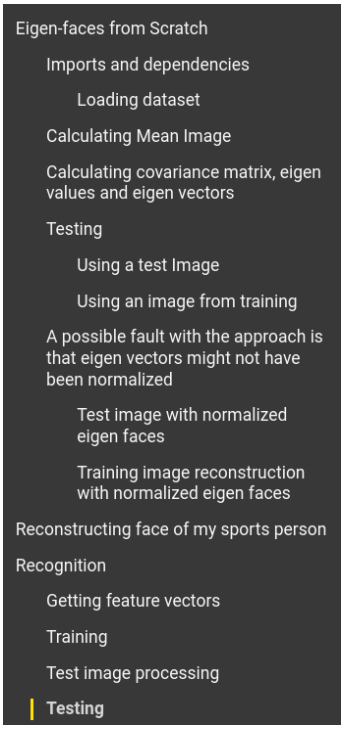
Computer Vision Assignment-3

Abu Shahid (B20CS003)

▼ Eigenfaces from Scratch

Link to collab file: here

Code structure is:



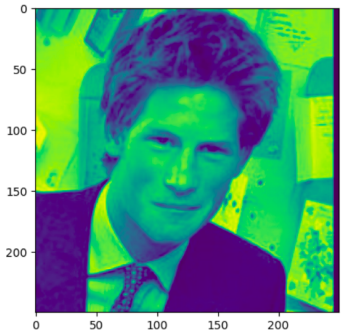
Task:

Use the subset of the LFW dataset provided with this assignment, include 1 face photograph of your favorite Indian sportsperson from the web to augment the dataset, and implement Eigen face recognition from scratch. You may use the PCA library, but other functionalities should be originally written. Show top-K Eigen's faces of the favorite Indian sportsperson you considered for different values of K. The report should also contain a detailed quantitative and qualitative analysis.

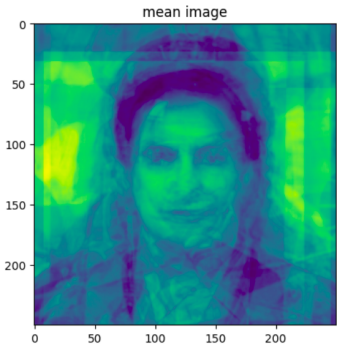
For my choice of favorite Indian Sportsperson, I chose Chess grandmaster Vishy Anand.

Steps involved in Implementing Eigen faces from scratch are:

1. Loading dataset
    a. For my implementation, I used the following libraries: `os`, `matplotlib`, `cv2`, `numpy`, `sklearn.preprocessing`
    b. I had 11 images in my train_set, all with the shape (250, 250, 3)



    c. All of them were converted to grayscale and flattened into vectors of size(62500, 1)
2. Calculating Mean Image
    a. For calculating the mean, all the train image vectors were added along the 0th axis and the resultant vector was divided by the total number of images.



3. Calculating Covariance matrix, Eigenvectors, and Eigen Values
    a. After calculating the mean image, we calculate Del
    b. All the delta images are then stacked, let's call it D. It is then multiplied with its transpose DT to give the covariance matrix.
        i. Calculating the eigenvectors of this matrix is unreasonable as its size is (62500, 62500) so we rather do the SVD of `DT.dot(D)` instead of `D.dot(DT)`
        ii. We get the original vectors by multiplying the vectors from the above operation with D.

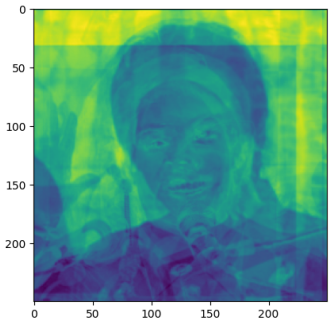
iii. The formal form of this is being taken as a given fact.

c. To calculate the eigenvectors and values, we use `np.linalg.eig`

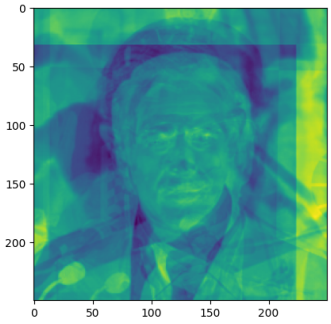
d. Eigenvalues are sorted in descending order and corresponding to then, respective eigenvectors are re-arranged.

e. Eigenvectors are then multiplied with matrix `D` to give eigenvectors of `D.dot(DT).`

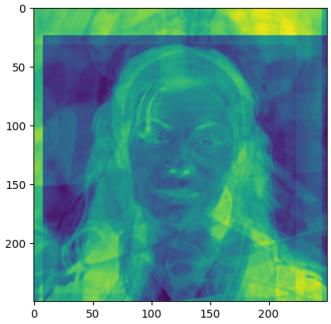
f. Following are the top 4 eigenfaces.



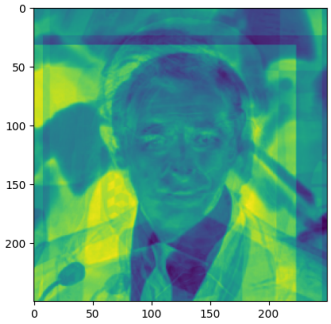
Eigenface1



Eigenface 2



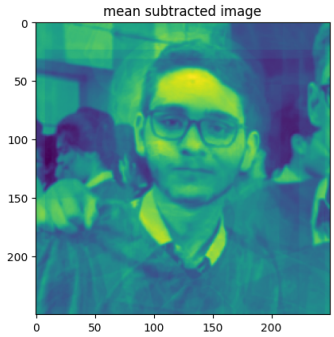
Eigenface 3



Eigenface 4

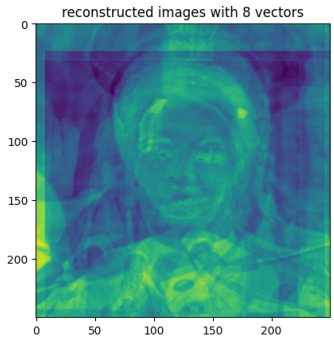
4. Testing and reconstruction

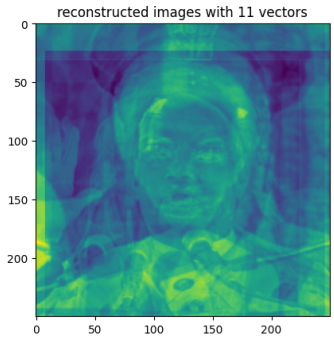
   For initial testing and analysis, instead of jumping to the image of sportsperson, I chose another image. I took my own image and tried to do its reconstruction

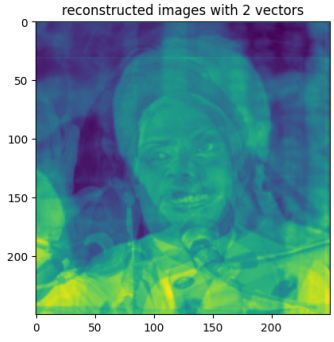


To perform reconstruction, I calculated Omega matrix, which is a list of dot products of `Eigenfaces` with the `mean subtracted image`

Here are the results

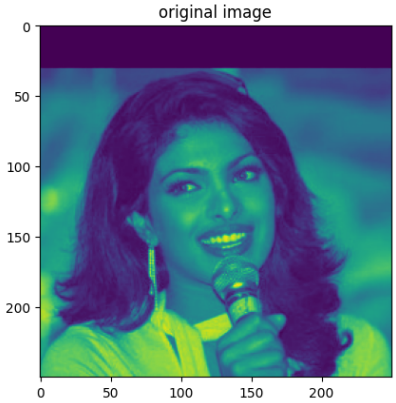


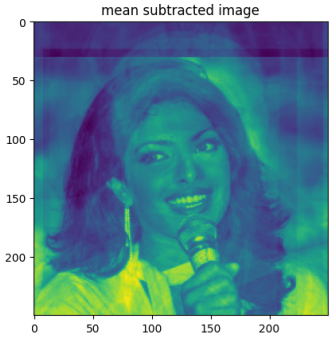


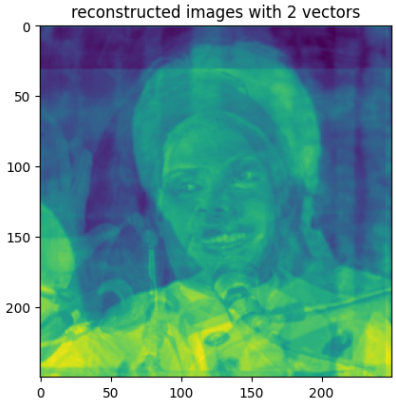


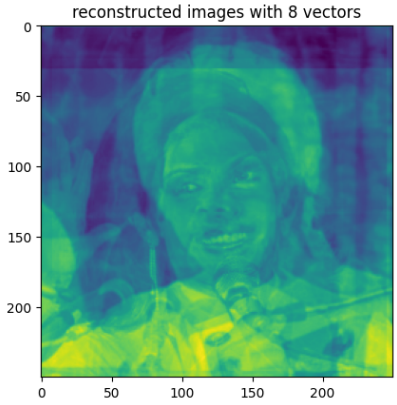
Clearly, the reconstructions are far from original image. There can be lots of reasons for this.

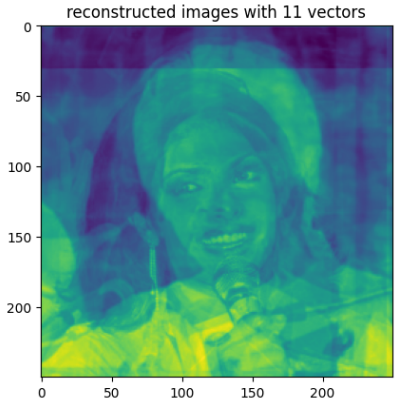
- The mean image is not representative of the image population. This is a very valid assumption considering we only had 11 images in our dataset for creating the mean image. In fact, mean_image is full of artifacts that are not representative of the face at all. A better way to evaluate can be by picking an image that is part of the training population. So next, I tried to do the reconstruction an image that is part of my training data.









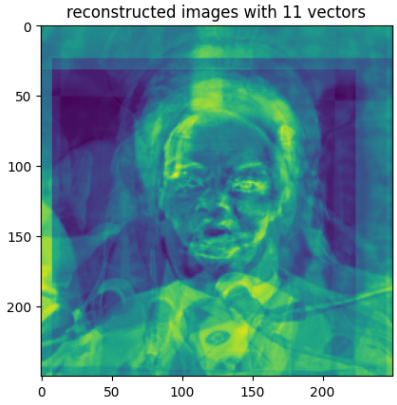


While still not perfect, we can at least make out it is `Priyanka Chopra` from the image. Therefore our assumption about the mean_image not being representative of all the images is not completely false.

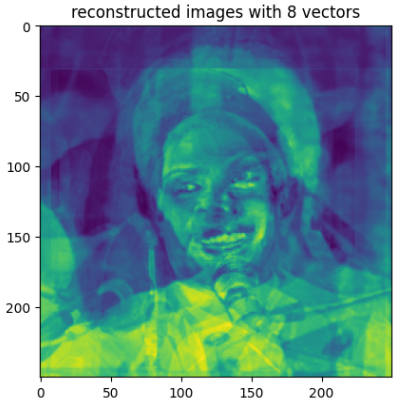
Further room for improvement??

Another possible fault with the approach is that eigen vectors might not have been normalized. So this time on, the eigenfaces were normalized and reconstruction was attempted again.

While no improvement was seen for the image from outside the training set; incremental clarity was gained in the case of images taken from the training set.

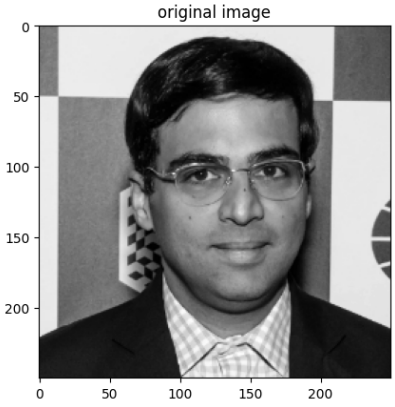


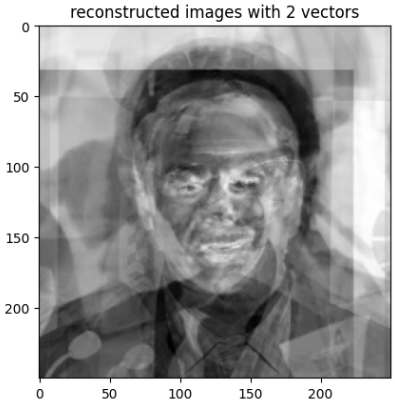
My image

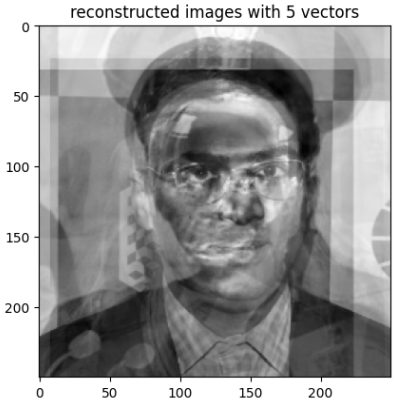


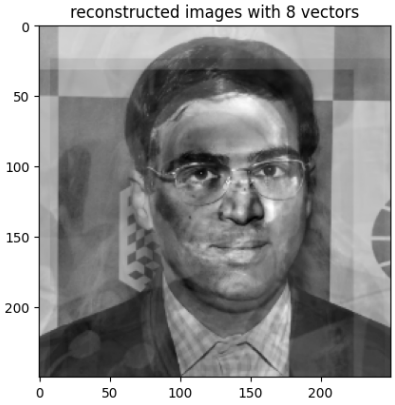
Priyanka Chopra

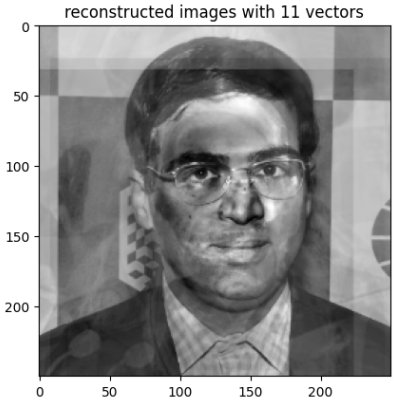
- Reconstructing Vishy Anand





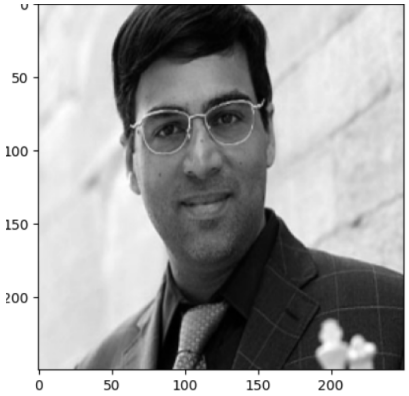




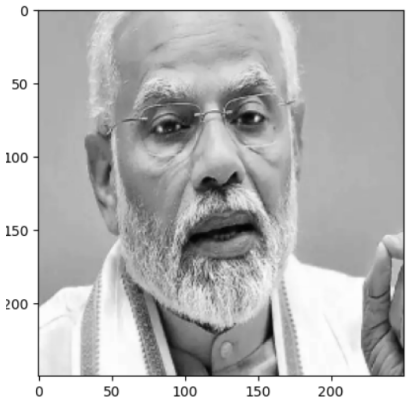


For recognition and testing task, following steps were taken:

1. Project each mean-subtracted training image onto the space of eigenfaces to get a set of feature vectors. Each feature vector is a set of weights that describes the image in terms of the eigenfaces.
2. Train a classifier, such as a Support Vector Machine (SVM), using the feature vectors and their corresponding labels (i.e. the identity of the person in the image).
    a. here, I used a `poly` kernel with degree= `3`
3. Collect a set of face images for testing and create a testing set.
    a. Here, I simplified the classifier that tells if a given image is `vishy` or `not vishy`.
    b. And, to demonstrate my results, I took the following images (both images were reshaped to 250*250 and converted to gray scale)



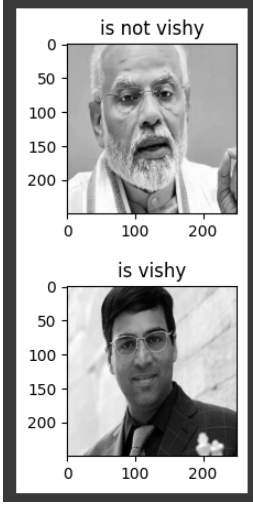
vishy.png



not_vishy.png

4. For each test image, subtract the mean face and project it onto the space of eigenfaces to get a feature vector.
5. Use the trained classifier to predict the identity of the person in the test image based on its feature vector.
    a. A prediction of `1` means that image is of vishy and `0` means its not.

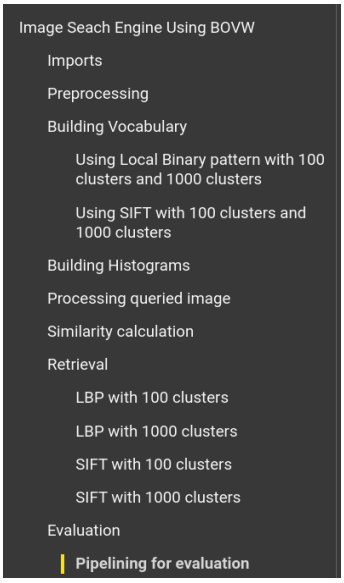
```
pred_1 =  clf.predict(feature_test_1)[0]
pred_2 =  clf.predict(feature_test_2)[0]
vishy_not_vishy = ['is not vishy', 'is vishy']
plt.figure(figsize= (2,2) )
plt.imshow( test_1, cmap='gray')
plt.title(vishy_not_vishy[pred_1])
plt.show()
plt.figure(figsize= (2,2) )
plt.imshow( test_2, cmap='gray')
plt.title(vishy_not_vishy[pred_2])
plt.show()
```



▼ Visual BoW

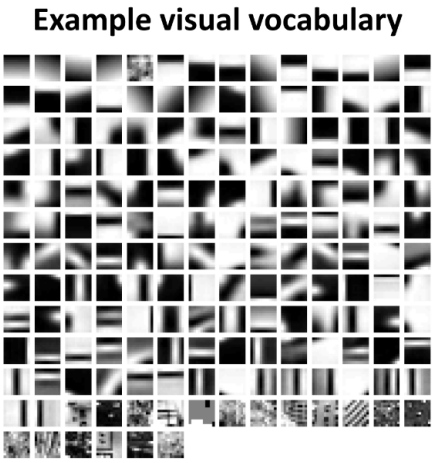
Link to collab file: here

The code structure is as follows:



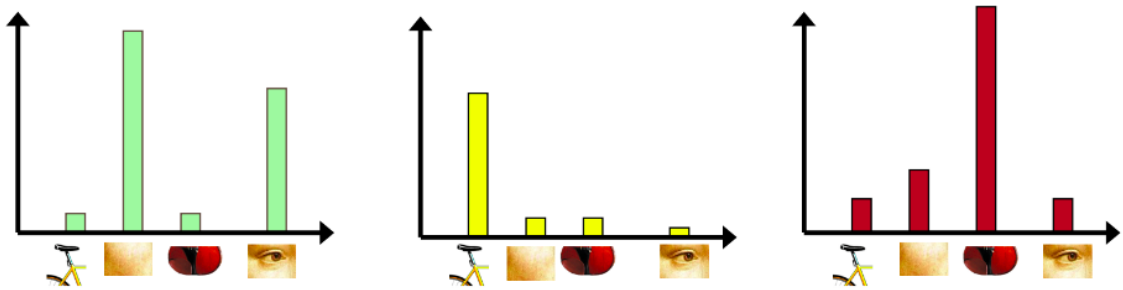
Steps involved:

1. Preprocessing: The first step is to preprocess the CIFAR-10 dataset. Images were already given to be 32x32 pixels. Preprocessing involved converting them to grayscale. We also need to extract local features from the images using a feature detector such as SIFT or SURF. This step also involved normalizing the images.
2. Building Visual Vocabulary: Next, we need to build a visual vocabulary using clustering algorithms such as K-means. We can use a subset of the training images to extract local features and cluster them into visual words.
    a. We used a subset of training data to extract local features which can then be clustered to form visual words.
    b. For my purpose, I used `10000` training images to extract visual words and make my vocabulary.
    c. We had a huge list of feature descriptors to pick from not knowing so may perform better than others. I, therefore, chose two feature detectors to work my way out
        i. SIFT: Scale Invariant Feature Transform
        ii. LBP: Local Binary Pattern [here]. I chose LBP for its lightweight
    d. Having found the features, they needed to be clustered. Again, I took the liberty to train 2 variations of the model, as we are unaware of the feature diversity of the data. I trained two KMeans models, one with 100 clusters and other with 1000 clusters.
    e. I now had 4 models
        i. LBP with 100 clusters
        ii. LBP with 1000 clusters
        iii. SIFT with 100 clusters
        iv. SIFT with 1000 clusters



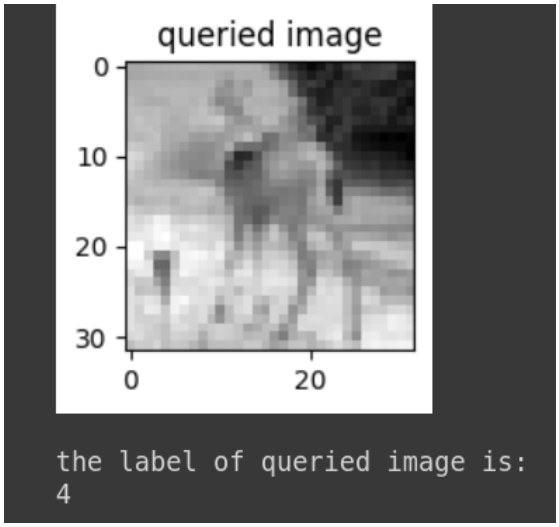
here

3. Building Histograms: For each image in the dataset, we need to build a histogram of visual words. This involves assigning each local feature in the image to the nearest visual word and counting the number of times each visual word appears in the image.
    a. This was again done 4 times over, once for each model.

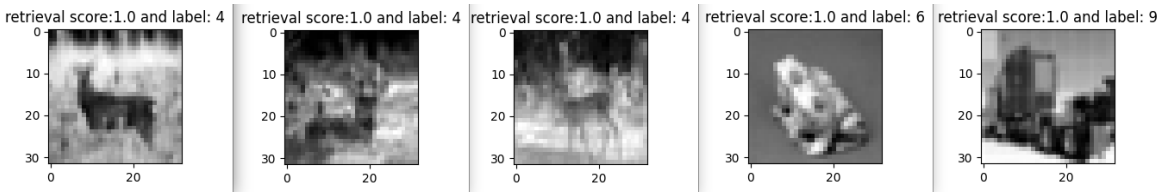


here

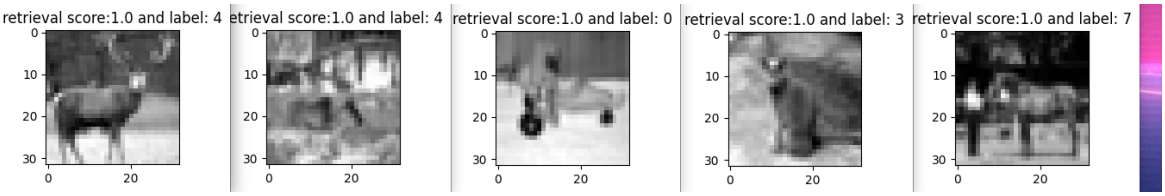
4. Query Image Processing: When a query image is submitted, we need to extract its local features, assign them to visual words, and build a histogram of visual words for the query image.
5. Similarity Calculation: We can calculate the similarity between the query image and each image in the dataset by comparing their histograms of visual words. One common method is to use the chi-squared distance or cosine similarity between the histograms.
    a. For LBP based models, I used chi-squared distance approach
    b. For SIFT based models, I used cosine similarity metric.
6. Retrieval of Top-5 Similar Images: Finally, we can retrieve the top-5 similar images to the query image by sorting the images in the dataset based on their similarity scores and returning the top-5 images.
    a. Let us now look at our results.



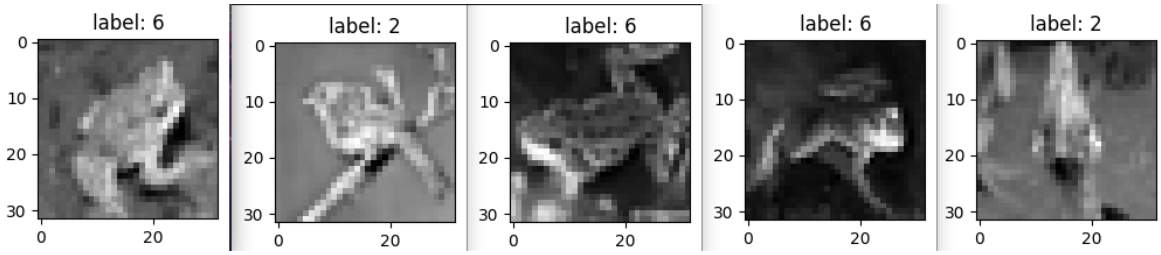
Results of LBP with 100 clusters



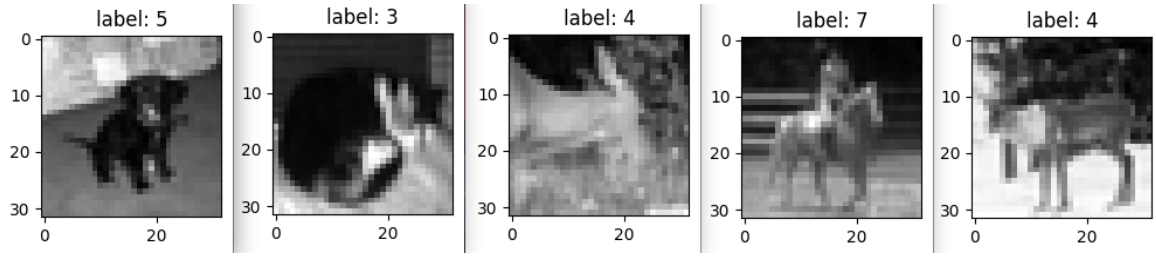
Results of LBP with 1000 clusters



Results of SIFT with 100 clusters



Result of SIFT with 1000 clusters



Looking at the above results, where ground truth was label 4, we can see `local binary pattern` models performing better than `SIFT` model. Therefore, I will be showing further analysis solely using `LBP`.

Even in `LBP`, I will concern myself with 100 cluster-center model because of its simplicity (PS: it even performs better than 1000 cluster model; although the above example might not be enough to make an inference about the dataset or the models themself)

Evaluation

My code until now was very modular as it was supposed to implement one step at a time. I finally pipeline my code using `LBP with 100 clusters` model.

- `get_labels()` takes an integer as an argument and returns the ground_truth_labels_list and prediciton_labels_list.
  - Lets say, if I pass `get_labels(100)`, it returns `gt_labels` and `pred_labels`. This means, ith image in test-set has a label of `gt_labels[i]` . Moreover, when image retrieval on i-th image is done, 5 images that we get have `pred_labels[i]`. Shape of `gt_labels` is (100,1) while that of `pred_labels` is (100,5)

`get_metrics():` takes `gt_labels` and `pred_labels` and k

  - k here is an integer parameter, signifying the number of relevant retrievals that an end user want to concern with. In a real world scenario where that are `n` retrievals, one may only concern with `k`<n top retrievals.
  - It calculates precision and recall for all the test-images using the following logic.

```
for i in range(len(gt_labels)):
        relevant = sum([gt_labels[i] == l for l in pred_labels[i][:k]])
        tp = relevant
        fp = k - relevant
        fn = 1 if relevant == 0 else 0

        precision = tp / (tp + fp)
        recall = tp / (tp + fn)
```

  - Precision, recall so calculated are for each individual test-image. They are then appended and returned as `precision_list`, `recall_list`
  - Mean precision for 100 test images with `k`=3 gave following results:

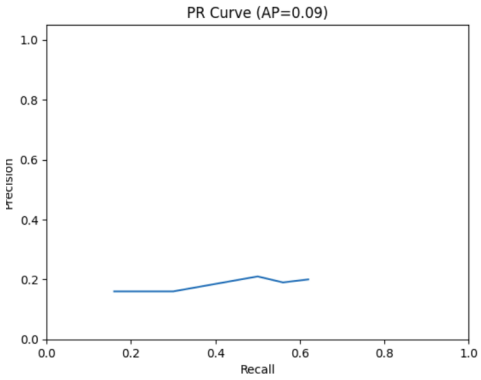
```
print(precision_at_3)
print(recall_at_3)

0.18929166666666666
0.42075
```

- I calculated mean precision and mean-recall for different values of `k` and had the following results.

```
when considering 1 top images among 5 retireved images; precision: 0.18175; recall: 0.18175
when considering 2 top images among 5 retireved images; precision: 0.186125; recall: 0.313625
when considering 3 top images among 5 retireved images; precision: 0.18929166666666666; recall: 0.42075
when considering 4 top images among 5 retireved images; precision: 0.1853125; recall: 0.51075
when considering 5 top images among 5 retireved images; precision: 0.18619999999999998; recall: 0.572125
```

This data was in turn used to plot the precision-recall curve.



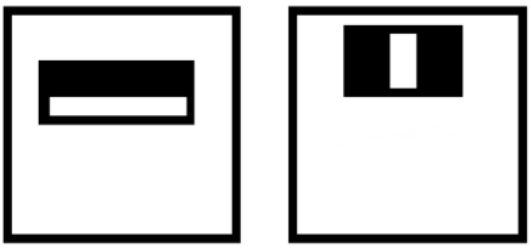
Note the sparsity of the curve is due the reason that we were confining our analysis only to the top 5 retrieved images. This gave us only 5 different values of thresholds and therefore 5 pairs of precision and recall.

This can easily be extended to have a denser P-R curve.

▼ Viola-Jones Face detection

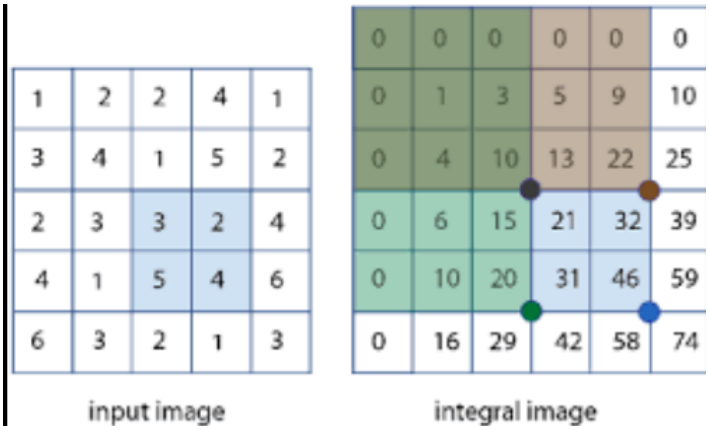
A popular and effective way to find faces in digital images is the Viola-Jones face detection algorithm. Here are the steps involved in the algorithm:

1. Haar Feature Selection: The first step is to select a set of Haar-like features that can help detect faces. These features are rectangular patterns that represent contrast differences in the image.
    a. Haar features are then convolved with a given region of the image and a score is calculated by it. Convolving Haar features in itself is compute expensive. Doing this for different sizes and for different regions of the image drives up the cost even further. Therefore, we use Integral Image.

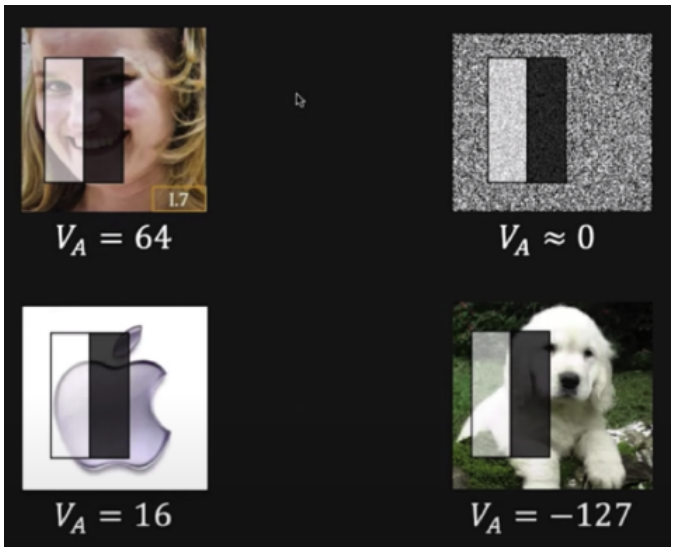


link

2. Integral Image Computation: The integral image of the input image is computed to speed up the Haar feature calculation. The integral image is a sum of pixel values from the top-left corner to the current pixel in the input image.



3. Training: These scores are annotated and fed to a classification machine learning model like SVM. The model then gives us a threshold. If the score of the test image is more than the threshold, it is likely to be a face else it is not.



4. Sliding Window Detection: The image is scanned with a sliding window to detect possible face regions. The Haar features are computed for each window and evaluated using the strong classifier. If the features meet a certain threshold, the window is classified as a face.
5. Non-Maximum Suppression: After the sliding window detection, multiple windows may be detected for a single face. Non-maximum suppression is applied to remove overlapping windows and keep only the most probable face detection.
    a. Face Region Refinement: Once a face is detected, the face region can be refined by using the Viola-Jones algorithm to locate the eyes, nose, and mouth. The face region is then adjusted to fit these features.

To further improve the Algorithm, we can use:

- Adaboost Training: Adaboost is a machine learning algorithm used to select the most important Haar features for detecting faces. Adaboost selects a subset of features that can best classify faces from non-faces in the training set.
- Cascading Classifiers: The AdaBoost algorithm generates multiple weak classifiers that are combined into a strong classifier using a cascading approach. Each weak classifier is designed to reject non-face images quickly, which reduces the overall computation time.

▼ Sliding window object detection using HOG

Link to collab : here

Code structure is as follows

Sliding window object detection using HOG

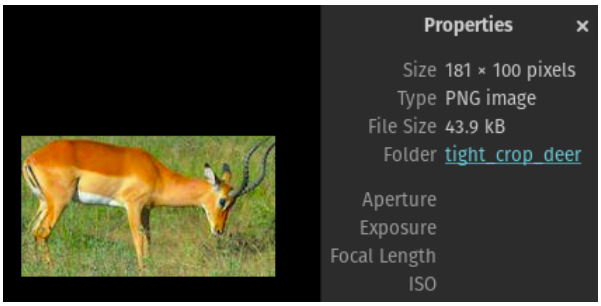
- Imports
- Compute HOG features
- Build an SVM classifier
- Sliding window object detection
- Non-maximum suppression

Task:

You are given a few deer train images with this assignment. Manually crop them to find out tight bounding boxes for Deer and also obtain some non-deer image patches of different sizes and aspect ratios. Compute HOG features for deer and non-deer image patches and build an SVM classifier to classify deer vs non-deer. Now, implement a sliding window object detection to find out deer in the test images. Write down each step in the report. Also, objectively evaluate your detection performance.

Steps taken:

1. Manually crop deer images: Each images was opened each image and manually cropped to include only the deer and save the cropped image. This will create a dataset of cropped deer images.
2. Obtain non-deer image patches: Similarly, I obtained non-deer image patches of different sizes and aspect ratios. These can be obtained by randomly selecting patches from images that do not contain any deer.
3. Compute HOG features: Next I computed HOG features for each image patch. HOG stands for Histogram of Oriented Gradients and is a popular feature descriptor used in object detection.
    a. I used the `hog()` function from the `skimage.feature` module in Python to compute the HOG features. with following parameters
        i. `orientations`: This parameter sets the number of orientation bins used to compute the histogram of gradients. It determines the granularity of the histogram and how well the HOG descriptor captures the directional information in the image. I took the value of `orientations` as 8.
        ii. `pixels_per_cell`: This parameter sets the size of the cell over which the histogram of gradients is computed. Larger cells capture more global information, while smaller cells capture more local information. A good rule of thumb is to choose a cell size that is roughly half the size of the smallest object you want to detect.
            1. Keeping this in mind, I analysed my train_set and the smallest image I found was of size `181*100` so I took the value of `pixels_per_cell` to be `(50,50)`



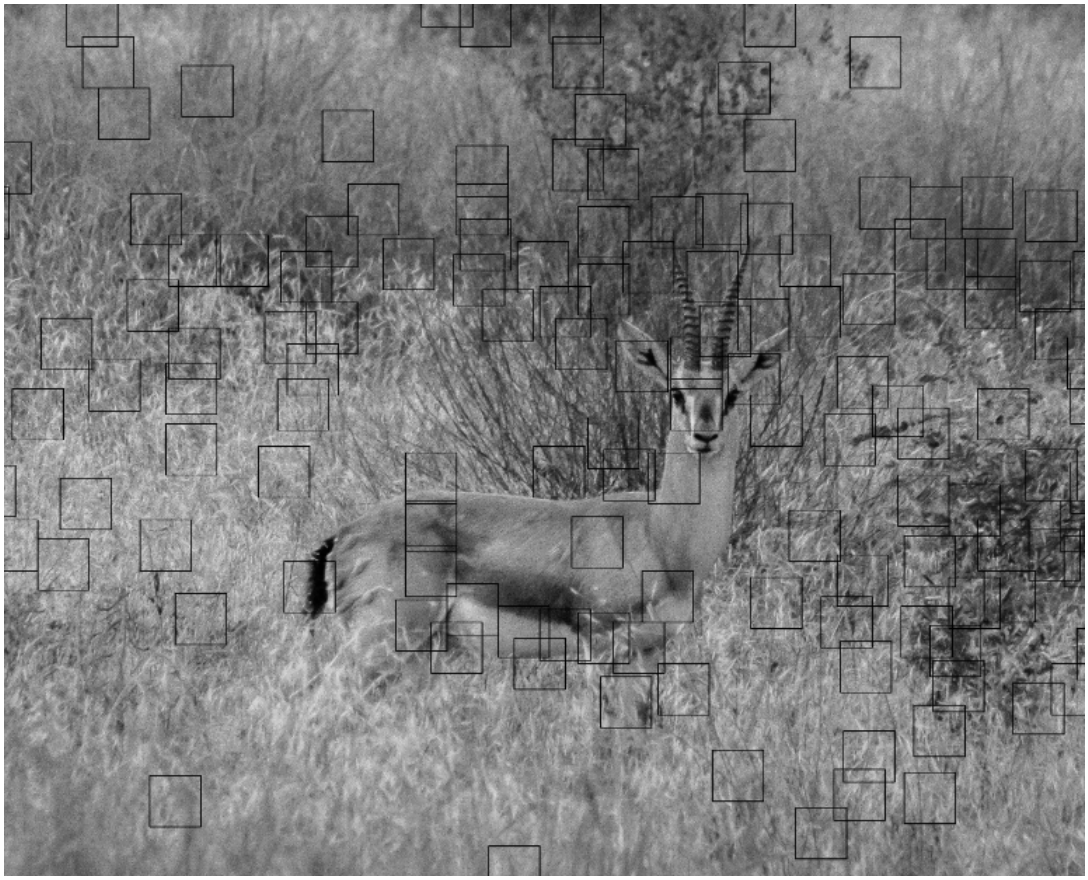
- `cells_per_block`: This parameter sets the number of cells per block over which normalization is applied. Normalization helps to account for variations in lighting and contrast, and improves the robustness of the HOG descriptor.

- `visualize`: This parameter determines whether or not to generate a visual representation of the HOG descriptor, which can be useful for debugging purposes.

4. Build an SVM classifier: Useing the computed HOG features I built an SVM classifier to classify deer vs non-deer. You can use the `SVC()` function from the `sklearn.svm` module in Python to build the SVM classifier.
5. Sliding window object detection: I used the SVM classifier to detect deer in the test images using a sliding window approach. This involves sliding a window of a fixed size across the test image and classifying each window as either deer or non-deer based on the SVM classifier.
6. Non-maximum suppression: After applying the sliding window approach, there may be multiple detections of the same deer in close proximity. To remove these duplicate detections, non-maximum suppression (NMS) was applied to the detected bounding boxes.

Analysis

With the classifier train, we not turned to test_set. With no baseline in place, there is no actual way of telling, but I in my personal opinion feel that the classifier was performing rather poorly.



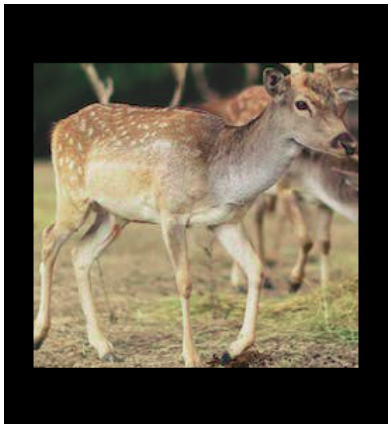
Clearly, the classifier is not up to the mark.

Results with other images were no better

Reasoning:

- Training data is very sparse: It might very much be possible that 11 images are not enough to learn the latent features, Although, I tuned the classifer to use a `poly` kernel with degree `3`, enough features could not be learned.
- Next, test_set is not representative of training data. While all the images in train_data have dimensions not more than `(600,500)` with the actual deer being localizable in `(400,400)`, the images in test_set are `(4000,2000)` on average with deer roughly localizable in `(1000,1000)`. Therefore the HOG parameters for detecting deer on train_set_population will be different from that on test_set_population
    - Another reasoning could be inclusion of `non-deer` patches in the tight-deer crop. To test this, I made another set of manually cropped deer images, but this time excluding their horns so as to keep the non-deer pixels in the crop to a minimal.
    - Please find attached both the variations of the crop as part of the assignment.

tight crop



tighter crop

Steps from 1 to 7 were again repeated. The model we got this time was more constrained.

While model trained with horns gave 2174 detections of a patch having probability higher than 60% of being a deer, model trained without horns gave only 18.

```
print(len(detections))
print(len(detections_nh))
```

```
2174
18
```

(Above number is before NMS)

Result of model trained without horns

Clearly, the result is better in terms of way fewer `false_positives`. The model however failed to

- Another reason could be occlusion and hyper-animation. For example, even the `no_horns` model failed to give any good results with the following image.