HOMEWORK #3

Issued: 10/7/2016 **Due: 11/6/2016**

By: Hariprasad Ravishankar **USC ID: 8991379333**

Problem 1: Texture Analysis and Segmentation

1.1. Abstract and Motivation

Images of Rocks, Grass, Straws, Weave, and Sand have an underlying texture that is unique in their pattern and structure. This problem tries to identify, classify and segment different textures using 25, 5x5 Laws filter. Each laws filter captures one unique feature of every texture and gives a unique feature map. These features might be, edges, low frequencies, spots, waves, or ripples. Since different textures have differing parents of the aforementioned features, these can be identified. The energy of the feature map produced by each of the 25 filters gives a metric to evaluate and classify pictures of different texture images. Further, Principal Component Analysis is used for dimensionality reduction and K-means clustering is used for unsupervised classification task.

In the Segmentation problem, the objective is to use the 5x5 laws filter to locally segment different textures within a single image. Features are extracted for every pixel by computing energy inside a window, centered at that pixel. This is repeated for each of the 25 feature maps. Unsupervised clustering is performed using K-means to label each pixel into one of 'n' different textures.

1.2. Approach and Procedure

Texture Classification

Removal of DC component

The first step in this approach is to remove the DC component from the image. Intensity difference between different images might give false results. The mean of all pixels in an image can be approximated as the DC value. This value is subtracted from each pixel in the image. cv::mean() is used to compute the mean of all pixels in a given image.

Computing Law's Filters

The first step involves constructing the 5x5 Laws Filters from the 1D Kernels. This is achieved as follows. Each of the 1D kernels, i.e. L5 (Level), E5 (Edge), S5 (Spot), W5 (Wave) and R5(Ripple) are linear vectors. Thus, A^TA will give us a 5x5 filter where A represents any of the 1D Kernels.

In all, there can be $5^2 = 25$ combinations of 5x5 filters, each capturing one unique feature of the texture. Each of the 1D kernels are stored in a 2D float vector as rows. A matrix vector stores the 25, 5x5 filters by calculating A^TA, using the .t() function that gives the transpose of a matrix in OpenCV.

Filtering

Once the 25 filters are constructed, the function *PerformFiltering()* is called, which takes as arguments, each of the 12 Texture Images, and each of the 25 Laws filters, one by one inside a nested loop. The *PerformFiltering()* function convolves the 5x5 Filter, supplied as argument to the Texture Image.

The *copyMakeBorder()* is first called to extend the boundary by 2 pixels on all 4 edges to center the filter on the first pixel of the original texture image. The *copyMakeBorder()* is an inbuilt OpenCV function that takes as argument the intended extension of the image, and type of padding. BORDER_REFLECT_101 was chosen to extend the image by reflecting upon the edge.

Convolution

The convolution operation is performed by multiplying and summing the 5x5 filter kernels with 5x5 matrix of pixels from the top-left corner of the padded image. The center pixel of 5x5 window is then replaced by the sum. The 5x5 matrix window then slides with 1 pixel stride and the process is repeated.

The *PerformFiltering()* function thus returns a filtered image, also referred as the feature map, produced after convolving one of the 25, 5 x5 filter with one Texture image.

Energy Calculation

Energy of a feature map F is computed using the equation

$$\frac{1}{m*n}\sum_i^m \sum_j^n F(i,j) * F(i,j) \ ,$$

where m refers to the height (number of rows) and n refers to the number of columns in the feature map.

The feature map returned by the *PerformFiltering* function is stored in a Mat file called dst. Thus, Energy is calculated by computing *sum(dst.mul(dst))/ (size*size).* Here *.mul()* is an element-wise multiplication operation in OpenCV and sum() computes the sum of all elements in the matrix and is defined in OpenCV.

The computed energy in each of the 25 feature maps for all 12 input images are stored in a 2D float array called *feature_vector*. Thus a 12 (rows) x 25 (column) feature vector is obtained.

Normalizing features

Some features have higher energy values than others, and this might lead to scaling problems while applying PCA. Thus the features are normalized prior to Dimensionality reduction. It was further noticed that Normalizing features using the response of $L5^TL5$ filter did not give good results, thus it was replaced by 0-1 normalizing procedure.
Normalization is performed by first computing the column-wise minimum and maximum for each of the 25 Dimensions in the feature_vector. This is computed using the cv::reduce() function in OpenCV that computes the column-wise minimum and maximum of a matrix and returns it as a linear vector.

Next, for every sample (row), the feature values are normalized using the formula

$$X' = \frac{X - X_{min}}{X_{max} - X_{min}}$$

Principal Component Analysis

PCA is a dimensionality reduction procedure that is used to reduce feature dimensions to a given number of dimensions by evaluating the largest magnitude eigen vectors of the covariance matrix corresponding to the feature matrix. The detailed discussion of PCA is explained in Section 1.4.

PCA is implemented in this problem to reduce the dimensions from 25 to 3 by invoking the PCA object in the OpenCV. The PCA function creates an object that takes as parameters the feature matrix, type pf returned matrix, how data is arranged in the feature matrix, and the intended number of returned dimensions.

PCA pca_analysis(features, Mat(), CV_PCA_DATA_AS_ROW,3);

- Thus an object pca_analysis is created.
- 'features' is the feature matrix obtained after normalization.
- The Mat() argument specifies that covariance needs to be evaluated
- CV_PCA_DATA_AS_ROW specifies that the 'feature' matrix has its samples as rows.
- 3 refers to the reduced dimensionality

In order to obtain the transformed features, the . *project()* function is invoked that gives the *output_features* containing the same number of rows but 3 columns

pca_analysis.project(features,output_features);

K – Means Clustering

K- Means clustering is an unsupervised clustering algorithm to predict labels and classify 'n' different textures. A detailed discussion of the K-means algorithms is explained in Section 1.4.

For the Texture Classification problem, the number of clusters is known a proiri and is equal to 4 for each kind of texture Rock, Sand, Weave and Grass. The K- means function thus, takes as arguments, the 12 x 3 feature vector, number of clusters, Termination Criteria, number of attempts, and initialization of the cluster centroids. The function then returns a linear vector of labels, for each of the 12 samples, and another matrix called centers that returns centers of the 4 cluster centroids.

kmeans(output_features, 4, labels,

TermCriteria(CV_TERMCRIT_ITER|CV_TERMCRIT_EPS, 30000, 0.00001),

10, KMEANS_RANDOM_CENTERS, centers);

kmeans() function is described under the cv:: namespace and is included in OpenCV.

The termination criteria determine when the algorithm terminates and returns the labels. Here the termination criterion is determined by CV_TERMCRIT_ITER|CV_TERMCRIT_EPS, which means that the algorithm will terminate when either of the two conditions are first met. i.e, after 30,000 iterations or if the change between the centroid coordinates is less than 0.00001.

The argument 10 refers to the number of times the K-means algorithm is rerun with new initialization of the centroids. This is because it is possible that with different initial centroid parameters, different optimum clusters might be computed which may not be the same and some may not represent the best clusters. The most optimum cluster is one in which the average distance between the samples and the centroids are the least among all 10 attempts/trials.

KMEANS_RANDOM_CENTERS initializes the centroids randomly in the 3 dimensional space.

Plotting 3-D feature vector in the feature space

In the Texture Classification task, the output of the PCA function gives a feature matrix of reduced dimension, i.e a 12 x 3 matrix, where the columns correspond to each feature dimension. This matrix is exported to MATLAB as matrix 'x' containing 12 rows and 3 columns.

The labels produces by kmeans() function is also exported to MATLAB and stored as a linear vector 'c'

The scatter3() MATLAB function is used to produce a scatter plot of the feature matrix in 3 dimensional space, and each of these points are colored based on the label vector passed as argument 'c'. The function command to produce the scatter plot is

>>scatter3(x(:,1),x(:,2),x(:,3),30,c,'filled');

The above command produces a scatter plot with X, Y and Z axis as first, second and third feature dimension of feature matrix 'x', each point is of size =30 and is circular and filled. The 'c' is a vector with the same number of rows as 'x' and contains the label of each of the samples in 'x'.

Texture Segmentation

For texture segmentation task, all the aforementioned procedures are followed in order with a few modifications.

Firstly, only one image is read from file, and all 25 laws filters are convolved separately giving rise to 25 feature maps.

For **Energy Computation**, instead of computing the energy of each feature map as a whole, energy is computed for each pixel from a neighborhood window of size W. If the dimension of a feature map is *m x n*, and we have 25 such feature maps, we thus end up with a feature vector of dimension (*m x n*) rows and 25 columns.

Thus, the *GetFeatures()* function takes as arguments a feature map, and a window size W and returns an energy vector that is a linear array of length (*m x n*) containing the energy feature for every pixel in raster scan approach. Thus for every pixel j, its energy within a W x W window is computed and is pushed back into the energy vector.

The GetFeatures() function is called repetitively for each of the 25 feature maps and a feature vector of dimensions (*m x n*) x 25 is created.

Normalization and Dimensionality reduction using PCA are performed.

For Texture Segmentation, the feature matrix has m x n rows and 3 columns. The number of clusters might vary from 4 in Comb_1 image to 6 in Comb_2 image.

Thus the K-Means function in OpenCV returns a label matrix which has *m x n* rows and 1 column. i.e it predicts a label for each pixel.

Thus we reshape the *m x n* label vector to shape *m x n* using OpenCV's reshape() function. Each of the values in the reshaped matrix is then multiplied by a factor depending on the number of textures in the input image. For Comb_1, the multiplying factor is 85 and for Comb_2 it is 51.

Thus a grayscale image is produced with each constant grayscale value representing a unique texture.

1.3. Experimental Results

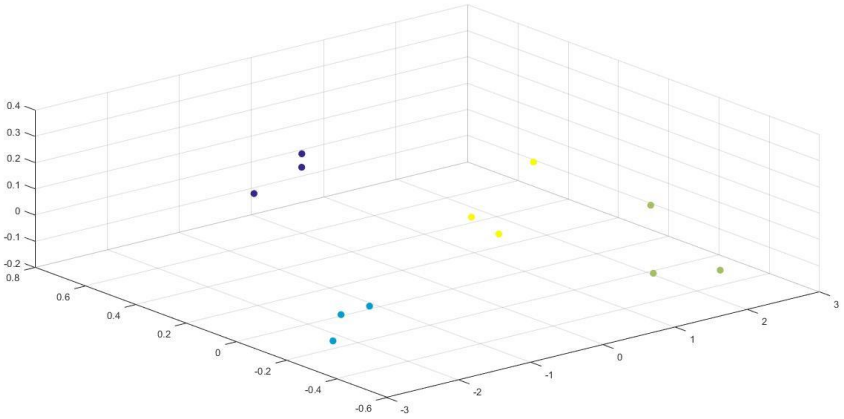


Figure 1a: 3D Scatter plot of Feature Matrix (Corner View)

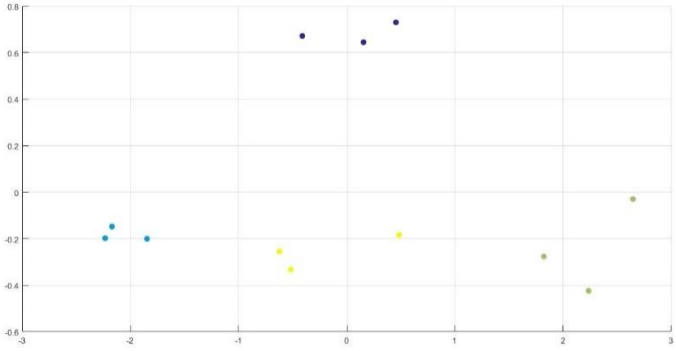


Figure 1b: 3D Scatter plot of Feature Matrix (Top View)

```
hariprasad@hariprasad-Inspiron-5558:~/Downloads/Homework 3/P1$ ./p1a_TextureClassification
 The feature vectors are as follows
[-1.8460166, -0.2002444, -0.070418209;
 -0.62320614, -0.25478178, 0.2061208;
 0.15561762, 0.64466137, 0.025246508;
 -2.2331014, -0.19787721, -0.077726178;
 -0.41081107, 0.67173296, -0.04668894;
 -2.1701958, -0.14763658, -0.19993116;
 1.8220049, -0.2764675, -0.16457973;
 2.2359903, -0.42397779, -0.12836783;
 0.48469162, -0.18333702, 0.3161886;
 2.645684, -0.029640377, -0.047936134;
 0.45523056, 0.73012304, 0.026139624;
 -0.51588732, -0.3325547, 0.16195257]

Printing the labels
[1;
 3;
 0;
 1;
 0;
 1;
 2;
 2;
 3;
 2;
 0;
 3]
```

Figure 2a: Output Labels of K- means using reduced feature matrix

```
Printing the labels
[3;
 1;
 0;
 3;
 0;
 3;
 2;
 2;
 0;
 2;
 0;
 1]
hariprasad@hariprasad-Inspiron-5558:~/Downloads/Homework 3/P1$
```

Figure 2b: Output Labels of K- means using entire feature matrix

Figure 3: 25 Grayscale filtered outputs for Comb1 image

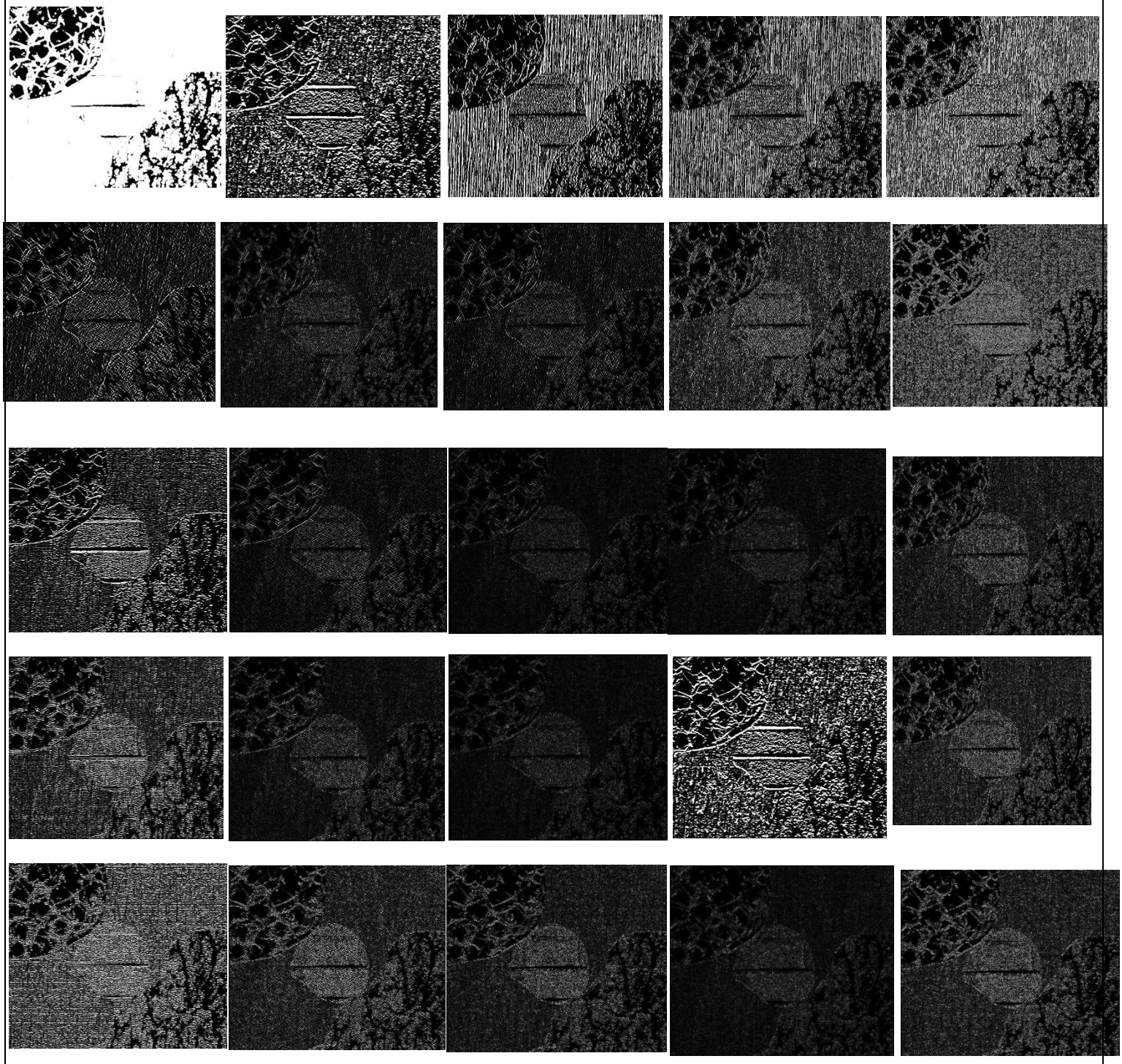


Figure 4: Segmented output of Comb1 using a window size of 49

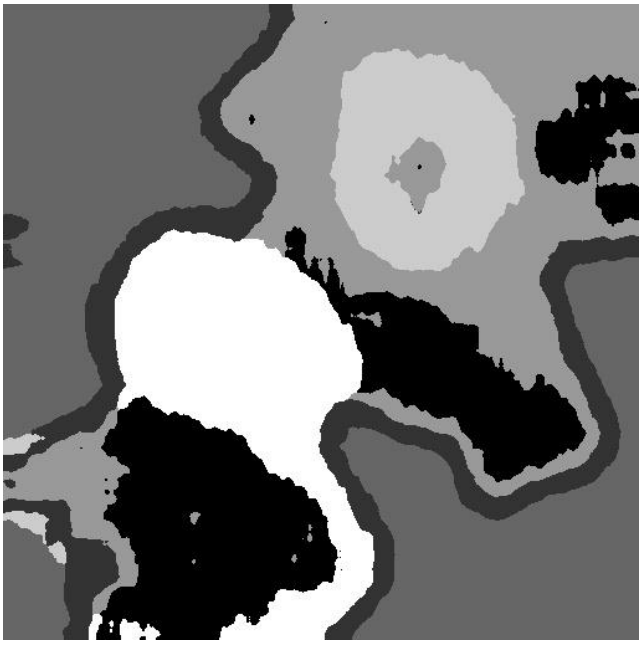


Figure 5: Segmented output of Comb2 using a window size of 49

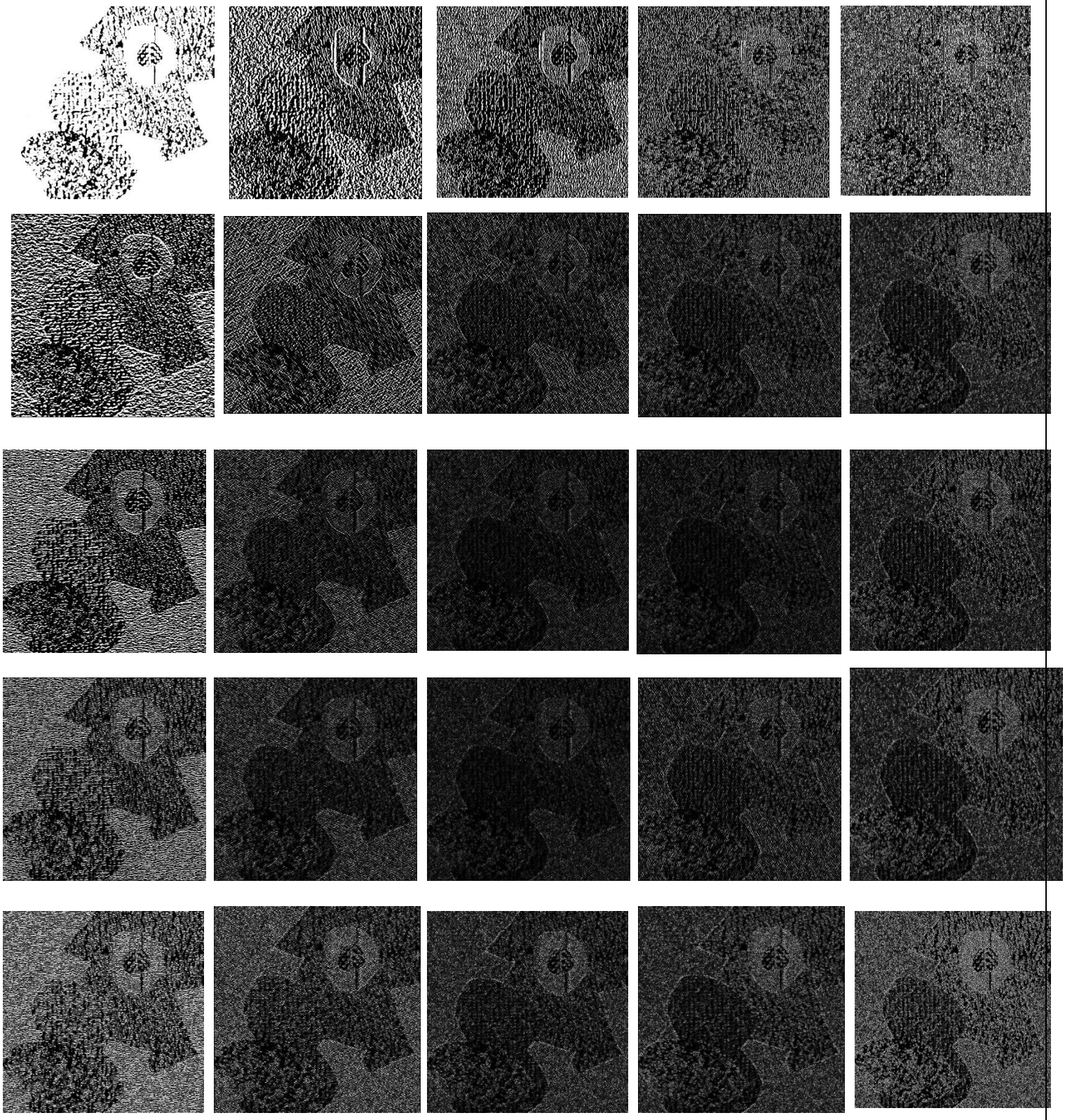


Figure 6: 25 Grayscale filtermaps of Comb2 image

1.4. Discussion
Texture Classification

Figure 1a and Figure 1b show the 4 Clusters color coded that were chosen by the K-means Algorithms. Figure 1a plots the reduced 3-D feature vector in 3 dimensional feature space whereas Figure 1b gives the top view projection. It can be seen the 4 chosen clusters are accurate and give an accuracy of 100%, correctly predicting a unique label for each unique texture. This clearly illustrates dimensionality reduction helps visualizing the feature space of higher dimensions. Further it can also be noticed that projecting the 25D feature vector into 3 dimensional space helps cluster the sample images better, thus saving computational time, and memory.

Discriminant power: In order to check which feature has the strongest discriminant power, the variance of the 25D feature matrix was computed. Dimensions that have low variance will have weak discriminant power, whereas features that have high discriminant power will improve classification. The Variance was computed using MATLAB's *var()* function, and it was noticed that dimension 10 corresponding to $E5^{T}R5$ has lowest discriminant power, and dimension 11 corresponding to feature $S5^{T}L5$ has the strongest discriminant power.

Figure 2a shows the Labels generated by the K-means function. The Label vector is a linear array with integers from 0 to 3, to predict the label for every input sample. The following is the interpretation of the label.

Table 1: Predicted Labels and the Ground truth using PCA

Sample	Ground Truth	Predicted Label
1	Rock	1
2	Grass	3
3	Weave	0
4	Rock	1
5	Weave	0
6	Rock	1
7	Sand	2
8	Sand	2
9	Grass	3
10	Sand	2
11	Weave	0
12	Grass	3

Thus we can see infer the following mapping of textures

Rock -> 1

Grass -> 3

Weave -> 0

Sand -> 2

Figure 2b shows the output of the K-means clustering when all 25D are used for clustering. i.e without dimensionality reduction. It can be seen that without PCA, the accuracy drops to 11/12 % or 91.66%. Highlighted in red in the table below denotes the wrongly labelled sample. Thus PCA helps to increase performance and prevent overfitting.

Table 2: Predicted Labels and the Ground truth without using PCA. Highlighted in red is misclassified label

Sample	Ground Truth	Predicted Label
1	Rock	3
2	Grass	1
3	Weave	0
4	Rock	3
5	Weave	0
6	Rock	3
7	Sand	2
8	Sand	2
9	Grass	0
10	Sand	2
11	Weave	0
12	Grass	1

Dimensionality Reduction using PCA

Principal Component Analysis is a dimensionality reduction procedure that projects a high dimensional feature vector onto to a lower dimensional subspace by selecting the most significant eigenvalue eigenvectors. This method is often performed when the high dimensional data has a lot of redundant features, or for the purpose of visualizing the data in 3 or 2 dimensions. The algorithm to perform PCA is described as follows [8]

- Let X be the feature matrix with D dimensions and m samples. Thus, X is an m x D matrix
- We compute the column-wise mean for every feature and subtract from every column element, its column-wise mean to obtain the $X^{'}$ feature matrix.
- The Singular Value Decomposition of X' is computed. Thus we can write X' as
  - $X' = U*S*V^{T}$
  - U gives the Eigen vector matrix and is of dimension n x n. S is a diagonal matrix whose diagonal elements are the singular values corresponding to Eigen vectors.
  - To reduce the dimension from D to d, we pick the eigen vectors corresponding to the to d largest eigen values in S matrix.
  - If the Eigen values are arranged in descending order, this corresponds to the first d columns of U, S and V.
- To compute the projected feature matrix, we find X^{p} as follows.
  $X^{p} = U^{'T}X^{'}$, where $U^{'}$ is a matrix that contains the first d columns of U.

K – Means Algorithm

The K-means Clustering is an unsupervised algorithm that can predict clusters in a feature matrix to classify data points into different clusters. The algorithm is as follows, [8]

- We begin by Randomly initializing K cluster centroids m_1, m_2… m_k
- Repeat {
  For i = 1 to m samples
  label$^{(i)}$ = index (integer from 0 to k-1) of cluster centroid of Euclidean distance to x(i), which denotes the i th data sample in feature matrix

  For k = 1 to K

  m_k = mean of points assigned to cluster k

}

Thus, it can be noticed that texture such as Rock which has a lot of high frequency components, give a strong response in the $S5^TS5$ filter, whereas the texture sand which has low frequency component give a very strong response for Wave filters.

b) Texture Segmentation

For Texture Segmentation, the objective is to segment out unique textures within a single image and represent them in a grayscale image, with each texture having a unique grayscale value. Comb_1.raw has 4 unique textures, whereas Comb_2.raw has 6 unique textures that we wish to segment out.

Figure 4 and 5 show the segmented output of Comb_1.raw and Comb_2.raw using a window size of 49. It can be noticed that the algorithm fails to accurately segment each unique texture. In Fig 4, the textures in the top left and bottom right are classified as the same texture. Similarly, in Fig 5, the boundary between two unique textures are segmented incorrectly. This is because at the boundary, the average energy within the Window remains different from the Window on either side of the window, and thus the algorithm finds difficulty in labelling the boundary pixels.

The procedure that was followed was similar to the Texture Classification problem where

- 25 laws filters were applied onto the mean subtracted input image, which gives 25 filtered outputs.
- For each pixel in each filtered image, it's energy within a window of length W is computed.
- Thus we obtain a feature vector of dimension (m x n) x 25.
- We normalize the Feature vector by computing the column-wise min and max.
- We apply PCA to reduce the dimensionality from 25 to 3.
- We use K-means Clustering to label each pixel as belonging to 1 of 4 or 6 clusters.
- We generate a Grayscale Segmented output image using the labelled pixels.

Figure 3 and Fig.6 shows the 25 Gray-scale Filtered images obtained using the 25, 5x5 Law's filters for both Comb1 and Comb2 images. Different responses are obtained for different types of filters. It can be seen that the first filtered output looks brighter than the other 24 filtered outputs. This is because, the first response corresponds to the $L5^TL5$ filter and this filter does not have a mean of zero or an average of 1 among the filter values. Thus, due to larger values in the filter kernel, a brighter filtered image is produced.

Problem 2: Salient Point Descriptors and Image Matching

2.1. Abstract and Motivation

Object detection is a very important problem in Computer Vision and has a wide array of applications from security to automation. For example, identifying objects on a road is critical for an automated self-driving applications where a type of vehicle, or sign needs to be identified. In order to identify objects, it is important to identify Scale and Rotational invariant features from objects. SIFT (Scale Invariant

Feature Transform) and SURF (Speeded Up Robust Features) are two approaches to obtain such scale and rotational invariant descriptors from object images. Both methods identify Keypoints in an image and output a 128 Dimensional feature vector for each KeyPoint known as a descriptor. These descriptors are unique to each Keypoint and they give us a metric to identify and match objects for object detection.

In this problem, the SIFT and SURF techniques have been implemented using OpenCV C++. Using the KeyPoints and Descriptors obtained from SIFT and SURF, each pair of images from Jeep, Bus, Rav4_1, Rav4_2 have been matched by calculating the least Euclidean Distance between the descriptors on each of the two selected images.

In part c) of the problem, a Bag of Words approach is implemented to return an image from training that represents a closest match to a test image. The Bag of Words technique uses K-means clustering to obtain K = 8 clusters that represents the most similar descriptors from the training images. Thus each training image will have a code book of 8 bins, containing the number of descriptors belonging to each cluster. When a test image is input, the image with most similar codebook based on least Euclidean distance is returned.

The SIFT and SURF algorithms have been explained in detailed in section 2.4.

2.2. Approach and Procedure

In part a) we extract Salient Points and Descriptors using the SIFT and SURF algorithm independently using OpenCV C++ functions

Using SIFT

We first use, OpenCV's imread() function to read the two JPEG images, passed as arguments to the program.

A SIFT object is created using OpenCV's xfeatures2d namespace. The following command creates the SIFT object in OpenCV

cv::Ptr<Feature2D> sift_object = xfeatures2d::SIFT::create();

This creates a pointer of type Feature2D called *sift_object*. SIFT operations such as computing the KeyPoints, and descriptors can be called using the *sift_object* point using the '->' syntax.

i.e. to Compute Keypoints, we call the *detect()* function as follows. This function returns the set of coordinate locations in the passed Image that are identified as Scale and Rotational Invariant Keypoints.

sift_object->detect(Image, SIFT_keypoints);

To compute the descriptors or the 128 D feature vector for each Keypoint, we call the *compute()* function.

For example, to compute the descriptors for the above found Keypoints we use the following command.

sift_object->compute(Image,SIFT_keypoints,SIFT_descriptors);

This function returns a 2D Mat file named SIFT_descriptors that is of dimensions n x 128, where n refers to the total number of Keypoints.

To overlay the Keypoint locations on top of the original image, the *cv::drawKeypoints()* function can be called.

Using SURF

The methodology for using and creating a SURF object in OpenCV is similar to the method followed in SIFT. The only difference is that the SURF object accepts a parameter called minHessian that states the minimum size of the Hessian that is computing using the SURF algorithm.

Ptr<Feature2D> detector = xfeatures2d::SURF::create(minHessian);

The procedures for detecting keypoints and computing descriptors remains unchanged from the method followed above for SIFT.

b) Object Matching

For object matching, we define a vector of type DMatch called matches.

vector< DMatch > matches;

We then call the Flann Matcher function called *FlannBasedMatcher [2]*. This function defines a matcher object that takes as arguments, the distance measure for matching. The FLANN [2] based matcher works similar to BFMatcher and computes the approximate Nearest Neighbours using a Fast Algorithm.

FlannBasedMatcher matcher;

The best matches are found by referencing the *match()* function through the *matcher* object, which gives the best matches of Keypoints from Image 1 to Image 2 as a *DMatch vector*.

The function *cv::drawMatches()* produces an images that draws the matches from Image 1 to Image 2.

c) Bag of Words

In the Bag of Words technique, we use K-Means clustering to cluster the SIFT descriptors of Jeep, Bus, and Rav4_1 images into 8 clusters or bins. We then count the number of descriptors in each of the three images that lie in each of the 8 clusters to maintain a codebook. Thus each of the three images contain a unique histogram or codebook, identified by their Keypoint descriptors.

Given a new test image, Rav4_2 we proceed by computing the SIFT descriptors for it, and generating its 8 bin codebook by counting the number of descriptors that lie in each cluster based on least distance from 8 centroids. Thus having the histogram of the test image we proceed by finding the similarity of the histogram from the histograms of each of the three images based on Euclidean Distance.

Algorithm is further explained as follows

- Read all four images using *cv::imread()*
- Define a SIFT object
- Compute Keypoints and Descriptors for all four images using the procedure explained above
- Vertically concatenate the descriptors of *Jeep, Bus* and *Rav4_1* images into one single Mat variable. Here, OpenCV's *vconcat()* function have been used.
- Use K-Means clustering to cluster the concatenated matrix into 8 clusters.

  kmeans(features, 8, labels,

  TermCriteria(CV_TERMCRIT_ITER|CV_TERMCRIT_EPS, 30000, 0.00001),

10, KMEANS_RANDOM_CENTERS, centers);

Here, the argument *features* consists of n_tot x 128 dimensions, where n_tot is the total sum of the number of Keypoints in the three images. The algorithm terminates when 30000 iterations have been completed or if the change in centroid locations is less than 0.00001.

Further, the Kmeans function returns a labels matrix that has a label from 0 to 7 for each descriptor in features.

The function also returns the 8 centroids in *centers.*

- Maintain a histogram vector with 8 bins for each of the three images and count the number of descriptors that lie in each of the bins. The *labels* matrix gives information about the associated cluster of each descriptor.
- Generate a histogram vector with 8 bins for the test image Rav4_2.raw and count the number of descriptors that fall into each bin. This is calculated by measuring the Euclidean distance of each descriptor from each of the 8 centroids and assigning the label as the cluster that is closest.
    - The *pow()* function is used to compute the squared distance between the centroids.
    - To find the cluster with least distance the *std::min_element(vec.begin(),vec.end())* – *vec.begin()* function has been called.

- Find the Euclidean distance between the test histogram and each of the 3 histograms from training images. The best match is the one with least distance.

2.3. Experimental Results

Figure 7: SIFT Keypoints on the Left and SURF keypoints on the Right for bus image

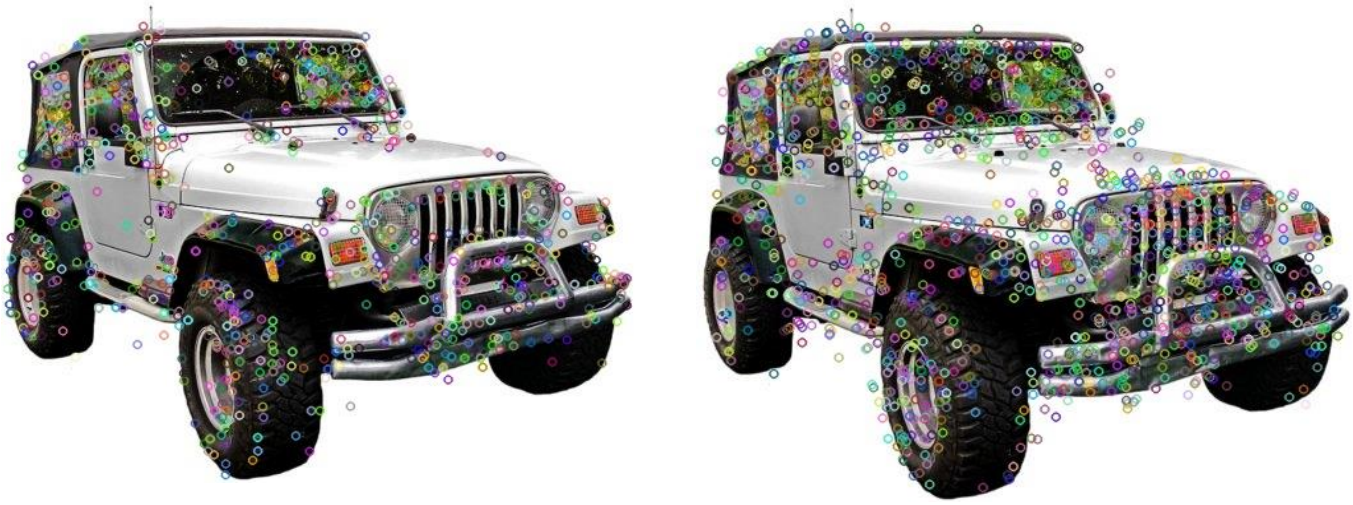


Figure 8: SIFT Keypoints on the Left and SURF keypoints on the Right for jeep image

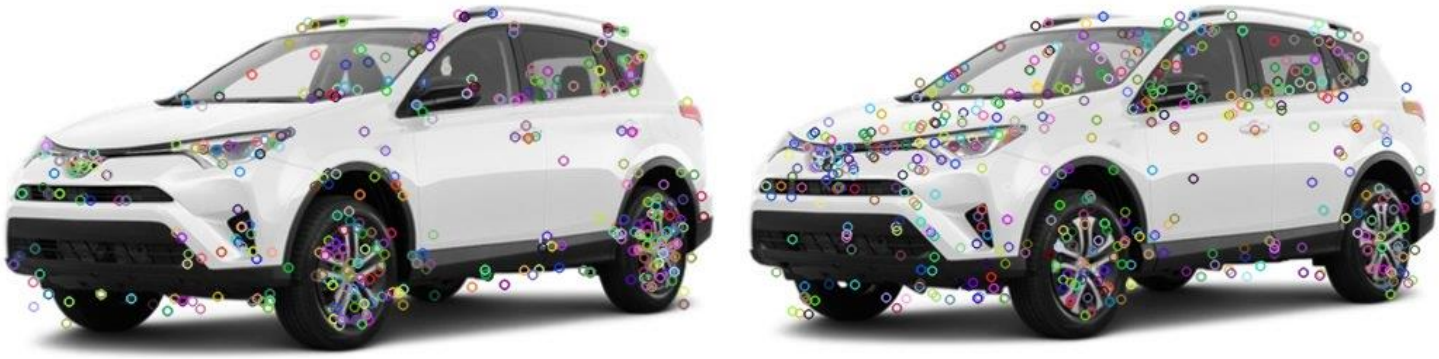


Figure 9: SIFT Keypoints on the Left and SURF keypoints on the Right for rav4_1 image

Figure 10: SIFT Keypoints on the Left and SURF keypoints on the Right for rav4_2 image

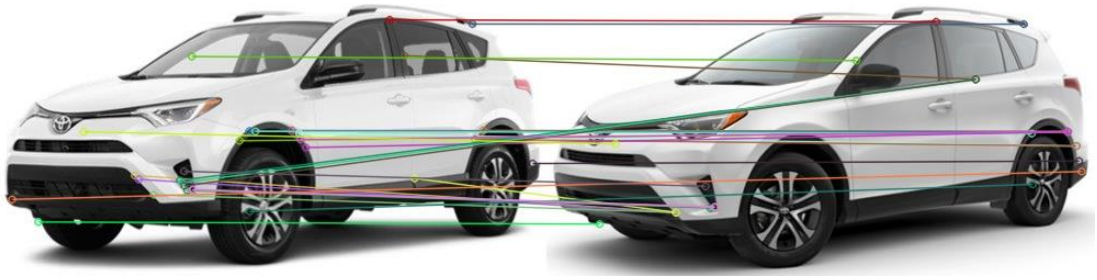


Figure 11: rav4_1 and rav4_2 matched images using SIFT descriptors

Figure 12: rav4_1 and rav4_2 matched images using SURF descriptors

Figure 13: rav4_1 and jeep matched images using SIFT descriptors

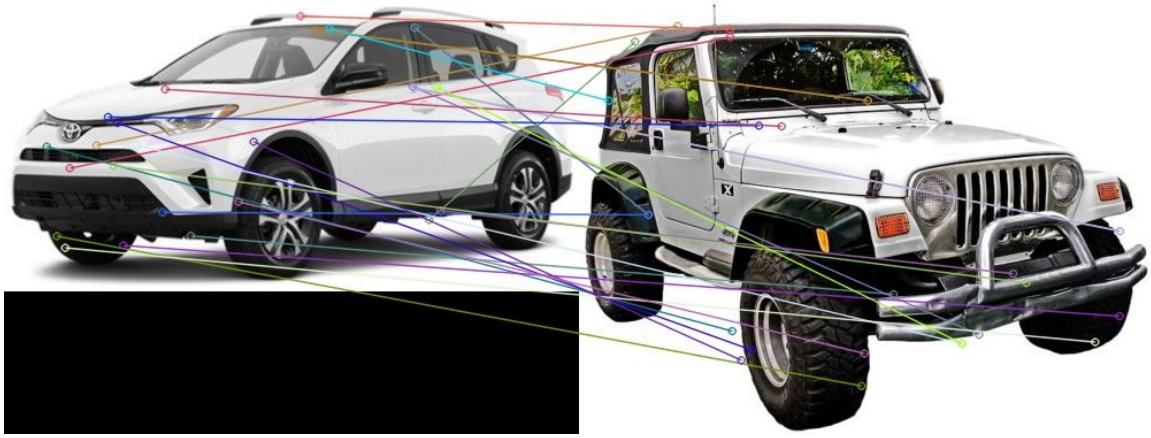


Figure 14: rav4_1 and jeep matched images using SURF descriptors

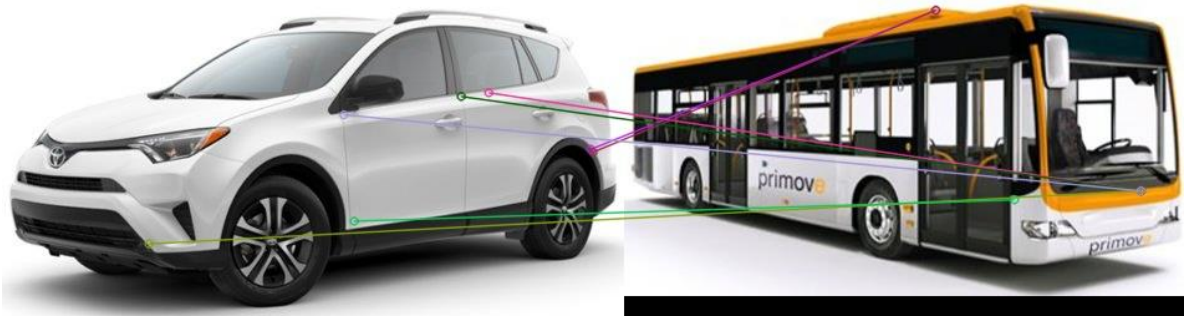


Figure 15: rav4_2 and bus matched images using SIFT descriptors

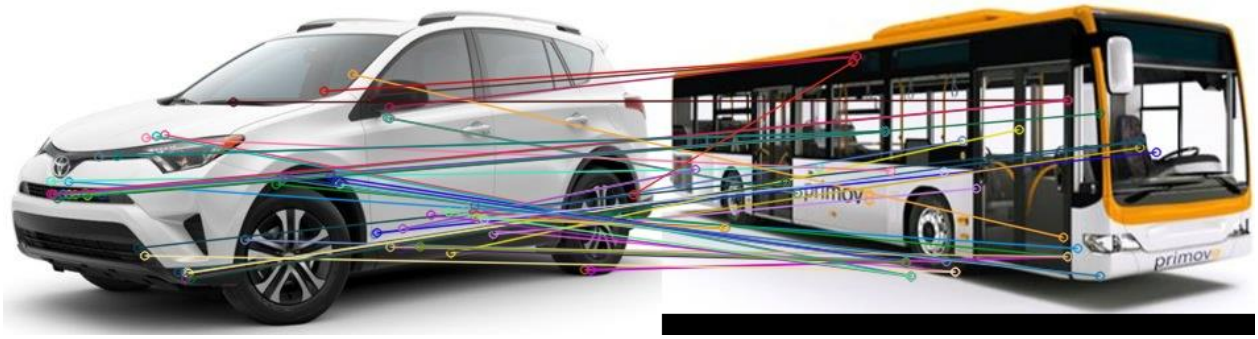


Figure 16: rav4_2 and bus matched images using SURF descriptors

Figure 17: Output of Bag of Words based object matching

2.4. Discussion

SIFT Algorithm

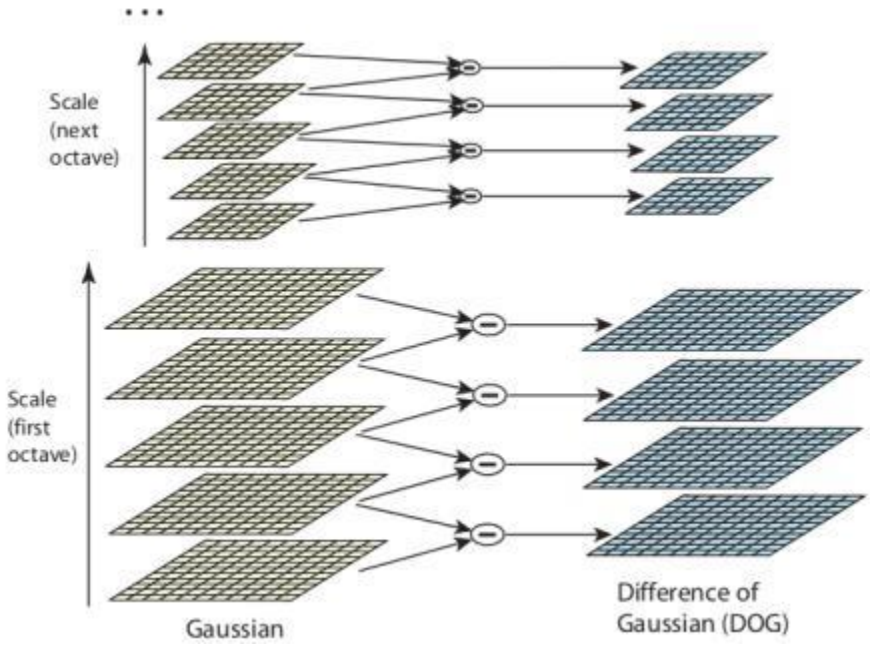
The SIFT algorithm detects Scale and Rotational invariant features that uniquely describe the an image, and thus can be detected in different projections and scales. The algorithm is as follows

Scale – Space Extrema Detection

Here, Scale invariant Keypoints are obtained. To achieve this, a scale space filter is first applied such as the Laplacian of Gaussian (LoG). Since LoG requires high computational cost, the Difference of Gaussian (DoG) is approximated as LoG. Thus DoG is obtained at various σ, where low σ allows the detection of small corner and a high σ helps to detect larger corners, acting as a scaling parameter. Thus

DoG is obtained by computing the difference of different Gaussian filtered images at varying σ, such as k σ. [4]

Figure 18: Principle of DoG at multiple Octaves: Image Cource: http://docs.opencv.org/3.1.0/sift_dog.jpg [4]



In the above image, 4 DoG's are computed at various Octaves. Each octave represents a unique resolution of the image. The image is progressively Subsampled to a lower resolution as we move to the next Octave.

At every Octave, the different DoG layers are searched for local extremes. A pixel is labelled as an extremum if it's values is maximum within a 3x3x4 volume of pixels on all 4 layers of the DoG. This gives us a potential Keypoint.

Keypoint Localization

It was noted that illumination of the image affects the KeyPoint detection problem and thus anomalies in illumination had to be removed to identify false Keypoints. For this purpose, a threshold value is selected based on the Taylor Series expansion of the Scale Space. Intensity values that exceed the threshold are rejected.

It was further noted that DoG's are sensitive to edges and thus affect the Keypoint detection problem. For this part, Edges are don't contribute to Keypoints are removed based on a Hessian Matrix. The Hessian is of dimension 2 x 2 and the Ratio of the eigen values is used to threshold the detection of Keypoints. [4]

$$H = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix}$$

Let r be the ratio of the Trace and Determent of H.

If ratio is $> \frac{(r+1)^2}{r}$, reject the feature point, where r = 10.

Orientation Assignment

This part of the algorithms tries to compute Rotational Invariant features. Here, a rotation is assigned to each Keypoint.

To compute the direction, the gradient magnitude and direction of a Keypoint is computed within a neighborhood region. The gradient vectors are binned into 36 possible bins covering 360 degrees. The vector with the highest gradient magnitude or magnitude above 80% is chosen. [4]

Generating the SIFT Descriptor

For every Keypoint, it's 16 x 16 neighborhood window is chosen and is divided into 16 sub-blocks of size 4 x 4. [4]

For each 4 x 4 block, an 8 bin orientation histogram is computed giving us 16 samples. In all 16 x 8 vector descriptor for each Keypoint is computed.

SURF Algorithm

The SURF algorithm is a modification of the SIFT algorithms that attempts to address the drawback of SIFT by reducing the computational costs

The SURT technique computes the LoG using a box filter as convolution with integral images is quicker using the box filter.

For orientation assignment, a neighborhood of 6x pixels [5] around the Keypoints are chosen and the Gaussian weighted wavelet response in the horizontal and vertical directions are computed.

These are then plotted across dx and dy directions, and the dominant direction is one in which the sum of responses along a moving orientation window at 60 degrees is maximum. [5]

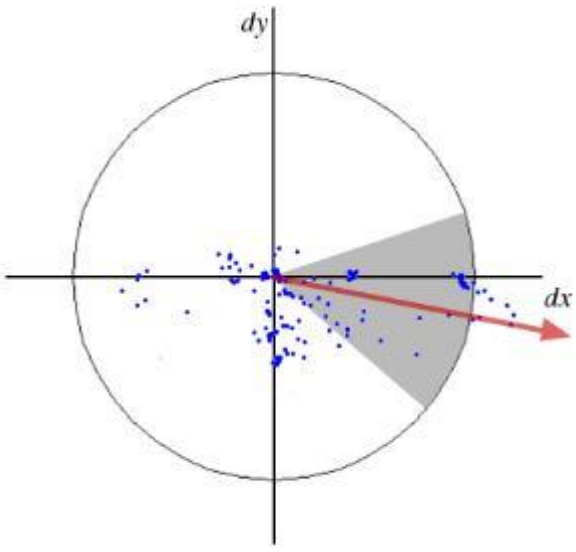


Figure 19: Orientation Assignment using a Sweeping section. Image Courtesy: http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_surf_intro/py_surf_intro.html

The feature descriptors are also computed using the wavelet response in x and y directions inside a 20x windows surrounding the Keypoint. The SURF algorithms returns a 128 dimensional feature vector for every Keypoint.

Given the response in d_x and d_y directions the feature descriptors are represented as follows [5]

Descriptor v = $(\sum d_x, \sum d_y, \sum |d_{x|}, \sum |d_y|, \sum d_x \ for\ d_y < 0, \sum d_x \ for\ d_y \geq 0, \sum |d_x| \ for\ d_y < 0, \sum |d_x| \ for\ d_y \geq 0, \sum d_y \ for\ d_x < 0, \sum d_y \ for\ d_x \geq 0, \sum |d_y| \ for\ d_x < 0, \sum |d_y| \ for\ d_x \geq 0)$

Figure 7,8,9,10 shows the Extracted SIFT and SURF Keypoints for Jeep and Bus, Rav4_1 and rav4_2 images. It can be seen that Keypoints are observed near the wheels, front grill, and roof might serve as key descriptors for object detection. Further, we can see that SURF has a lot more KeyPoints that SIFT methods.

SIFT vs SURF

SIFT and SURF both extract similar Keypoints for both Jeep and Bus images. The SURF algorithm is computationally faster than SIFT and detects more KeyPoints than SIFT. SURF hover ever fails as compared to SIFT in illumination change and Viewpoint change. [3]. In Image matching, it is noticed that SIFT features produce more reliable matches than SURF features, as SIFT features match the tire, bumper etc well.

b) Image Matching

Figure 11 , 12 ,13 ,14 ,15 16 shows the Image Matches between Rav4_1 and Jeep, and Rav4_1 and Bus. The FLANN based matcher function computes match for all Keypoints and thresholds those whose distances are greater than 2 time the minimum distance. Thus, only the most similar KeyPoints, based on the descriptors are matched. It is noticed that images that look very similar such as rav4_1 and rav4_2 are matched well. This is seen as KeyPoints in the roof, wheel, grill, bumper are respectively matched with the second image.

However, we can see one of the drawbacks of matching two different objects. Some Keypoints near the wheel of Rav4_1 image are matched to non-wheel points in Jeep and Bus images. Further certain Keypoints near the rear view mirror are incorrectly match. In summary, image matching of two dissimilar images does not give good results. This can be because of the orientation or project of Image in Rav4_1 image. It is noted that the vehicle in Rav4_1 is oriented opposite to the direction of the Jeep and Bus image.

C) Bag of Words

Figure 17 shows the output of Bag of words matching. The bag of words algorithm correctly returns Rav4_1 image as the closest image to Rav4_2. Thus the code word generated for Rav4_1 and Rav4_2 have good similarity and proves that the Bag of Words technique is a viable option for matching images. This is because the Bag of Words techniques groups similar descriptors into clusters and thus all key points that are near the wheels are clustered into one bin and similarly for the headlights. Thus a more intuitive match is obtained.

Problem 3: Edge Detection

3.1. Abstract and Motivation

An edge is a region in an image where there is a sudden difference adjacent pixel values. Edge detection has been an age old problem in Image Processing and is a crucial part of Image Segmentation for Computer Vision algorithms. For example, an application to count the number of coins in an image will first require to find the edges of the coin that separate the background from the coins.

Over the years several approaches have been followed for edge detection. The main approaches can be segregated into three main categories

1. Template Matching eg. Prewitt filter, Nevatia-Babu filter
2. 1st Order gradient information. Eg. Sobel filter
3. 2nd Order gradient information eg. Laplacian filter

Other major techniques for Edge detection lately have been Sketch Tokens and Structured Edge.

In parts a) and b) of this problem, the Canny Edge detector and the Structured Edge detection techniques have been applied to Zebra and Jaguar Images, and their performance, drawbacks and advantages have been explained in details along with the theoretical explanation of the two techniques.

For Performance evaluation, the F1 score has been computed which is a function of Recall and Precision with ground truth images.

MATLAB has been used in this problem for all three parts, and the Structured Edge toolbox developed by Piotr Dollar has been used. The toolbox can be downloaded from https://github.com/pdollar/edges by following the installation steps.

It was noticed that the Structured Edge technique performs much better than Canny edge detector and the detected edges are closer to the ground truth edges. This outcome is based on the performance in the mean F1 score.

3.2. Approach and Procedure

Canny Edge Detector

The Canny Edge detector is an approach that has 5 main procedures. These procedures have been explained and discussed in detail in Section 3.4. The main procedures are

1. Gaussian Filter. To remove noise
2. Determining the intensity gradient of the image
3. Non- Maximum Suppression. To thin the edges
4. Double threshold. To remove false edges
5. Edge tracking by hysteresis

The MATLAB Image Processing toolbox comes with an inbuilt function to implement edge detection. This function has been used to detect edges using the Canny approach in this problem.

The *p3a_canny.m* implements a function that accepts a filename to a raw image, and its dimensions in width, height and number of channels as arguments.

It can be called as follows

Canny_edge_Image = p3a_canny(raw_image_filename,width,height,number_of_channels)

Inside the function, a function named *readraw()* is called that has been modified to accept a raw image and return a 3D variable that represents the colour image.

Since the Canny edge detector is only applied to grayscale image, the *rgb2gray()* is called that converts the 3 channel image to a single channel gray scale image.

This grayscale image is then passed to the *edge()* function as follows

```
Canny_edge = edge(rgb2gray(Image),'Canny',threshold,sigma);
```

The argument threshold is a vector with 2 elements denoting the low and high threshold such that low threshold < high threshold < 1.

Sigma refers to the degree of blurring and is a parameter for the Gaussian filter that is applied prior to edge detection.

Different values of low and high threshold, and sigma were applied and the results have been displayed and discussed in Section 3.3. and 3.4. respectively.

Structured Edge Detection

The Structured Edge detection implements a Machine Learning approach using Random Forest Classifier to learn outcomes from the BSDS500 ground truths. This gives the technique better robustness and a closer edge map to the ground truth. This approach is further explained in detail in section 3.4.

For the purpose of implementation, the Structured Edge MATLAB toolbox provided by the Author Piotr Dollar has been used. The toolbox can be downloaded from this GitHub repository. [1]

The MATLAB package includes a *edgesDemo.m* file to train a Random Forest Model [1] and detect the edges for a given image. This file has been modified to accept as arguments the address to the raw image filename and dimensions of the raw image. Further the following parameter have been fine tuned to give a better edge map of the image.

```
%% set detection parameters (can set after training)
model.opts.multiscale=0;          % for top accuracy set multiscale=1
model.opts.sharpen=2;             % for top speed set sharpen=0
model.opts.nTreesEval=4;          % for top speed set nTreesEval=1
model.opts.nThreads=4;            % max number threads for evaluation
model.opts.nms=1;                 % set to true to enable nms
```

The *nTreesEval* parameter determines the number of trees to evaluate in parallel. The fine edge map for a patch that is chosen is the average of all the outcomes of all the trees and choosing the outcome that is maximum. Increasing the number of trees trains more decision trees and thus improve the accuracy.

The number of threads represents the number of parallel processes to run to evaluate the trees. Making the number of thread equal.

Setting the sharpen parameter to 0 blues the edges and thus we increase the sharpen parameter to get crisp edge.

After Model training a Probabilty Edge Map is generated by calling the function

```
E=edgesDetect(I,model);
```

We obtain a Binary Edge map from the Probabilty Edge by Thresholding. The Thresholding Parameter T can be fine-tuned to increase the F score. Values that are above the Threshold are set to Binary 1 and values below the Threshold are set to 0

Thus, a Binary Edge map is obtained that gives the Edges detected by the Structured Edge approach.

Performance Evaluation

Prior to performing evaluation, the ground truths are loaded into a **struct** as follows

1. A struct G is created with a member named ‘groundTruth’.
2. We defile G.groundTruth as a 1 x 5 cell to accommodate 5 ground truth images.
   For each i in 1 to 5

a. We create a struct with member named 'Boundaries' and assign it to G.groundTruth(i)
b. Each element in G.groundTruth(1,i).Boundaries is assigned to a groundtruth image that is read using the *readraw()* function and is converted to Binary where edges are represented by 1 and Background as 0.

3. The struct G is then saved as mat file using the *save()* function.

F1 Score

The F1 score is computed by calling the function, where the detected binary edge map E and ground truth structure filename are passed as arguments

```
[thrs,cntR,sumR,cntP,sumP,V] = edgesEvalImg( E, 'G.mat');
```

Recall and Precision are computed as follows

Recall = cntR./ sumR;
Precision = cntP./ sumP;

Having computed Recall and Precision, F score can be computed using the formula

$$F = \frac{2 * Recall * Precision}{Recall + Precision}$$

Higher the F score, the better the performance of the edge detector.

3.3. Experimental Results

Figure 20 i: Canny Edge, Low = .15, High = .3, σ = 1

Figure 20 ii: Canny Edge, Low = .15, High = .3, σ = 2

Figure 20 iii: Canny Edge, Low = 0, High = .26, σ = 2

Figure 20 iv: Canny Edge, Low = .13, High = .26, σ = 2

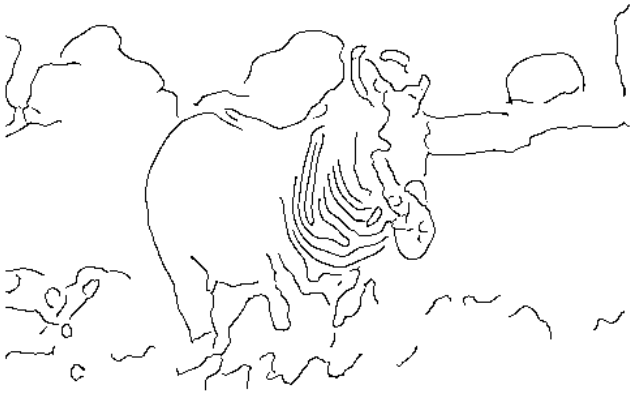


Figure 20 v: Canny Edge, Low = .1, High = .2, σ = 3

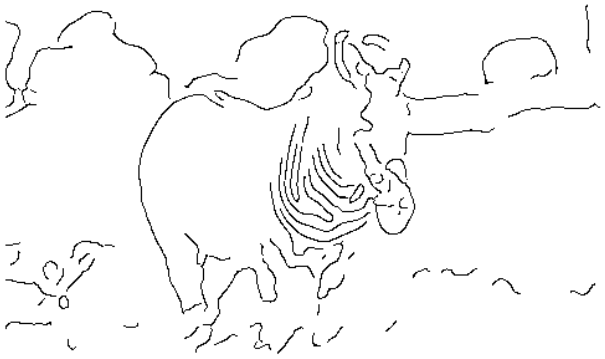


Figure 20 vi: Canny Edge, Low = .15, High = .2, σ = 3

Figure 21 i: Canny Edge, Low = .2, High = .5, σ = 1

Figure 21 ii: Canny Edge, Low = .2, High = .5, σ = 2

Figure 21 iii: Canny Edge, Low = .15, High = .24, σ = 3

Figure 21 iv: Canny Edge, Low = .18, High = .4, σ = 3

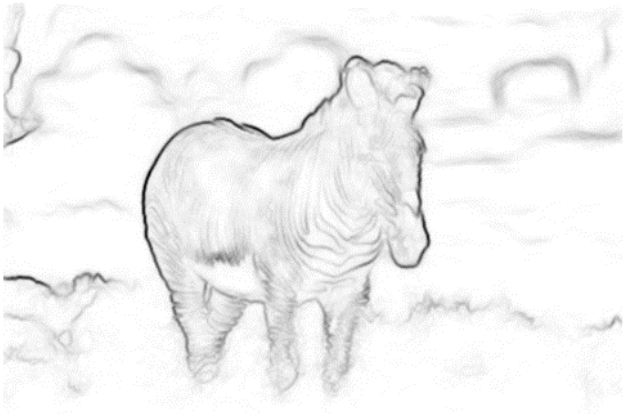


Figure 22 i: Structured Edge Zebra probability map without NMS

Figure 22 ii: Structured Edge Zebra binary map without NMS

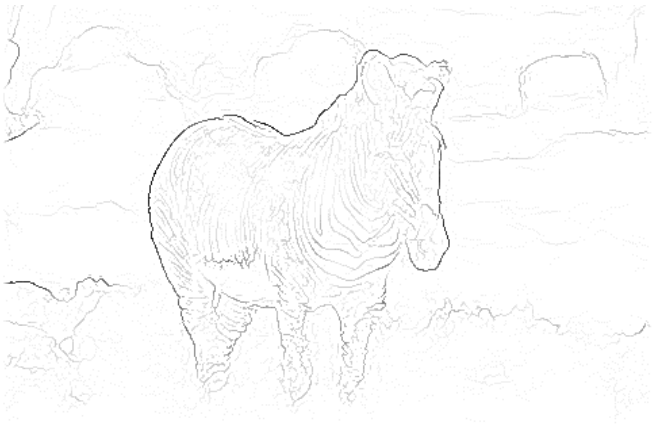


Figure 22 iii: Structured Edge Zebra probability map with NMS

Figure 22 iv: Structured Edge Zebra binary map with NMS

Figure 23 i: Structured Edge Jaguar probability map without NM

Figure 23 ii: Structured Edge Jaguar binary map without NMS

Figure 23 iii: Structured Edge Jaguar probability map with NMS

Figure 23 iii: Structured Edge Jaguar binary map with NMS

Performance Evalutation

The columns represent F score obtained using the SE package

	Ground Truth 1	Ground Truth 2	Ground Truth 3	Ground Truth 4	Ground Truth 5	Ground Truth 6	Mean
Zebra - SE	0.5654	0.4549	0.5038	0.5611	0.5703	-	0.5985
Jaguar- SE	0.6303	0.3615	0.6565	0.6714	0.4742	0.3768	0.7572
Zebra - Canny	0.4775	0.4532	0.3658	0.4024	0.485	-	0.5601
Jaguar - Canny	0.6629	0.3452	0.6259	0.6826	0.4323	0.3656	0.7527

Table 3: F score of Zebra and Jaguar using Structured Edge and Canny Edge Detector

Zebra SE using binary Threshold = 0.166	F Score = 0.5926
Zebra SE using binary threshold= 0.186	F Score = 0.5867

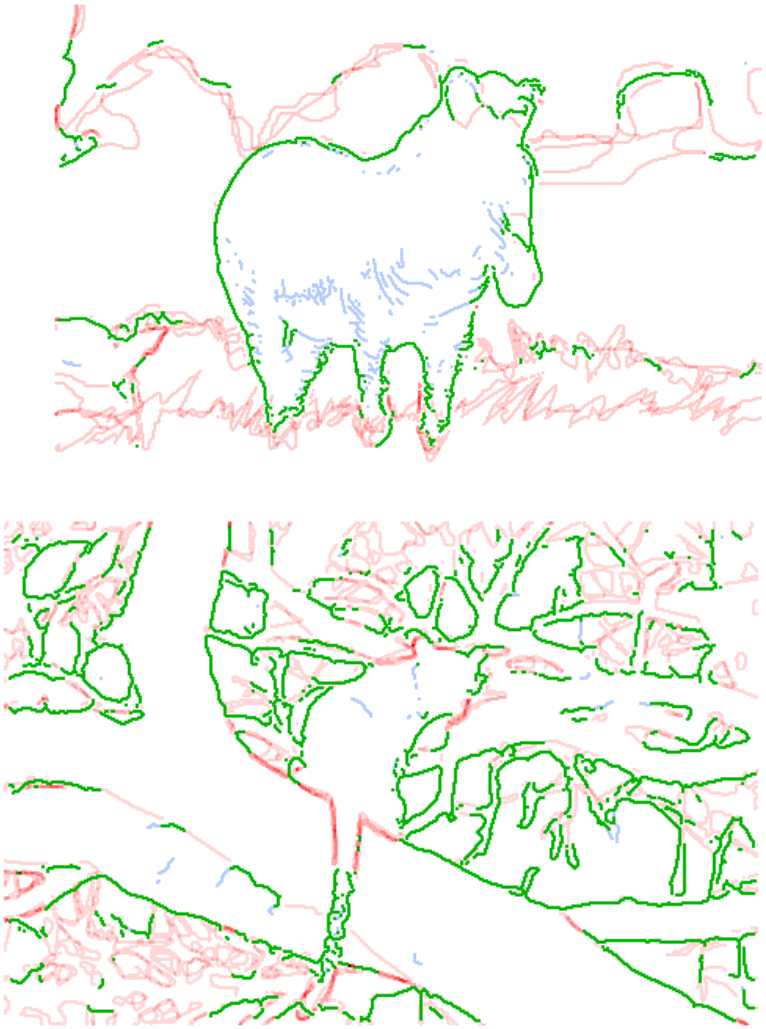


Figure 24: Color coded output representing TP, TN, FP,FN for Zebra and Jaguar Images

3.4. Discussion

The Canny Edge Detection Algorithm

As explained in Section 3.2. the Canny Edge detection approach guarantees better robustness than older methods such as Sobel or LoG by removing false edges. The primary procedure is as follows. [3]

a) Noise Removal using Gaussian smoothening

The presence of noise in an image degrades the performance of edge detection algorithms as most edge detection procedures such as Sobel and LoG are susceptible to noise and give false edges. Thus the first step involves the application of a 5 x 5 Gaussian filter to remove the noise. The more the σ, the greater the blurring effect.

b) Computing the Intensity Gradient

Once the Image has been smoothened using the Gaussian filter, the intensity gradient is computed using the Sobel filters in Horizontal and Vertical Direction. This gives the first derivative in horizontal and vertical directions at every pixel location.

Let G_x be the gradient in horizontal direction and G_y be the gradient in vertical direction, then

$$\text{Gradient Magnitude} = \sqrt{G_x^2 + G_y^2}$$

$$\text{Gradient Angle } (\theta) = \tan^{-1}\frac{G_y}{G_x}$$

Thus higher the magnitude, the greater is the edge gradient.

c) Non-Maximum Suppression

Non-Maximum suppression is used to thin the edges to one-pixel edge, using the gradient magnitude, and gradient information.

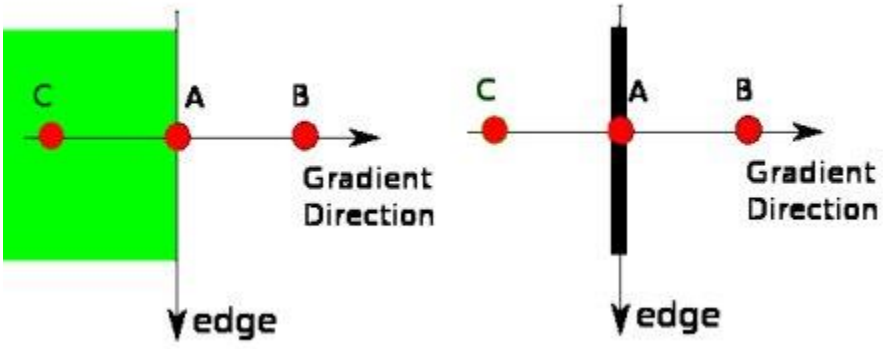


Figure 25: Non – Maximum Suppression. Image courtesy: http://docs.opencv.org/trunk/da/d22/tutorial_py_canny.html [3]

For every pixel A in the image, it's neighboring pixels B and C that are before and after it in the direction of the gradient are compared. If value of A is maximum of B and C, it is preserved, if not, it is suppressed and threshold to 0. Thus only one using edge is obtained after Non-Maximum Suppression.

d) Double Thresholding

Post application of Non-Maximum Suppression, a double threshold is applied, ie, a low threshold and a high threshold. Edge values that are above the High Threshold are treated as definite edges while edge values that are lower than the Low Threshold are false edges and suppressed. The values in-between the High and Low Thresholds are decided based on a process called Hysteresis Thresholding. [3]

e) Hysteresis Thresholding

Hysteresis Tracking is a procedure where edge values that lie in-between the Low and High Threshold are inspected based on their connectivity.

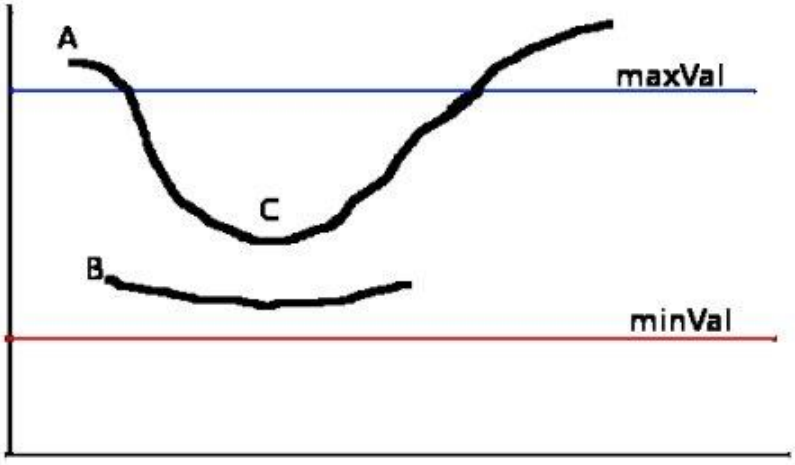


Figure 26: Non – Hysteresis Tracking. Image courtesy: http://docs.opencv.org/trunk/da/d22/tutorial_py_canny.html [3]

In the above image, edge values in region A are strong edges. Edge values in C and B lie in-between. In Hysteresis tracking, the edge values in region C are preserved as it is connected to strong edges A, whereas Edge values in region B are neglected as they are not connected to any strong edge.

Analysis of Results

Figure 20 and 21 show the results on applied the Canny edge detector on Zebra and Jaguar images. In figure 20, different edge maps are obtained when the parameters such as Low Threshold, High Threshold and sigma are varied. It can be noticed that for low sigma, i.e sigma = 1, the local texture is captured and leads to a lot of edges. For Zebra image, the local texture includes the stripes of the zebra and the grass. Increasing the sigma increases the blurring of the image, and this helps in removing the underlying texture. The results in images 20) ii, iii and iv shows that when the low and high threshold are chosen as 0.15 and 0.3, some of the false texture images can be removed. Lowering the lower threshold to 0 or 0.13 does not suppress some of the false edges. Similarly lowering the high threshold to 0.26 degrades the quality of the edge detection. In all the images with sigma =2, there is still some amount of texture visible. Thus, Sigma was increased to 3. This showed superior results and the obtained output does well in removing the local texture in the Zebra and Jaguar Images. The images in 20 v, vi resemble more of hand drawn ground truths. Further, the patterns of stripes on the body of the Zebra are removed.

Similarly, in 21 i) it can be notices that choosing a lower value of Sigma leads to the detection of a lot of edges, and thus increasing the value of Sigma leads to suppression of textures. In 21) ii and iii) it is

noticed that ii) gives better results with lesser false edges as compared to iii) Thus increasing the Lower threshold from 0.15 to 0.2 and Higher Threshold from 0.24 to 0.5 helps better decide the stronger edges and suppress the weak edges that are not connected to the strong edges. In figure 21) iii , increasing the Sigma to 3 helps refine the edges by removing some texture. Further a threshold of 0.18 and 0.4 (high) yields good results.

Among Zebra and Jaguar image, the Zebra image performs better than the Jaguar image using the Canny Edge detector as the object in Jaguar image is submerged inside the thick texture of branches in the background. Thus the detection of edges of the object is harder for the Jaguar image than the Zebra image. The Zebra's location can be clearly detected from the Canny Edge map whereas the Jaguar's location cannot be precisely determining from the Canny Edge map.

Tuning the parameters by adjusting the Low , High threshold and Sigma helps to extract the contour of the Zebra image, but however some amount of texture is still captured in the background. The best chosen parameter for the Zebra image using the Canny edge detector is Low threshold = 0.15, High threshold = 0.2, Sigma =3.

For Jaguar image, tuning the parameters does not help to identify the Jaguar object. As explained above, the background texture along with the overlaying shadow on the Jaguar image makes it harder to detect the Jaguar's body contour.

Structured Edge Detection Algorithm

The structured edge uses a machine learning approach for edge detection. As the algorithm learns from ground truth data that is segmented by humans, it succeeds in giving a more robust result with a high.

Structured edge trains a Random Forrest Classifier as follows [1]

Step 1: 32 x 32 image patches are extracted from the image and from human segmented ground truth.

Step 2: Features are extracted from the 32 x 32 image using two main approaches, namely Pixel lookups and Pairwise differences [1]

Pixel Lookups [1]

Here we generate additional channels of information in addition to the 32 x 32 channel. 3 colour channels in the CIE-LUV color space are first obtained. Normalized gradient magnitude is calculated as two different scales of the image. Further, from these two gradient magnitude images, 4 channels are obtained based on gradient orientation. All the above mentioned channels are blurred by a triangle filter of radius 2 and are down sampled by a factor of 2. Thus 13 channels are obtained, 3 color, 2 magnitudes and 8 orientations.

The channels are further down sampled by 2 giving 32 * 32 * 13/4 = 3328 features.

Pairwise Lookups:

Each channel is blurred with a triangle filter of 8-pixel radius and are down sampled to 5 x 5 resolution. Each pixel pair in each channel is sampled and difference is computed giving ${}^{5*5}_{2}C * 13$ possibilities. Thus totaling 7728 features for every patch

Step 3: Decision tree

In this step, we decide threshold conditions and features to choose at each node of the decision tree so that an image patch outputs an optimum edge segment in the output space. To decide the feature at the first node, we choose the feature that gives the highest entropy gain when an input patch is passed through the node. In other words, clusters of image segments that are similar are given binary labels, and the feature that gives the least entropy based on these labels is chosen. This principle is chosen to choose features are different nodes and eventually form a decision tree.

Step 4: Similarity Mapping [1]

The outcome of the Structured Edge classifier is a Structured Label $y \epsilon Y$ which is a 16 x 16 segmentation mask with binary values. Thus the possible permutations is of the order 2^{16*16}. Computing Euclidean distance for similarity might be infeasible. To avoid this, similarity is computed for pairs of pixels at two different locations to check if they have the same value. Thus for a 16 x 16 image there are $\binom{256}{2}$ such combinations which is of lower dimension. Further, pair of pixels carry more information than a single pixel alone, and this aids the algorithm.

Step 5: We thus use this similarity measure to train the decision tree at every node to correctly divide the examples into two classes. This process is repeated at every node, to finally form a decision tree in which the samples of similar edges will be clustered together at the leaf nodes.

Step 6:

Multiple decision trees are trained in parallel by randomly selecting features. Thus for a given test patch, multiple outcomes will be obtained, and these outcome probabilities are averaged to evaluate the argmax. The structured label corresponding to the argmax is chosen.

Step 7: While testing, the test image is raster scanned for 32 x 32 patches and output edge map is computed.

The following flow explains the procedure of the Structured Edge training algorithm.

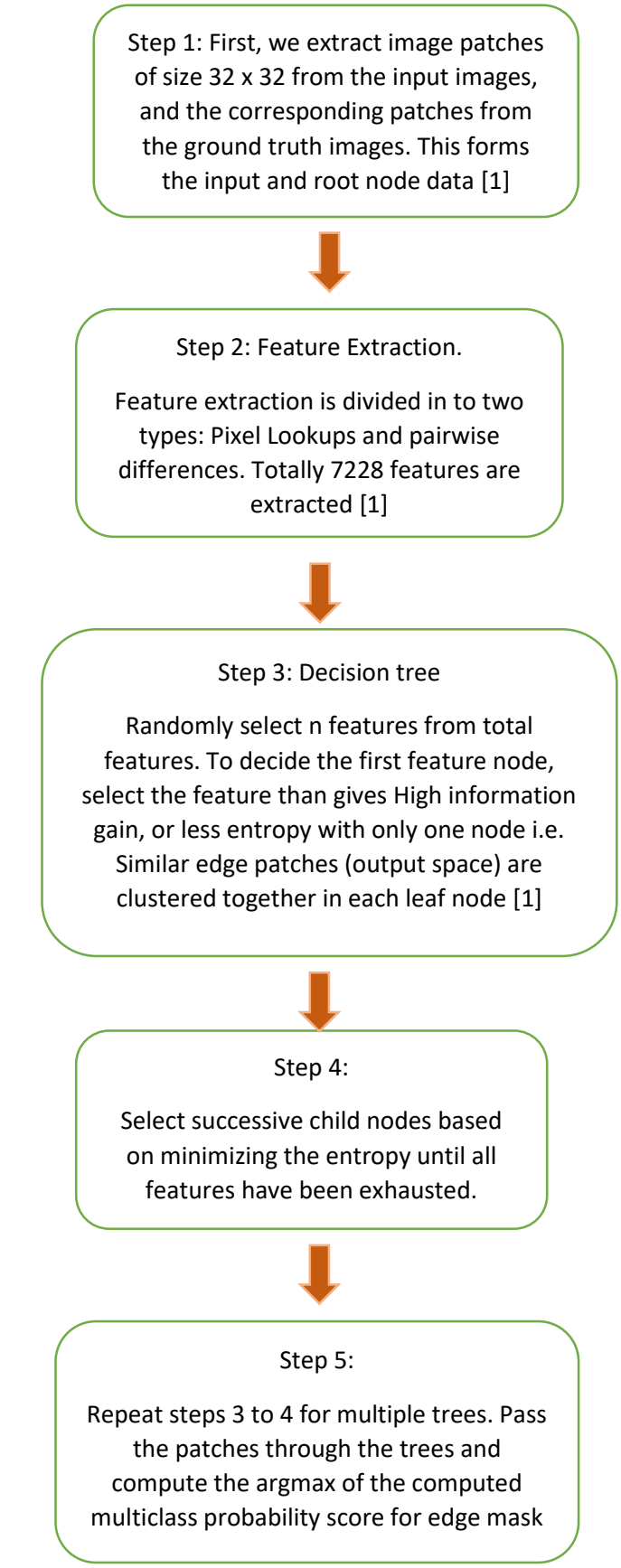


Figure 27: Flow-chart of Structured Edge Algorithm

Decision Tree and Random Forrest Classifier

Decision Tree:

Decision Tree learning is a machine learning approach for classification and regression and works as a binary tree as multiple levels [7]. The decision tree is essentially structure that will categorize data points into classes by multiple splitting based some condition evaluated at every level of the tree.

We start with a root node [7] that denotes the first entry point of the data to be classified. At the root node a decision criterion is checked, and based on the evaluation of the condition, the data entry travels down through the right child or the left child of the root node.

As the data sample traverses down the tree, it encounters internal notes where additional conditions are evaluated based on which the data traverses through the left or right child nodes. Eventually the data reaches a leaf node where a class probability outcome is obtained. In practical example, the input data has a set of 'd' features and C classes. Each feature is checked based on a threshold at every node. The leaf node then outputs a set of C classes.

Tree Construction:

The crucial part of constructing a decision is to choose which features are checked at the internal nodes and how to choose a threshold condition to ensure good performance of the tree.

A good attribute or feature to choose is one that splits the examples into subsets that are similar to each other. Mathematically we can say [9]

- A chosen attribute A, with K distinct values, divides the training set E into subsets E_1, E_2, ….., E_k.
- The Expected Entropy (EH) remaining after trying attribute A (with branches i = 1,2,…K) [9]

$$EH(A) = \sum_{i=1}^{K} \frac{p_i + n_i}{p+n} H\left(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i}\right)$$

For a dataset that consists of positive examples p and negative examples n

Entropy given by [9]

$$H\left(\frac{p}{p+n}, \frac{n}{p+n}\right) = -\frac{p}{p+n} Log2\left(\frac{p}{p+n}\right) - \frac{n}{p+n}\ log2\left(\frac{n}{p+n}\right)$$

For a dataset with more than 2 classes or outcomes, the above formula just replicates for additional classes.

We compute a quantity called Information Gain(I) or reduction in entropy as [9]

$$I(A) = H\left(\frac{p}{p+n}, \frac{n}{p+n}\right) - EH(A)$$

We then choose the attribute that gives the largest Information gain. This procedure is repeated at every internal node to choose the best attributes.[9]

The thresholds are randomly computes for each feature. It is observed that using Random Forest Classifier, using random splits actually works better for the classifier.

Random Forest Classifier:

The Random forest classifier is an extension of decision tree learning, where multiple decision trees are evaluated in parallel [6]. The Random Forest classifier is advantageous in cases when the Dimensionality of data and the number of training data are very large.

In essence, the Random Forests algorithms randomly picks a subset of the features and a subset of the training data with replacement and trains multiple decision trees in parallel [6].

Each decision tree outputs a probability and averaging the class probability outcome over a number of trees gives a metric to decide a class outcome for an example.

Analysis of Results

Figure 22 and 23 show the results obtained when Structured Forest Edge detection is applied on Zebra and Jaguar image. The parameters chosen are mentioned below

For Jaguar Image

```
model.opts.multiscale=0;          % for top accuracy set multiscale=1
model.opts.sharpen=2;             % for top speed set sharpen=0
model.opts.nTreesEval=4;          % for top speed set nTreesEval=1
model.opts.nThreads=4;            % max number threads for evaluation
model.opts.nms=1;                 % set to true to enable nms
```

The threshold used for Jaguar image is 0.17

For Zebra Image, the threshold was 0.176.

These thresholds where chosen as they provided the best F score on both the images. Further, reducing the number of trees decreased the F score and thus, the number of trees were set to 4 as recommended by the other in the paper. It can be seen from figures 22 and 23 than using NMS reduces the thickness of the edges and gives a results can be used to compare performance with the ground truth images.

Performance Evaluation

To compare the performance of various edge detector methods, we need to define a metric to evaluate each method. We define a good edge detection as one that predicts contours and edges that are of importance or priority to humans. In other words, the detected edges resemble those that are hand drawn by humans. We define quantities such as Recall, Precision and F score to evaluate the degree of similarity to the human segmented edges.

In addition, we classify each pixel in out Edge detected output as belonging to 1 of the four classes.

They are

True Positive: Edge pixels that are correctly predicted as edges.(Green)

True Negative: Non- edge pixels that are correctly predicted as non edge points.

False Positive: Non edge pixels that are predicted as edge pixels (Blue)

False Negative: Edge Pixels that are failed to be predicted by the edge detection algorithm. (Red)

We then define Precision and Recall as

$$\text{Precision} = \frac{No\ of\ True\ Positive}{No\ of\ True\ Positive + False\ Positive}\ ;\ \text{Recall} = \frac{No\ of\ True\ Positive}{No\ of\ True\ Positive + False\ Negative}$$

However there exists a trade off between obtaining high precision and high recall and we thus define a new metric called F to evaluate the performance of the Edge Detection Algorithm

$$F = 2\frac{Precision * recall}{Precision + Recall}$$

Thus if Precision and Recall are both equal to 1, we get F score as 1, which represent the best results.

Analysis of Results

Based on the results obtained from table 3, we can notice that based on F score, the Structured Edge technique outperforms the Canny Edge detector technique on both Zebra and Jaguar Image.

The F score for Jaguar using Structured Edge is 0.7572 whereas the F score using Canny Edge detector using parameters as Low th = 0.18, High Threshold = 0.4, sigma to 3 was 0.7527. Similarly, the F score for Zebra using Structured Edge is 0.5985 whereas the F score using Canny Edge detector using parameters as Low th = 0.15, High Threshold = 0.2, sigma to 3 was 0.5601.

Figure 24 shows the color coded output using the Structured Edge detector. It can be noticed that there exists a good number of Green pixels. This denotes True Positive. Since the Structured edge detector learns from human drawn labels, it succeeds in predicting these edges. However, it also predicts some red and blue pixels which affect the performance.

2. From the statistics on Table 3, it can be observed that is it easier to achieve higher F1 score in Jaguar image as compared to the Zebra image. The reason for this is because, the ground truth of Zebra images has segmented labels for the texture of grass and the background clouds. By nature of the Structured Edge learning, these texture are not detected, and thus contribute to a lot of False Negative. This greatly affects the Recall and brings down the F1 score.

3. Rational for F1 score

For good performance, we ideally want both Precision and Recall to be maximum. However, there exists a tradeoff between Precision and Recall in that, the Precision and Recall are approximately inversely proportional and thus a High Recall will correspond to a low Precision and vice versa as shown below.

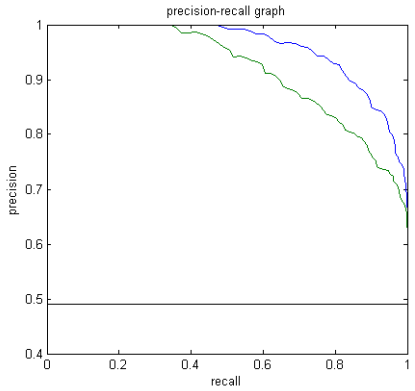


Figure 28: Non – Precision Recall Curve. Image source: https://www.mathworks.com/matlabcentral/mlc-

Thus, we define a Cost function such that we penalize low Recall and Precision non linearly with their corresponding inverse function. i.e 1/R and 1/P respectively.

Thus if R =1, we don't not penalize, but if R <1, we penalize using 1/R. Vice versa for P.

Thus the Total Penalty function can be taken as the average of penalty function for both Recall and Precision.

$$\text{Total penalty function} = \frac{1}{2}\left(\frac{1}{R} + \frac{1}{P}\right) = \frac{R+P}{2RP}$$

Since the F score is a reward function, we can equate it to the inverse of the penalty function

$$\text{Thus, F score} = \text{Total Penalty Function}^{-1} = \frac{2RP}{R+P}$$

Answers to questions in document

- No, it is not possible to get high F measure if Precision is significantly higher than Recall or vice versa.

If Recall is lower than Precision, the Penalty function will be high (1/R) and thus the F measure will be low as it is inverse of penalty function. The same reason can be given for the case when Precision is lower than Recall.

- Given that sum of Precision and Recall is a constant.

  Thus R + P = C

  We can write F score as

  $$F = \frac{2RP}{C} = \frac{2R(C-R)}{C} = 2R - 2\frac{R^2}{C}$$

  To maximize F, we differentiate the above equation with respect to R, and equate it to 0

  $$\frac{dF}{dR} = 2 - 4\frac{R}{C} = 0$$

  $$1 = 2\frac{R}{C}$$

  $$R = \frac{C}{2}$$

  Since R + P = C, P = $\frac{C}{2}$

  Thus R = P; Precision = Recall

References

[1] Piotr Dollar, C. Lawrence Zitnick, Structured Forests for Fast Edge Detection, ICCV 13

[2] Flann ~ http://docs.opencv.org/2.4/doc/tutorials/features2d/feature_flann_matcher/feature_flann_matcher.html

[3] Canny Edge Detector ~ http://docs.opencv.org/trunk/da/d22/tutorial_py_canny.html

[4] SIFT ~ http://docs.opencv.org/3.1.0/da/df5/tutorial_py_sift_intro.html

[5] SURF ~ http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_surf_intro/py_surf_intro.html

[6] Structured Forests slides ~ http://www.cs.utoronto.ca/~fidler/slides/CSC420/lecture5.pdf

[7] Decision Tree Learning ~ https://en.wikipedia.org/wiki/Decision_tree_learning

[8] Dimensionality Reduction – Machine Learning Course – Coursera by Andrew Ng, PCA

[9] Decision Tree learning CPSC540 course Slides, https://www.youtube.com/watch?v=-dCtJjlEEgM