CSCI-B 551
Elements of Artificial Intelligence
hrakholi-parishar-rkasture-a4
Assignment-4 Report

Part-1 Implementing the KNN (K-nearest neighbor) algorithms for identifying the image orientation.

Note: Comments about the functions and data structure are included in the source code file (**orient.py**).

Brief Description: For the KNN implementation, the training data is stored as such in the memory. Each test data point is compared to all the points in the training data and the K nearest points are selected to vote on the label of the test data under consideration. The majority label (most frequent label amongst the selected K points) is assigned as the label of the test data point under consideration.

Design Decision(s):

1) **Choice of K**: The program was run on a range of values namely (3,5,7,9,11,21,55,99) to find which value of k results in highest prediction accuracy. For values of K beyond 100, the prediction accuracy was stagnant and so only the odd values between (1-100) were tried. The even value of K was not used to avoid situations of tie breaking (in case there is no clear majority amongst the k points).

   The experiments show that the highest prediction accuracy was obtained at **k = 55**. So, this value of K is now fixed in the source code and will be used for prediction on new test data.

2) **Choice of distance function**: I tried 4 different distance functions namely (Euclidean metric, chess board metric, dimensionality invariant similarity measure (DISM) [1], cosine similarity). However, all of them perform within some small range of each other (in terms of the prediction accuracy). The best prediction accuracy that I got was 71% using the DISM metric while Euclidean gave 70% accuracy on the full test set.

3) **Co-relation study and finding the maximum variance**: In order to identify the features (pixels) of maximum variance a PCA analysis was done on the training data which showed that first 50-80 pixels can explain maximum variance in the data (up to 95%). Thus, a training data of only first 50-80 pixels was created to study if it can improve the prediction accuracy.

   Also, a second co-relation analysis was done to identify which columns relate positively with the target orientation and a selective (20 pixels) were determined. A new training data was created that consisted only these 20 pixels.

   When tested on small and intermediate data (10 and 100 randomly selected data points from the test set), these measures gave very good accuracy (80 and 90% respectively) but fail to scale properly in the full test data set. For testing, the respective number of pixels were selected from the test set, for example: for the PCA approach, only the first 50-80 pixels were used to calculate the dis-similarity (distance) between the training and test set.

   Conclusion: This implies that there are pixels in the test data which don't agree with the variance of the training data and so, those pixels have to be included in the distance calculation too for increasing the prediction accuracy.

   Execution Speed of the program: The initial version of the program took about 25 minutes to complete one round of prediction (on 1 value of K). I optimized the code using vectorized instructions and leveraged native numpy arrays for vector arithmetic. The second version ran in around 5 minutes. Few more optimizations were made to avoid for loops and the final version is running in 25 seconds-2 minutes (including training and testing) on the burrow server.

Results: As evident from the Table-1 below, the highest accuracy **(71.89 %)** is obtained at K=55 (for the full test data set). Whereas, small values of K do better on small data set but as the size of the test data increases, more and more points are classified correctly for **K=55**.

Prediction Accuracy (%)											
Data Set size	K = 3	K = 5	K = 7	K = 9	K = 11	K = 21	K = 55	K = 99	K = 101	K = 151	K = 199
Small subset of test data (10 randomly selected points from the test data)	90	90	90	90	90	80	80	80	80	80	80
Intermediate subset of test data (100 randomly selected points from the test data)	76	77	75	77	81	79	75	79	79	80	80
Full data set of size 943	68.50 646/943 correct	69.03 651/943 correct	69.03 651/943 correct	70.41 664/943 correct	69.98 660/943 correct	(Highest accuracy) ***71.89*** ***678/943 correct***	69.67 657/943 correct	71.68 676/943 correct	71.26 672/943 correct	71.79 677/943 correct	71.36 673/943 correct

Table-1: Prediction accuracy as a function of K and data size.

References: For the DISM metric, [1] "https://www.researchgate.net/publication/264995324_Dimensionality_Invariant_Similarity_Measure"

Part-2 Implementing Adaboost

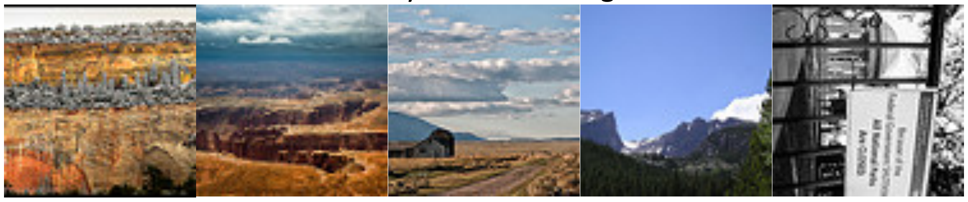
Choice of weak classifiers: We used all possible combinations of pixels pairs (${}^{192}C_2$ combinations) as weak Bayesian classifiers. i.e. if any color value at pixel (x1, y1) compared to color value at pixel (x2, y2) gives the information about the orientation of the image. We didn't want to lose any information contained in images by picking only random combinations. Also, since Adaboost is good at picking good classifiers, we should let it do that work for us. We also added two handpicked classifiers which were relatively better based on intuition. These were comparing blue values(B) and light(R+G+B) in half of an image compared to other half of the image and predicting orientation based on the difference.

Observations:
Below are the accuracies after each iteration of Adaboost. Accuracy doesn't change much in the first 18 iterations because handpicked classifiers are relative stronger compared to other classifiers. Though towards the end, they are slightly improving the accuracy. Our speculation is that as we increase the number of iterations these weak classifiers will gradually improve the accuracy though by very small amounts. Now we didn't have much time to test for large number of iterations since running Adaboost with ${}^{192}C_2$ pixel pairs takes so much of time even for 1 iteration. For each iteration, the time complexity would be $\Theta(N*{}^{192}C_2)$ where N is the number of training images.

No of Iterations	Accuracy
1	69.03%
2	69.03%
3	69.03%
4	69.03%
5	69.03%
6	69.03%
7	69.03%
8	69.03%
9	69.03%
10	69.03%
11	69.03%
12	69.03%
13	69.03%
14	69.03%
15	69.14%
16	69.03%
17	69.03%
18	69.03%
19	69.25%
20	69.25%

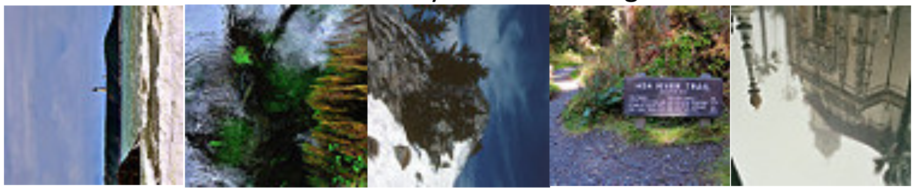
Another interesting observation we came upon is regarding the number of training images. We used 400 images and then 4000 images. And there was drastic improvement in the accuracy. But there wasn't much difference in accuracy when we used 4000 and 37000 images for training.

Below are some of the correctly classified images:



Notice that the top half of the images contain more blue color and light compared to the other half of the images.

Below are some of the incorrectly classified images:



Possible improvements: It was a silly though somehow logical decision to try all possible combinations of pixel pairs. A pair of single pixels is a very weak classifier and would add much to the accuracy compared to the computational time it takes. A better thing would be to use Haar features (used by Viola and Jones for face detection) by taking window of multiple pixels at a time and comparing it to another window of the same size. Neighbor pixels would be collectively a better classifier compared to individual ones. Also, number of these windows would be less than $^{192}C_2$ as we are considering multiple pixels at a time. Thus, it would also improve time complexity.

Part-3 Implementing the Artificial Neural Network (ANN) for image orientation problem.

Brief Description: The fully connected ANN has the following architecture.

- **The input layer**: This is the first layer that just plugs in the input features (the pixel values in this case) to the next hidden layer. In our case, the input layer simply has as many neurons as the number of data points in the training set. So, each example (in the training data) becomes an input for the ANN. For example, the first neuron in the input layer will receive 1*192 as its input. Similarly, there are 36796 neurons (each receiving 1*192) in the input layer.

  This is done so that the code can be vectorized and thus speed is obtained by avoiding the loop to parse each training data separately.

- **The hidden layers**: There are 2 hidden layers in our ANN implementation. The first layer has (192*6) weights and the second layer has (6*6) weights in it. There are 6 neurons each in both the hidden layers. We chose the number of neurons as 6 based on empirical tests as 6 hidden units in each hidden layer gave the best results (in terms of learning faster).

- **The output layer**: The output layer has 4 output neurons (and has 6*4 weights). So, the network outputs a vector of (4*1) which is then used to classify the test example based on which bit of the vector has the highest value. For example, the bits in the vectors are trained to predict the classes in the order 0, 90, 180 and 270. The bit having the highest predicted weight is then assigned as the label for the test data under consideration.

  Activation function: For this task, each neuron is a sigmoid function that predicts the probability of the data point as (0 or 1) based on what activation was given to the neuron. Thus, each neuron in the network is a sigmoid function.

  Feed Forward Pass: The feed forward pass is implemented using vetorized matrix multiplication. For optimizing execution speed, the python lists are converted to numpy arrays.

  Back-Propagation Pass: The back-propagation is implemented by calculating the margin of error at each layer (starting from the output layer) and adjusting the weights on each layer backwards. For implementing the backpropagation, I referred the following sources ([1] [2]). This is equivalent to the gradient descent algorithm where on each back-propagation pass, the goal is to minimize the error between the ideal output and the predicted output.

References

https://stevenmiller888.github.io/mind-how-to-build-a-neural-network/ [1]
https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/ [2]