**REPORT FOR ASSIGNMENT 1 B**

**Name: Aakash Anand Thatte**

**Roll number: 12041590**

For ease of work and to make required data available quickly, I have used a lot of helper functions and stored my data (like image paths on drive, hog features, rgb average features, image names, labels etc). These helper functions and various other code used in the process is present in the data.ipynb file.

**1b\_a: Using LwP(learning with prototype)**

**Notebook name:** LwP\_RGB.ipynb

In this I first tried using the RGB average as a feature. Here what I did was that I computed the Red , Green and Blue average value for each image(it came out as a length 3 vector). Then I calculated the mean of such vectors for each class(mean of all 3 length feature vectors of all images in that class). This I used as the prototype, with whom I computed the distance(cosine and euclidean) of the average rgb vector of each test image.

With this approach, I got an accuracy of around 35 % using cosine distance as a distance metric and around 33 % using euclidean distance.

To give an example of what type of images were misclassified:

The image below(the left one) shows an apple of class “Apple Braeburn” (image name :32\_100.jpg) . But it was classified as Apple Crimson Snow (looks like the right one).

(Note: I couldn’t understand why this happened in case of RGB averages, as both images seem to have sufficiently similar color profile)



**Notebook name:** LwP\_hog.ipynb

Another feature extraction method that I used for LwP was using histogram of oriented gradients. In this, I tweaked the parameters of the hog() function such that I do not get a very long feature vector. Finally, I settled for a 256 length vector. Then I calculated this vector for all training and test images and calculated the average(prototype) in a similar way as the RGB prototype. Using this method, the accuracy improved a little bit. I got an accuracy of about 42 % using LwP with hog.

To give an example of what type of images were misclassified:

Apple Braeburn/3\_100.jpg (left) was misclassified as Apple golden 3(right)





The two images being very different to see, it was very surprising as to why the left one got classified as being of the class of the right one.

According to me, as HOG captures the edges and their directions(gradients), in a way, it captures the shape from an image. So I think that this was the reason for the image to get misclassified. A combined model, that is, one which uses RGB model and HOG model both, will probably solve this problem as it will consider both factors, color of image and shape of image.

**Notebook Name:** Combined\_LwP.ipynb

Then I also tried using a combined approach using the above two approaches. In this I found the 3 closest images that each of them gave(similar to KNN using above two features with k as 3) and then returned the most frequent class from the total 6 images received as nearest. But this approach failed badly as it correctly classified only 157 of the total 22,688 test images. I would like to discuss this (with TA or Instructor) in terms of why this one (combined approach) failed and why usually such grouped approaches work well(as told in class).

**1b\_b: Using K Nearest Neighbors:**

**Notebook name:** KNN\_hog.ipynb

For the KNN approach, I decided to use only the hog 256 feature vector as the feature and tried two three approaches to implement the KNN algorithm.

Initially I wrote a very simple and straightforward code which compared a test image feature vector with all training image feature vectors and found out the nearest neighbors and thus classified the image. This being a very naive code, it was taking around 14 seconds to finish for 1 training image. As we had 22,688 test images, I did not run this on the entire testing data( because it would then take around 88 hours to finish). Rather I selected 50 images, and it gave a 70 % accuracy.

To improve the running time of this approach, I later used the **KDTree** from the scipy library and it improved the running time substantially. But due to the large RAM requirement in this approach, the session of Google Colab (which gives a 12 Gb RAM) used to get crashed whenever I ran this on whole training data.

Then I finally found out something really interesting and useful. It was the **scipy.spatial.distance.cdist()** function. In this function, the python code calls the C code and hence it is way faster than pure python code.

Then I found the np.argpartition() function to finally give indices of the k nearest neighbors.

[Note:**numpy.argpartition()** function is used to create an indirect partitioned copy of input array with its elements rearranged in such a way that the value of the element in k-th position is in the position it would be in a sorted array.]

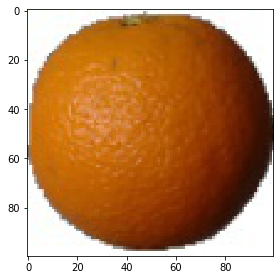
This way I could finally run and test the KNN algorithm on all the training data (67,692 images) and all the test data (22,688 images). This also gave an accuracy of 70 % and this finished running in a very short time.

Example of a misclassified image:

On the left: Test/Apple Braeburn/60\_100.jpg On the right: Test/Orange/227\_100.jpg

(Note: I was unable to figure out this as well. The images being different, still got classified as the same. And to my surprise, all 3 neighbors for the left image were oranges.)





(Note: Other implementation details can be found in the notebooks)

**1b\_c: Using Neural Network (took help from internet and the starter code provided)**

I have taken help from the starter code provided by sir and some websites to implement this one. I faced some problems initially to pass training data to the neural network, so I have copied some code from a github link, but made sure to understand the code. This method gives around 6 % accuracy when run on 5 classes. When run on whole training data, the colab notebook crashes and so I was unable to run the neural network on entire dataset.

During implementation, I tried to tweak the parameters like the activation function and such things.