Math 480 Final paper

Brian Sherrill

June 2, 2014

Intoduction

GITHUB TO FILES: https://github.com/bsherrill480/final

This project is focused on basic image recognition using python. I chose this project to improve upon my understanding of python programming and learn some techniques of image recogition. I am a rather novice programming student, so I while what I have done may appear meager, it took a surprising amount of work. Figuring out how to write the code, debugging, checking for correct functionality, and general researching turned out to be much more labor intensive than I had planned. To keep things simple I will attempt to recognize just capital Roman letters.

In my project I use Python 2.7 and Three packages Numpy, OpenCV and Scipy. Numpy allows for more efficient operations of data structures such a matrices, OpenCV is a libray for image processing and manipulation, and Scipy allows access to numerical methods.

Contents

1 Setup of Project and Code

My project was done locally on my own Laptop, and as such I can not guarentee how it will preform when ran on other systems. I found a set of hand drawn letters online, which are contained in a folder called "training". This set contained 55 samples each of Roman letters (capital and lower case), as well as Arabic numerals 0-9. Each set of 55 samples is contained in a folder in training; the Roman capitals are in folders Sample011 through Sample036. My code is written such that it uses these folders and expects 55 samples, but I have left some fexibility so if desired it could be extended to include all letters/numbers and less samples. All my code contains minimal comments, so that one should be able to understand the idea of the purpose of the method in a class. My python code uses the

```
if __name__ == "__main__"
```

which is just a trick to make that code in the if statment to behave like the main method of java.

The python file LoadData.py handles the loading and normalizing of the images. Normalizing is the proccess of attempting to standardize data such that each peice of data will be the same scale. To normalize the images of letters, I will trim the excess around the letter and resize all images to a standard size of 16x16 pixels. All images are loaded as black and white images, and then are interved (so that the black text becomes white, and the white background becomes black). This is done to make the process of trimming the image easier (as black is represented as a "0", while white is represented as 255).

Implementing this proved to be a little tricky. I first attempted to isolate the letter by finding the first instance of some percent (tollerance level) of white to total white appearing in a row some row/column of the image. This proved problematic as I soon found that larger images needed a different tollerance level than smaller images. I solved this by resizing the image then finding an appropraite tollerance for that image size. From there I could isolate the letter and normalize. If you would like to see the normalize images, all normalized images were saved in the folder "dump".

I also built in methods to get the average of each letter. Again this was not as straight forwards as I had expected. In the packages I was using, images are represented as numpy arrays of uint8 (unsigned integers of values between 0-255). Then adding uint8 modular arithmetic is used, e.g. $255+1 = 0$. To solve this I converted to foating points, then was able to add and divide to retrieve the average. If you would like to see the averaged images, all averaged images were saved in the folder "averages".

In the use the images in the folder "HandLetters" as my unknown letters. For best results one should draw from the same pool of data that was used to construct the training set (but not from the items in the trainnig set), however since there were only 55 samples of each letter in the training set, I opted to use that data for training and construct my own hand drawn letters.

2 Difference method

2.1 Concept

The method was intended to be a conceptually simple and easy to program. The idea is to use the average letter from the training data to compare against the unknown letter. To determine the unknown letter's identity, one can take a difference between every average letter and the unknown letter. What is left after taking the difference is "error", and the result with the least "error" can be used to determine the unknown letter's identity.

2.2 Implementation

Implementation was rather easy once LoadData was fully debugged. I used the fourth power of the error, thus all entries would be positive and small differences would contribute less and large differences would contribute more error.

2.3 results

Results directly from my code:

```
N.png was identified to be: N
G.png was identified to be: G
F.png was identified to be: F
A.png was identified to be: A
M.png was identified to be: M
P.png was identified to be: P
D2.jpg was identified to be: D
Z.png was identified to be: Z
K.png was identified to be: K
H2.png was identified to be: M
S.png was identified to be: S
X.png was identified to be: L
H.png was identified to be: H
E2.jpg was identified to be: E
B2.jpg was identified to be: B
G2.jpg was identified to be: G
R.png was identified to be: R
O.png was identified to be: O
W.png was identified to be: W
B.png was identified to be: L
L.png was identified to be: L
I.png was identified to be: I
Q.png was identified to be: Q
A2.jpg was identified to be: A
C2.jpg was identified to be: C
T.png was identified to be: F
Y.png was identified to be: Y
V.png was identified to be: V
F2.jpg was identified to be: F
C.png was identified to be: E
```

```
D.png was identified to be: D
U.png was identified to be: U
E.png was identified to be: E
J.png was identified to be: T
6/34 errors testing (difference_average)
```

2.4 Conclusion

This is a relativley simple method, but surprisinly accurate. Thanks to averaging it suffers less from outliers affecting it. This method would not generalize to larger data, because as data size increases there are more objects for something to appear similar to. Also, this method would be poor at detecting fine differences, as fine differences get lost in averaging.

3 k-means

3.1 Concept

K-means (or at least my use of it) is a technique of treating each piece of data as a point in euclidean space. In my situation each image is a single vector of length 16*16=256. Then a set of k "centroids" points are picked and are permuted to find a local minimum of the distance between data points and centroids. One then can hope the centroid to be at the center of a cluster. These clusters can be used to partition the data into distinct catagories. One can find a good description on wikipedia with nice visuals if this is unclear.

To use this as a technique of image recognition I first use the training data to find a set of centroids, with k=26 (because there are 26 letters). Then I can find which centroid belongs to which group of letters (assuming the letters are distinct enough to be their own clusters). Then I can compare the unknown image's distance from each centroid. The centroid that minimized the distance is the cluster to which the unknown image will belong. Since each cluster is associated with a letter, we can determine the unknown letter.

3.2 Implementation

To acheive this I used Scipy's cluster's package. Implementing this proved to be very tricky. The first issue I ran into was there existed "kmeans" and "kmeans2". These two did not seem very different, but the existance of "kmeans2" indicated there was obviously a differnce. After doing som reading I found "kmeans2" to be the algorithm more focused on classifying data into catagories, while "kmeans" was more focused on generating a codebook. kmeans2() returns an ndarray of centroids, and an ndarray relating the ith observation (data point) to the closet centroid.

Next I needed to deal with the "whiten" feature. "whiten"'s documentation states that "Before running k-means, it is beneficial to rescale each feature dimension of the observation set with whitening. Each feature is divided by its standard deviation across all observations to give it unit variance." I was dubious of this claim so I planned to test it out. I also found that one could specify an initial guess of where the centroids would be, or use an integer and let it try and find the centroids. I reasoned that the image averages would be a

good place to guess the centroids (because centroids are already kinda like the averages).

I made the file k_means.py to do my k-means image recognition. Looking at this completed file, you will see it is quite long. The first stage of development I made the methods "ObsAndKInt", "WhitenObsAndKInt", "ObsAndKGuess", and "WhitenObsAndKWhitenGuess". As one could probably get from the names, these encompass the four logical permutation I would like to consider to find the best setup for my k-means file. The only one that might require a little thought is WhitenObsAndKWhitenGuess, which reprsents the logic that I would need to normalize my guesses in the same way my data for them to still work. Formatting proved tricky here as kmeans2() takes in an numpy array, however images are ndarray. This made my code appear a little messy and it takes some thinking to understand.

Now I needed a way to determine which method was best. I made the methods "RoughtErrors()" and "ComplexErrors()". RoughtErrors()" counts the number of points closest to each centroid, the reasons that there should be 55 points in each cluster (because there were 55 images of that letter). If one letter strayed into another letter then it reasons that there is a net difference of 2 away from 55 (because 1 cluster lost a point but another gained it). It then takes the absolute value of the distances from 55 and adds them to make a rough guess at the error. This method does not account for the fact that some letter may drift between eachother (e.g. 25 O's may be in Q cluster and 25 Q's may be in o).

"ComplexErrors()" is unsurprisingly a little more complex. It isolated each block of 55 observations (where the first 55 are A's, the second 55 are B's...). It then finds the most common centroid in this block and uses that as the true centroid for that letter. Then it counts how many are of the observations are not the true centroid in the 55 block and sums this across all blocks.

Now that I had a way to count the errors, I could see how "ObsAndKInt", "WhitenObsAndKInt", "ObsAndKGuess", and "WhitenObsAndKWhitenGuess" preformed. I created the method "testMethods()" (right above the if __name__== "main") to test each method. From multiple tests I found ObsAndKGuess to preform much better than all the rest. I'll list an example of the results.

results directly from my code + slight formatting:

```
Complex Errors: 334 (WhitenObsAndKInt)
Complex Errors: 381 (ObsAndKInt)
Complex Errors: 234 (ObsAndKGuess)
Complex Errors: 282 (WhitenObsAndKGuess)
Rough Errors: 247 (WhitenObsAndKInt)
Rough Errors: 253 (ObsAndKInt)
Rough Errors: 102 (ObsAndKGuess)
Rough Errors: 308 (WhitenObsAndKGuess)
```

Infact, ObsAndKGuess had the very satisfying dictionary of centroid to letter:

```
{0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'E', 5: 'F', 6: 'G', 7: 'H',
8: 'I', 9: 'J', 10: 'K', 11: 'L', 12: 'M', 13: 'N', 14: 'O',
15: 'P', 16: 'Q', 17: 'R', 18: 'S', 19: 'T', 20: 'U', 21: 'V',
22: 'W', 23: 'X', 24: 'Y', 25: 'Z'}
```

Now that I knew ObsAndKGuess was the optimal choice of method, I could begin my image recognition task. This proved to be the hardest part. I developed my first method, "ClassifyTestM1()", to use the centroids and compare an unknown letter's euclidean distance from each centroid. To my dismay all my tests failed to yeild a correct identification. After 2 hours of working, I decieded to attempt a second, much less efficient and logical method.

This second method, "ClassifyTestM2" method was to do a kmeans2 with the unknown image as an observation. Then I could extract the unknown image and find it's cluster. After adding the unknown image and then attempting to reextract it, I would get a different image back. This drove me mad because when I tested on smaller, but identical code, I would get the correct values back. From what I could tell I was putting in a value into an array, then getting a different one back, a result which makes no sense. After an hour or so I gave up of this second method, but I left the code so that one day I might figure out my error.

I then decieded I needed to come back to "ClassifyTestM1()". I then tested "ClassifyTestM1()" and found it returning the the same training data I had used to the wrong letters. This was very confusing. Finally in a moment of clarity I realized my euclidean norm was to blame, because my norm was operating using uint8, which as stated earlier is modular. Finally 5 hours later from when I began debugging "ClassifyTestM1()", I had a working image recognition file.

3.3 results

Results directly from my code:

```
N.png was identified to be: N
G.png was identified to be: S
F.png was identified to be: F
A.png was identified to be: A
M.png was identified to be: M
P.png was identified to be: P
D2.jpg was identified to be: D
Z.png was identified to be: Z
K.png was identified to be: K
H2.png was identified to be: M
S.png was identified to be: S
X.png was identified to be: V
H.png was identified to be: H
E2.jpg was identified to be: E
B2.jpg was identified to be: B
G2.jpg was identified to be: G
R.png was identified to be: R
O.png was identified to be: O
W.png was identified to be: W
B.png was identified to be: H
L.png was identified to be: I
I.png was identified to be: T
Q.png was identified to be: O
A2.jpg was identified to be: A
```

```
C2.jpg was identified to be: C
T.png was identified to be: F
Y.png was identified to be: Y
V.png was identified to be: V
F2.jpg was identified to be: F
C.png was identified to be: C
D.png was identified to be: D
U.png was identified to be: U
E.png was identified to be: E
J.png was identified to be: T
9/34 errors testing (k-means)
```

3.4 Conclusion

k-means did a decent job at classifying the letters. I bet the results could be improved if I removed odd outliars from the training data (if you take a look, you'll find some uncommon writing styles). I would suspect k-means handle larger data slightly better, but would eventually hit a cap at how effective it could be.

4 1D FFT to extract frequiencies and compare to kmeans

4.1 concept

An image can be compressed into a single vector (as I did in k-means). Then we can compute a discrete fourier transform (DFT) of an image (there exists a 2d transform as well, but for the moment I'm going to try a 1D). The DFT breaks a series down into a set of sines and cosines, which each have a different frequency. From the DFT one can examine the power spectrum (which frequencies are most powerful).

One could extract these frequincies from the training data. This would hopefully provide tighter clustering than just a k-means. The hope is that frequency power would be paritioned quite differently for different letters.

4.2 Implementation

I got to start working on this, however I was not able to complete it before this project was due.

5 Other methods I did not get to try

There are a few methods that I did not get to try, but I did learn about a little in some of my research. The first method I did not get to try was the k-means 2d discrete fourier transform. This is similar to the 1D transform but I suspect would have paritioned the data even better. The second I would have like to try would have been using a principle component analysis (PCA). I am still a little unsure about the exact mechanics behind the PCA, but I do know

it returns orthognol matricies, called principle components. From these PCA you can get patterns which may parition the data. Interesting enough there is a relation between k-means clutering and principle components, although much of the stuff relating to PCA's are above my understanding.