying, it will use the dot product rule on the weights array and feature array, and the perceptron algorithm will then take the one with the max value to decide whether it's the target or not.

2.1 Testing for Perceptron

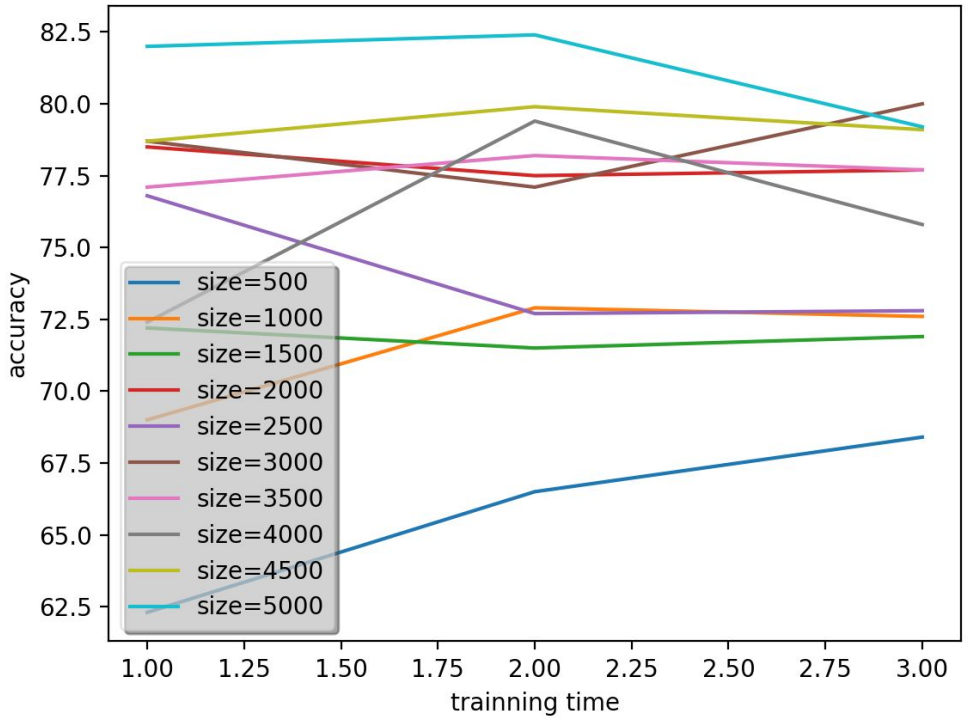
From the given thought in the lecture slide, the algorithm will add or mines given feature data to or from corresponding influences directly. This provides us the following result shown in Figure 1. The accuracy dramatically reduces after the training time increases with the same data collection, and with a large data set (like one the size of 5000). It will drive to a lower accuracy with training time increased. As the weight is directly influenced by each training data, we decide to decrease the influence of each data by adding a learning ratio.
By changing

$$w_j \leftarrow w_j \ (+/-) \ \theta(x_j)$$

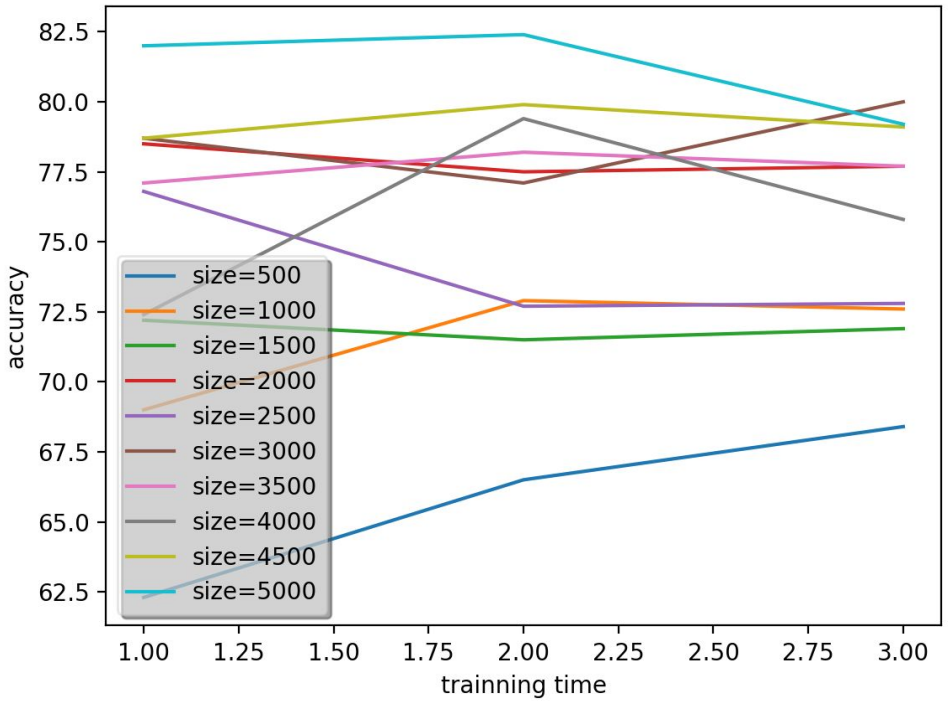
to

$$w_j \leftarrow w_j \ (+/-) \ \text{ratio} * \theta(x_j)$$

and setting the ratio to 0.5, we will get the resulting picture in Figure 2. However, the result is still the same as the result of having no learning ratio. My hypothesis to this conclusion or effect is that for perceptron if there is a max accuracy point for a given data set, like the dark blue line result of size 500 data set, and after reaching that point, the algorithm will stop learning with the same data. Moreover, when the number of data size increases, it has a lower chance to have a fix max accuracy point for a data set, ultimately, in one loop of the data set, there are some data changing the weights back and forth, resulting in a wave-like an accuracy in the resulting graph.



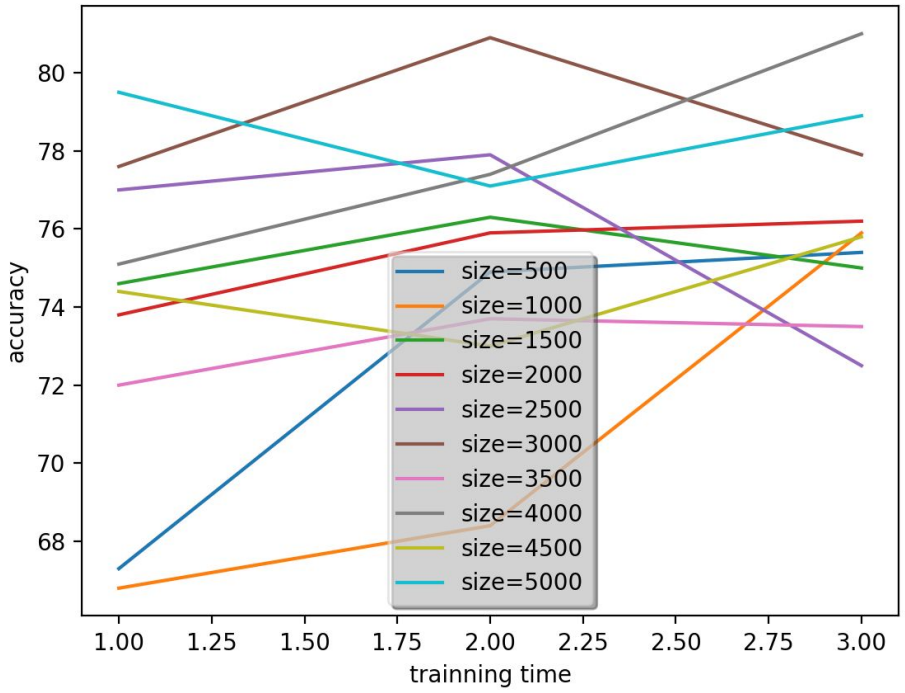
(Figure 1)



(Figure 2 without 0.5 ratio)

2.1.1 Data Order

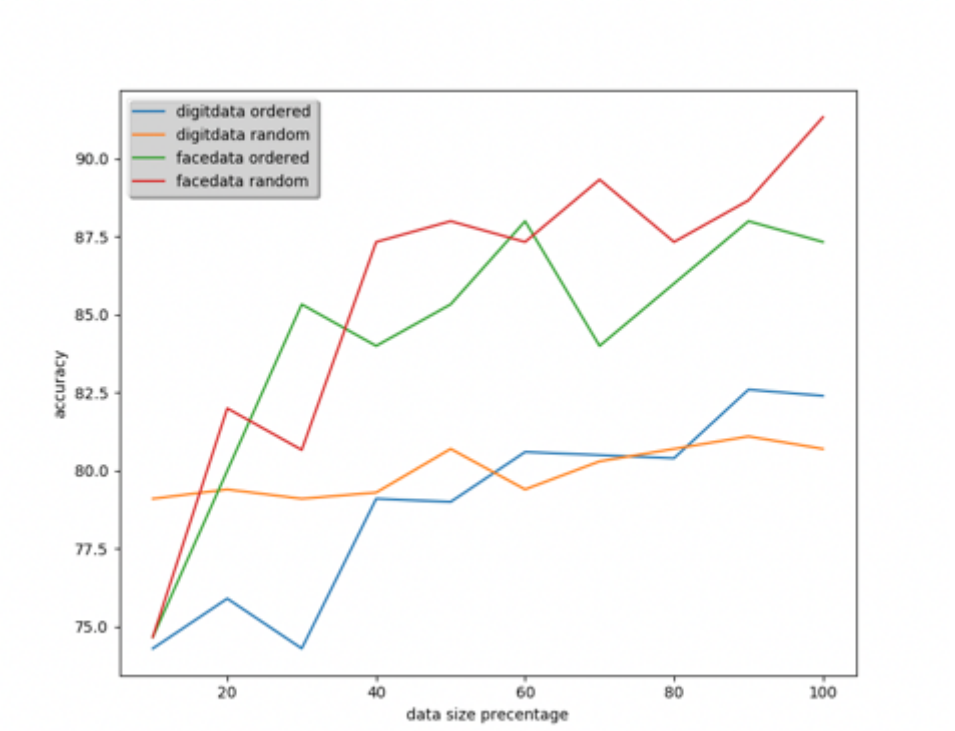
The past two results are based on 10 percent to 100 percent of the whole given data within the delivered order. This doesn't bring the conclusion of whether or not the order of training data will have an impact on the outcome. Thus, we performed an extra function in the data class which will return a random n percent of the whole data set and the results data set is randomly picked from the given data set. Then we have the result shown in Figure 3. From this picture, we see the connections between the accuracy and the size of the data, which is more randomly spread related to ordered data. With the first training,data with the size of 3000 has the lowest accuracy and also the highest accuracy is achieved by data with the size of 2500. Our thoughts to this outcome are that the quality of the data is not the same, which means some data may have a bad impact towards the final accuracy, and with better quality data, the even less size of data is able to achieve greater accuracy.



(Figure 3 without 0.5 ratio without order)

2.1.2 Accuracy and DataSize

For the final result, the correctness is picked from 30 times of training of the same data set and pick the max accuracy as the final exactitude, next in Figure 4, we have four lines which blue and orange lines are the result of digit data with orderly and randomly picked data percentage and the same as green line and red line with corresponding to their label.
Now for faces, the accuracy rises when the size of data increases with both random and ordered data. But in digits, the random line has a much lower degree of slope compared to the ordered data. My hypothesis of this result is the order of the given data set might have been operated to have a better slope for digit data, it means that the quality of the digit data is not as well even as the face data.



(Figure 4)

3 Implementation for Naive Bayes

The image data for this consist of 28 × 28 pixel images stored as text. Each image has 28 lines of text with each line containing 28 characters. The values of the pixels are encoded as ' ' for white and '+' for gray and finally '#' for black. You can view the images by simply looking at them with a fixed width font in a text editor. We will be classifying images of the digits 0-9, meaning that given a picture of a number we should be able to accurately label which number it is. Since computers can't tell what a picture is we're going to have to describe the input data to it in a way that it can understand. We do this by taking an input image and extracting a set of features from it. A feature simply describes something about our data. In the case of our images we can simply break the image up into the grid formed by its pixels and say each pixel has a feature that describes its value. We will be using these images as a set of binary features to be used by our classifier. Since the image is 28 × 28 pixels we will have 28 × 28 = 784 features for each input image. To produce binary features from this data we will treat pixel i, j as a feature and say that Fi,j has a value of 0 if it is a background pixel - white, and 1 if foreground - grey or black. In this case we are generalizing the two foreground values of gray and black as the same to simplify our task.

Feature enhance for digit: We extract the features in three ways and combine them together: Use all the basic features extraction, that is, denote black and white as features for each pixel. For example, feature (x, y) = 1 means pixel (x, y) is non-white feature, while pixel (x, y) = 0 is white feature.

1. Since the grey pixels ('+') are more likely to capture the outline of the digits, we calculate the total gray pixels for each row and the total pixel for each row. For example, suppose a variable grey store the number of grey pixels. Then (DIGIT

DATUM WIDTH , y, 2, grey) = 1 means row y has 'grey' number of grey pixels. Similarly, (x, DIGIT DATUM HEIGHT, 2, grey) = 1 means column x has 'grey' number of grey pixels.

2. Since a digit may end up taking only a small area of the whole image, all the white-space peripheral to the digit can be considered as useless features. Therefore, we trimmed the peripheral of the digit first. After that, we want to find a way to calculate the black pixel and white pixel variation of the image. For example, suppose row 5 has 10 black pixels and row 6 has 12 black pixels, we define the feature to be ('black', 'row', 5, 6, 1) = 1.

3. Basically, the feature format is ('color', 'row(column)', i, i + 1, num), where color can be 'black' or 'white'; 'row' means the feature record the row feature difference between i and (i + 1) row, where 'column' means the feature records the column feature difference between column i and column (i + 1). Num can only take 5 values: -2, -1, 0, 1, 2. For example, '-2' means the ith row(column) has more than 3 additional black(white) pixels than i+1th row(column). '-1' means the ith row(column) has 1-3 additional black(white) pixels than i+1th row(column). '0' means the ith row(column) has the same black(white) column as the (i+1) th row(column). Similar rules apply for the positive value of Num, where i th row(column) has less black(white) pixels than the (i + 1) th row(column).

4. Though the method above indeed improved the overall accuracy for both

4 Testing for Perceptron and Naive Bayes

Digit	Naive bayes	Naive bayes	Naive bayes		Perceptron 3 iterations	Perceptron 3 iterations	Perceptron 3 iterations
Data set	Run time	Validation	Testing		Run time	Validation	Testing
500	69.59	82%	76%		33.95	79%	74%
1000	65.51	88%	80%		71.37	84%	79%
1500	70.97	87%	79%		105.87	80%	79%
2000	65.53	86%	78%		140.89	87%	79%
2500	67.17	88%	79%		180.61	81%	77%
3000	74.22	89%	80%		206.13	92%	85%
3500	81.19	89%	79%		257.89	90%	89%
4000	81.59	89%	79%		293.77	90%	83%
4500	79.83	89%	79%		331.95	87%	86%
5000	82.66	90%	80%		364.19	84%	82%

Figure 5: Results from the Digit recognition

1. For the Digit recognition using the Naive Bayes algorithm, the accuracy and run time both generally increase as we increase the training data points. The correctness of validating data points reaches 90%, and the correctness of testing data points reaches 80%, as we finish training 100% of the data point set.

For testing the digit using Naive bayes, we using the following commands:
python dataClassifier.py -d digits -c naiveBayes -f -a -t 500
python dataClassifier.py -d digits -c naiveBayes -f -a -t 1000
python dataClassifier.py -d digits -c naiveBayes -f -a -t 1500
python dataClassifier.py -d digits -c naiveBayes -f -a -t 2000
python dataClassifier.py -d digits -c naiveBayes -f -a -t 2500
python dataClassifier.py -d digits -c naiveBayes -f -a -t 3000
python dataClassifier.py -d digits -c naiveBayes -f -a -t 3500
python dataClassifier.py -d digits -c naiveBayes -f -a -t 4000
python dataClassifier.py -d digits -c naiveBayes -f -a -t 4500
python dataClassifier.py -d digits -c naiveBayes -f -a -t 5000

2. For the Digit recognition using the Perceptron algorithm, the accuracy and run time both generally increase as we increase the training data points. The correctness of validating data points reaches 84%, and the correctness of testing data points reaches 82%, as we finish training 100% of the data point set.

For testing the digit using Naive bayes, we using the following commands:
python dataClassifier.py -d digits -c perceptron -i 3 -f -a -t 500
python dataClassifier.py -d digits -c perceptron -i 3 -f -a -t 1000
python dataClassifier.py -d digits -c perceptron -i 3 -f -a -t 1500
python dataClassifier.py -d digits -c perceptron -i 3 -f -a -t 2000
python dataClassifier.py -d digits -c perceptron -i 3 -f -a -t 2500
python dataClassifier.py -d digits -c perceptron -i 3 -f -a -t 3000
python dataClassifier.py -d digits -c perceptron -i 3 -f -a -t 3500
python dataClassifier.py -d digits -c perceptron -i 3 -f -a -t 4000
python dataClassifier.py -d digits -c perceptron -i 3 -f -a -t 4500
python dataClassifier.py -d digits -c perceptron -i 3 -f -a -t 5000

Naive Bayes	Run time mean	Validation mean	testing mean	testing std
500	63.448	65.5%	58%	0.06976149845
1000	69.169	72.6%	67.9%	0.03984692924
1500	67.685	77.9%	70.6%	0.02590581228
2000	67.831	78.8%	72.6%	0.04299870788
2500	67.504	84.4%	77.6%	0.03777124126
3000	64.49	83.5%	78.4%	0.03062319382
3500	67.077	83.1%	79.2%	0.002149930231
4000	80.036	85.6%	78%	0.03
4500	68.73	85%	77.5%	0.0313581462
5000	63.837	84.9%	78.5%	0.02460803843

Figure 6 : Final Results from the Digit detection using Naive Bayes

Perceptron	Run time mean	Validation mean	testing mean	testing std
500	0.27	0.747	0.648	5.18
1000	0.34	0.777	0.665	2.47
1500	0.46	0.739	0.728	0.75
2000	0.60	0.780	0.709	2.96
2500	0.72	0.787	0.735	0.44
3000	0.89	0.792	0.768	2.19
3500	0.99	0.776	0.778	0.32
4000	1.16	0.821	0.760	2.75
4500	1.24	0.830	0.776	1.89
5000	1.29	0.771	0.796	1.22

Figure 7 : Final Results from the Digit detection using Perceptron

Digit	Naive bayes	Naive bayes	Naive bayes		Perceptron 3 iterations	Perceptron 3 iterations	Perceptron 3 iterations
Data set	Run time	Validation	Testing		Run time	Validation	Testing
45	21.28	80%	74%		1.07	74%	62%
90	22.02	97%	77%		2.39	91%	76%
135	23.11	100%	85%		3.39	91%	72%
180	22.81	98%	85%		4.27	94%	81%
225	23.05	97%	84%		5.39	95%	79%
270	21.82	97%	86%		6.79	99%	87%
315	21.81	97%	86%		7.91	95%	80%
360	23.69	97%	86%		9.15	93%	80%
405	24.29	96%	88%		10.22	98%	82%
451	22.91	96%	89%		11.28	100%	85%

Figure 8: Results from the Face detection

1. For the Face recognition using the Naive Bayes algorithm, the accuracy and run time both generally increase as we increase the training data points. The correctness of validating data points reaches 96%, and the correctness of testing data points reaches 89%, as we finish training 100% of the data point set.

   For testing the faces using Naive bayes, we using the following commands:

   ```
   python dataClassifier.py -c naiveBayes -d faces -f -a -t 45
   python dataClassifier.py -c naiveBayes -d faces -f -a -t 90
   python dataClassifier.py -c naiveBayes -d faces -f -a -t 135
   python dataClassifier.py -c naiveBayes -d faces -f -a -t 180
   python dataClassifier.py -c naiveBayes -d faces -f -a -t 225
   python dataClassifier.py -c naiveBayes -d faces -f -a -t 270
   python dataClassifier.py -c naiveBayes -d faces -f -a -t 315
   python dataClassifier.py -c naiveBayes -d faces -f -a -t 360
   python dataClassifier.py -c naiveBayes -d faces -f -a -t 405
   python dataClassifier.py -c naiveBayes -d faces -f -a -t 451
   ```

2. For the Face recognition using the Perceptron algorithm, the accuracy and run time both generally increase as we increase the training data points. The correctness of validating data points reaches 100%, and the correctness of testing data points reaches 85%, as we finish training 100% of the data point set.

Naive Bayes	Run time mean	Validation mean	testing mean	testing std
45	19.276	52.6	52	0.021602466899
90	21.332	64.5	59.4	0.05015531433
135	19.978	64.9	60.6	0.07260241808
180	20.442	66	65.6	0.08408989898
225	20.371	65.7	62.5	0.06823163163
270	19.964	67.1	65.6	0.07396695959
315	19.488	74.7	68.5	0.05873670062
360	19.776	74.4	74.8	0.05827139568
405	19.416	72.1	70.7	0.06848357467
451	20.704	74.8	75	0.02924988129

Figure 9 : Final Results from the Faces detection using Naive Bayes

Perceptron	Run time mean	Validation mean	testing mean	testing std
45	0.26	0.643	0.702	2.06
90	0.48	0.779	0.788	3.49
135	0.73	0.821	0.806	2.49
180	0.82	0.765	0.799	1.63
225	0.91	0.792	0.815	5.45
270	1.06	0.786	0.835	2.99
315	1.26	0.801	0.877	0.31
360	1.31	0.844	0.855	1.25
405	1.47	0.859	0.855	2.19
451	1.69	0.876	0.868	1.66

Figure 10 : Final Results from the Face detection using Perceptron

Observation: If the size of our training data set is small, the Perceptron algorithm is faster than the Naive Bayes algorithm, however, the Perceptron algorithm will be much slower if the training data size becomes relatively big, especially if we increase the number of iterations.

We believe the accuracy of the Perceptron algorithm is strongly dependent on the number of iterations we set in some range for training the same number of data points. For example, using perceptron to train 500 data points using 3 iteration gives 75% validation accuracy and 72% testing accuracy; with 5 iterations it gives us 85% validation accuracy and 77% testing accuracy; with 10 iterations, however, the accuracy of both validation and testing don't increase anymore. In addition, the Perceptron algorithm gives us a clear increase of accuracy when we increase the training data points. Comparing with the Perceptron algorithm, the accuracy of the Naive Bayes algorithm barely increases when we increase the sample size.

5 K-NN Implementation

How to run it:

For Digit data: Validation: Python digitv.py Test: Python digitt.py
For Face data: Validation: python facev.py Test: Python facet.py

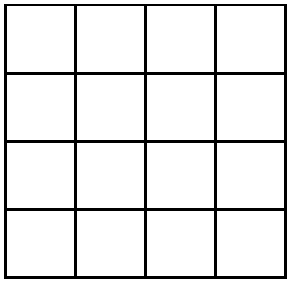
Load Data:

Similar to the previous 2 algorithms, we begin with loading training data. After loading all the data from the dataset, we begin to get the features from data.

Features:

The features I used in KNN is a number of pixels inside a specific region. To have a better result I use totally 4*4 = 16 features as shown below. Which means I divided the image 16 parts and treat each part as one feature. As a result, I will get an array with 16 numbers which represent a number of pixels in each part.

4*4 = 16 features.



Training:

The training part includes the use of Euclidean distance, we need to use the Euclidean distance to calculate the distance between the image in a 16-dimensional graph. After loading the training data with features into a list, we begin the testing.

Testing:

Load each image from the testing dataset as we did with training data. Get the features from test data in the same way we did for training data. We will end up with a list with 16 elements. Use Euclidean distance function we define before to calculate the distance of the test image to each training data and put the distance a min heap. Based on different K value we defined, we will find the smallest K distance and find the most common number within this K numbers. The most common number will be the final result.

Result:
When doing the test we will begin with 10% of data and raising 10% each time to see the trending of the final result:

Digit:

Validation:

```
nbp-197-209:KNN yzh$ python digit8.py
percent is :1
time: 1.2929778099060059
accuracy is 53.7%
percent is :2
time: 2.35910701751709
accuracy is 66.0%
percent is :3
time: 1.9284589290618896
accuracy is 67.3%
percent is :4
time: 1.929197072982788
accuracy is 68.1%
percent is :5
time: 2.485875129699707
accuracy is 70.3%
percent is :6
time: 4.37490701675415
accuracy is 70.3%
percent is :7
time: 3.594860792160034
accuracy is 71.2%
percent is :8
time: 5.2049620151519775
accuracy is 71.2%
percent is :9
time: 5.182692050933838
accuracy is 71.8%
percent is :10
time: 8.421178102493286
accuracy is 72.1%
mean accuracy:68.2
Standard Deviation accuracy:5.486346689738079
mean time:3.6774215936660766
Standard Deviation time:2.1744530535156033
```

Test:

```
nbp-197-209:KNN yzh$ python digit7.py
percent is :1
time: 0.9915280342102051
accuracy is 53.7%
percent is :2
time: 1.791356086730957
accuracy is 66.0%
percent is :3
time: 1.472080945968628
accuracy is 67.3%
percent is :4
time: 2.079306125640869
accuracy is 68.1%
percent is :5
time: 2.332411050796509
accuracy is 70.3%
percent is :6
time: 4.69870400428772
accuracy is 70.3%
percent is :7
time: 3.6575820446014404
accuracy is 71.2%
percent is :8
time: 7.41100287437439
accuracy is 71.2%
percent is :9
time: 5.983299016952515
accuracy is 71.8%
percent is :10
time: 8.343661069869995
accuracy is 72.1%
mean accuracy:68.2
Standard Deviation accuracy:5.486346689738079
mean time:3.876093125343323
Standard Deviation time:2.6178239353040946
```

FACE:

Validation:

```
nbp-197-209:KNN yzh$ python face8.py
percent is :1
time: 0.06083989143371582
accuracy is 43.333333333333336%
percent is :2
time: 0.09498286247253418
accuracy is 46.666666666666664%
percent is :3
time: 0.1188039779663086
accuracy is 52.0%
percent is :4
time: 0.15016603469848633
accuracy is 51.333333333333336%
percent is :5
time: 0.21261906623840332
accuracy is 53.333333333333336%
percent is :6
time: 0.2255840301513672
accuracy is 49.333333333333336%
percent is :7
time: 0.26769495010375977
accuracy is 46.0%
percent is :8
time: 0.3462200164794922
accuracy is 48.666666666666664%
percent is :9
time: 0.34482693672180176
accuracy is 48.0%
percent is :10
time: 0.40467405319213867
accuracy is 50.0%
mean accuracy:48.86666666666667
Standard Deviation accuracy:3.015187892104132
mean time:0.2226411819458008
Standard Deviation time:0.11737268017067602
```

Test:

```
nbp-197-209:KNN yzh$ python face7.py
percent is :1
time: 0.0666801929473877
accuracy is 48.0%
percent is :2
time: 0.10713601112365723
accuracy is 46.666666666666664%
percent is :3
time: 0.12615704536437988
accuracy is 50.666666666666664%
percent is :4
time: 0.13808107376098633
accuracy is 52.666666666666664%
percent is :5
time: 0.16598200798034668
accuracy is 48.0%
percent is :6
time: 0.19442081451416016
accuracy is 49.333333333333336%
percent is :7
time: 0.24417400360107422
accuracy is 54.0%
percent is :8
time: 0.25425291061401367
accuracy is 49.333333333333336%
percent is :9
time: 0.28497982025146484
accuracy is 51.333333333333336%
percent is :10
time: 0.31711816787719727
accuracy is 46.666666666666664%
mean accuracy:49.66666666666667
Standard Deviation accuracy:2.479545956046822
mean time:0.1898982048034668
Standard Deviation time:0.08278853328371029
```

Result Table:

Digit	KNN	KNN	KNN	Face	KNN	KNN	KNN
Data set	Run time	Validation	Testing	Data set	Run time	Validation	Testing
10%	0.9915	53.7%	53.7%	10%	0.0488	43.3%	48.0%
20%	1.7914	66%	66%	20%	0.0817	46.7%	46.7%
30%	1.4721	67.3%	67.3%	30%	0.1111	52%	50.7%
40%	2.0793	68.1%	68.1%	40%	0.1420	51.3%	52.7%
50%	2.3324	70.3%	70.3%	50%	0.1836	53.3%	48%
60%	4.6987	70.3%	70.3%	60%	0.2056	49.3%	49.3%
70%	3.6576	71.2%	71.2%	70%	0.2359	46%	54%
80%	7.4110	71.2%	71.2%	80%	0.2659	48.7%	49.3%
90%	5.9833	71.8%	71.8%	90%	0.3329	48%	51.3%
100%	8.3437	72.1%	72.1%	100%	0.3138	50%	46.7%

Mean and Standard Deviation:

Digit Data(Validation): Mean Time:3.6774s Mean Accuracy:68.2%
Standard Deviation time:3.8761 Standard Deviation accuracy:2.6178

Digit Data(Test): Mean Time:3.4585s Mean Accuracy:68.2%
Standard Deviation time:1.9357 Standard Deviation accuracy:5.4863

Face Data(Validation): Mean Time:0.2226s Mean Accuracy:48.9%
Standard Deviation time:0.1174 Standard Deviation accuracy:3.015

Face Data(Test): Mean Time:0.1921s Mean Accuracy:49.7%
Standard Deviation time:0.0966 Standard Deviation accuracy:2.4795

Improvement:

There are several way to implement the algorithm to help improvement the prediction: Change the feature to a more complex one: For example, the ratio of white and black space. Or treat each one pixels as one features.

6 Conclusion

Implementing the algorithms is not the hardest part of this project, however, finding good features to extract is very critical in this project. In our case, having more features does not always indicate better outcomes, and does improve the accuracy of the classifiers to some degree. Comparatively, our implementations do not give us good results when our training set is small. When we gradually increase the size of the data set, the recognition/detection accuracy can reach a good level (around 80%-90%) in general. We surprisingly found out that the Perceptron algorithm gives us decent results for both the digit recognition and face detection, and the Perceptron does a better job than the Naive Bayes in general. During our testing process, we only used 3 iterations for the sake of time; if we process training with more iterations or with more training data set, we will result in better accuracy. In addition, face detection generally produces better results, and it may be because the face detection only allows two outcomes, either "True" or "False", which means that it has less of a chance to "make mistakes" compared with digit recognition.