EE 569: Digital Image Processing
Homework #2

Faiyadh Shahid
Student ID: 4054-4699-70
fshahid@usc.edu

October 11, 2015

Table of Contents

1 Problem 1

1.1 Abstract and Motivation

Texture analysis and classification is an important part of computer vision systems. It can actually be classified as one of the low-level computer vision problems, which uses various ideas derived from basic image processing problems. A block diagram below can roughly provide an idea on where we stand in terms of solving problems related to texture analysis (based on Dr. Kou's lecture):

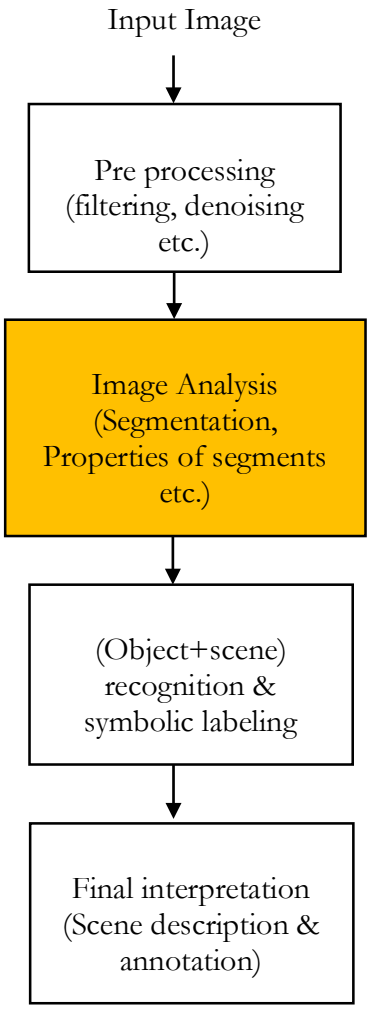

Figure 1.1.1: A rough block diagram of Computer Vision system

In order to analyze texture, we need to extract features from the images, which are used as training data for the testing set. One of the ways to implement extraction features from textures is using Law's filter from which we compute the local energy and use the feature vectors of each texture to form feature matrix of training data. These feature matrix is the base from which we develop an algorithm to recognize an unknown labeled texture.

In this problem, we will do texture classification in for two separate cases:

In the first case, there are 96 texture images of two types: grass and straw. 36 of them are labeled as grass and another 36 of them are labeled as straw. The rest 24 are left unknown. Using the concept of minimum mean distance classifier, we will find the texture of 24 unknowns.

In the second case, there are 192 (48x4) images divided equally among four texture classes: grass, leather, sand, and straw. We have to implement four binary classifiers with two methods: minimum mean distance classifier (like the first case) and support vector machine. In this case, we will randomly pick 36 samples in each class as training set and the rest as testing set. To train each classifier, we will use 36 training samples from the class of interest as positive and select 12 random textures from training samples for the other three classes as negative – calculating to a total of While testing for each classifier, we will use 12 training samples from the class of interest and select 4 from each testing samples for the other classes, in total of 24 images.

1.2 Approach and Procedure

1.2.1 Feature Extraction using Law's Filters

To extract features from a texture, Law's filter provides us with many types of filter kernels. In our case, we are provided with five 1D kernels: Level, Edge, Spot, Wave and Ripple [Table 1.2.1]. From these 1-D kernel, we create 25 filter banks which are then applied on the images to get feature vector. The steps followed to obtain the feature matrix of training and test textures are discussed below.

Table 1.2.1: 5x5 Law's 1-D filter kernel

1D Kernel Name	Filter Coefficients
L5 (Level)	[1 4 6 4 1]
E5 (Edge)	[-1 -2 0 2 1]
S5 (Spot)	[-1 0 2 0 -1]
W5 (Wave)	[-1 2 0 -2 1]
R5 (Ripple)	[1 -4 6 -4 1]

Before applying Law's filter, we have to perform a pre-processing step on every texture sample. In this pre-processing step, we remove the DC component of the samples since they do not contribute to providing information related to features. We find the local mean using a 5x5 window and subtract the local mean from the original intensity values. Thus we get rid of the dc component and have direct access to AC components which includes frequency information. Now we move towards applying the law's filter on the sample to obtain a feature vector.

Step-1: We create a filter bank that consists of 25 Law's filter. The filter bank is created by using tensor product of each 1D kernel going through various permutations, which leads us to get 5x5 =25 filters.

Step-2: After we have created 25 Law's filters, we apply those filters to each texture sample centering each pixel. In this case, we are required to do the common extension of boundaries and convolve the filter masks through the sample.

Step-3: We compute the energy content of each pixel obtained each law's filter content to form a 25-D feature for the underlying pixel. Next, we take the mean of all feature vectors to obtain a feature vector of size 25. We normalize based on the maximum and minimum values of the each feature.

This concludes the extraction of feature from a texture sample. We can repeat the similar procedure for all textures and save the obtained matrix in mat files to ease our computation time. In the first case, we get two matrices: training matrix of size 72x25 and test matrix of size 24x25.

1.2.2 Minimum Mean Distance Classifier (Using Mahanolobi's Distance)

The basic concept of minimum mean distance classifier dictates that for a number of classifiers, the classifier with the least distance is the outcome. For example: in our case of grass and straw classification, if the distance of a test texture from grass is less than that of straw, we will classify the texture as grass. In this case, we will use Mahanolobi's distance to compute the distance. Mathematically, Mahanolobi's distance is defined as follows:

$$d(\vec{x},\vec{y}) = \sqrt{(\vec{x}-\vec{y})^T S^{-1}(\vec{x}-\vec{y})}.$$

MATLAB comes with a built-in function to compute the distance: *mahal*. However, I created my own function in order to get an in-depth understanding of the calculation. I verified my function with MATLAB's built-in function. After computing the distances, we compare the distance for both grass and straw textures and the one with the least distance is the outcome of the unknown image.

1.2.3 Dimension Reduction using Principle Component Analysis (PCA)

It is seen from the above method of classification that we are actually using multiple dimensions without any reduction. However, not all features are equally important in a classification. Hence, we have a tool called Principle Component Analysis (PCA) which brings down the feature dimensionality. An obvious advantage of such tools is computational time, which is one of the important considerations in machine learning. The basic principle of PCA is to find directions, known as principle components, that maximizes the variance of the data. PCA projects the entire samples onto a different subspace. A summary of general steps followed in implementing PCA are as follows [1]:

a) Standardize the samples.
b) Compute covariance matrix and obtain eigenvalues and eigenvectors via singular vector decomposition.

c) Sort the eigenvalues in descending order which implies that the one at the top has the highest variance.
d) Choose *k* largest eigenvalues where *k* is the reduced dimension and keep the corresponding eigenvectors. In our case, we choose the first one for *k=1* and the first three for *k=3*.
e) Construct a projection matrix from the selected eigenvectors and transform feature vector via projection matrix to a *k*-dimensional subspace.

MATLAB has a built-in function called *pca* provided with Statistical and Machine Learning Tools in 2015b. We apply the function to our feature matrix of training and test set. We will obtain a projection matrix of 25x25 from which we choose the first row in the first case and the first three rows in the second case. MATLAB sorts the coefficients in descending ordering corresponding. After that, we apply minimum-mean distance classification via Mahanolobi's distance to find out the classification of unknown labels.

1.2.4 Dimension Reduction using Linear Discriminant Analysis (LDA)

Linear Discriminant Analysis (LDA) is used for dimensional reduction like PCA. However, it works on different principle from PCA. The main principle of LDA is to find directions, known as linear discriminants, with an aim to maximize separation between classes. LDA considers class labels whereas PCA does not. Hence, it is called a 'supervised' algorithm. A summary of general steps followed in implementing PCA are as follows [2]:

a) Compute *d*-dimensional mean vectors for different classes in a dataset.
b) Compute intra-class and infra-class scatter matrices.
c) Compute the eigenvalues and eigenvectors for each scatter matrix.
d) Sort the eigenvalues in a descending order and select *k* eigenvectors with the largest eigenvalues to create a transformation matrix.
e) Transform feature vector via projection matrix to a single dimensional subspace.

MATLAB has a built-in function called *fitcdiscr* provided with Machine Learning Tools in 2015b. To apply the function, we have label the classes in our matrix for the training dataset only. We will obtain two coefficients calculated for each class separately by the function. These are the coefficients specified for each class. After that, we apply minimum-mean distance classification via Mahanolobi's distance to find out the classification of unknown labels. The output of LDA depends on the number of classes *C* and can be reduced at most *C-1*.

1.2.5 Classification using Support Vector Machine (SVM)

Support Vector Machine (SVM), one of the advanced machine learning tools, is widely used in solving classification problems. The basic principle of SVM is to find a separating hyperplane that aims in finding classification. The aim is to achieve a separating hyperplane such that the distance between hyperplane and the closest points close to decision boundary are optimized. Although solving the mathematics of SVM is beyond the scope of this report, a higher level of understanding is necessary to visualize the classification. The basic procedure behind SVM is training a model such that

an optimized hyperplane is achieved. Next, we find out from the hyperplane the classification of an unknown label.

To implement this part, I used **libsvm-3.20.2** library in MATLAB [3]. Two functions were used to solve this part namely **svmtrain** and **svmpredict**. The function **svmtrain** is used to train the training set with class labels assigned (+1-classifier & -1- not-classifier). Next, **svmpredict** is used on the testing set to find the predicted label. If the predicted label was +1, the outcome is a 'Yes' for the classifier. Otherwise, it is a 'No' for the classifier. To be mentioned, the kernel was chosen to be **linear** and the data was scaled as per the guidelines provided by the author of the code. The default setting was for C-SVC.

1.2.6 Implementation in programming

Case 1 The implementation of this part in programming was quite straightforward once all the above-mentioned task is performed. The steps followed are jotted below:

a) Apply Law's filter to extract features of 36 grass and 36 straw training textures and form a 72x25 feature matrix. Similarly, create a 24x25 feature matrix of unknown labels. These are saved in mat files to save computational energy.
b) Use Mahanolobi's distance to find out which classifier the unknown labels belong to. Thus, a classification is performed without any feature reduction.
c) Reduce the dimensions of feature vector of training set to 25x25 and select the first row as a reduced feature vector using PCA. Use Mahanolobi's distance based on PCA to find out about classification of unknown labels. Similarly apply LDA and find out about classification. A detailed description of each procedure is provided above.
d) Finally compare the results.

Case 2 For this case, I generalized the entire problem based on classifiers. The user is asked to provide an input regarding the classifier they want to test for: Grass, Leather, Sand and Straw [Fig. 1.2.1]. After the user input is provided, the program randomly selects training sets and testing sets according to the described problem. The program has been written such that it sorts the testing files in orders of classifiers first and non-classifier later. For example: if the user input is Leather, the test files are chosen in order of 12 leather files, 4 grass, 4 sand and 4 straw files without repetition in other files. It is to be mentioned that the computer system does the sorting based on matrix indices. It does not have any prior knowledge regarding the texture classification. All it is dealing with a feature vector of texture.

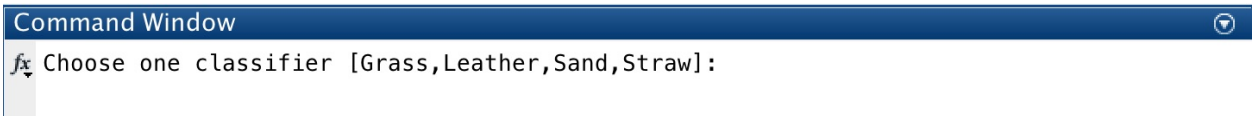

Figure 1.2.1 User input asked for classifier testing

The randomness of choice in training sets and testing sets are created using *randperm*, which returns a vector containing a random permutation of the integers 1: N. An example can be used from the code to explain how this works [Fig. 1.2.2]:

```
r = [1:48];
train_ind_classifier = sort(randperm(48,36));
test_ind_classifier= setdiff(r,train_ind_classifier);
```

Figure 1.2.2 A snippet of random selection

Our classifier contains 48 images whose feature vectors are saved apriori. The program selects randomly any 36 values from 1 to 48 and they are sorted. After that, the testing set is formed by choosing the rest of the values using a function *setdiff*. Hence, we form random training and testing data sets. I also implemented program for each classifier by via manual random selection of files. However, I felt it was too naïve to do it that way. Hence, I implemented random selection based program. The results when compared were similar.

After the training and test set are selected, the program runs through a similar set of steps like Case-1 where it solves for minimum mean distance classification without any feature reduction, using PCA and LDA. In this case, PCA dimension is reduced to $k=3$. An added function to this program is SVM whose implementation is discussed above. After the program is completed, it provides output in the form Yes or No for the classification set by the user.

1.3 Experimental Results

1.3.1 Two Classes + Minimum Mean Distance Classifier

The result obtained for Grass vs. Straw classifier on the unknown labels is provided below in Table 1.3.1:

Table 1.3.1: Results of classification for two classes (Case-1)

File Name	No feature reduction	Reduction w/o class label (PCA)	Reduction w. labeled training data (LDA)	Manual annotation
'unknown_01.raw'	'Straw'	'Straw'	'Straw'	Straw
'unknown_02.raw'	'Straw'	'Straw'	'Straw'	Straw
'unknown_03.raw'	'Grass'	'Grass'	'Grass'	Grass
'unknown_04.raw'	'Grass'	'Grass'	'Grass'	Grass
'unknown_05.raw'	'Grass'	'Grass'	'Grass'	Grass
'unknown_06.raw'	'Straw'	'Straw'	'Straw'	Straw
'unknown_07.raw'	'Grass'	'Grass'	'Grass'	Grass
'unknown_08.raw'	'Straw'	'Straw'	'Straw'	Straw
'unknown_09.raw'	'Straw'	'Straw'	'Straw'	Straw
'unknown_10.raw'	'Grass'	'Grass'	'Grass'	Grass
'unknown_11.raw'	'Grass'	'Grass'	'Grass'	Grass
'unknown_12.raw'	'Grass'	'Grass'	'Grass'	Grass
'unknown_13.raw'	'Straw'	'Straw'	'Straw'	Straw
'unknown_14.raw'	'Grass'	'Grass'	'Grass'	Grass
'unknown_15.raw'	'Straw'	'Straw'	'Straw'	Straw
'unknown_16.raw'	'Straw'	'Straw'	'Straw'	Straw

'unknown_17.raw'	'Straw'	'Straw'	'Straw'	Straw
'unknown_18.raw'	'Straw'	'Straw'	'Straw'	Straw
'unknown_19.raw'	'Grass'	'Grass'	'Grass'	Grass
'unknown_20.raw'	'Grass'	'Grass'	'Grass'	Grass
'unknown_21.raw'	'Grass'	'Grass'	'Grass'	Grass
'unknown_22.raw'	'Straw'	'Straw'	'Straw'	Straw
'unknown_23.raw'	'Straw'	'Straw'	'Straw'	Straw
'unknown_24.raw'	'Grass'	'Grass'	'Grass'	Grass

Based on the manual annotation, we can create a class-confusion matrix to check for the error rate:

Table 1.3.2: Class-confusion matrix

		1	2	Error rate (%)
Grass	1	12		0%
Straw	2		12	0%

Here entries are numbers. For example: Grass is labeled as 1 and there are 12 grass textures have been correctly classified and not a single texture was misclassified.

1.3.2 Multi-Classes + SVM

In this part, the program was tested for 10 times for each classifier (40 testing samples). Although there were some errors found at certain classifiers but they were negligible (4-8% error). The report will include results for 7th test trial since that is where a reasonable amount of error was found. Random selection of data sets by the program has made it possible for us to check for the entire data set's possible outcome.

Table 1.3.3: Results for multi-classes (7th test trial)

Texture	Method	Grass vs Non Grass	Leather vs non-leather	Sand vs non-sand	Straw vs non-straw
Classifier Image 01	No reduction	Yes	Yes	Yes	Yes
	PCA	Yes	Yes	Yes	Yes
	LDA	Yes	Yes	Yes	Yes
	SVM	Yes	Yes	Yes	Yes
Classifier Image 02	No reduction	Yes	Yes	Yes	Yes
	PCA	Yes	Yes	Yes	Yes
	LDA	Yes	Yes	Yes	Yes
	SVM	Yes	Yes	Yes	Yes
Classifier Image 03	No reduction	Yes	Yes	Yes	Yes
	PCA	Yes	Yes	Yes	Yes
	LDA	Yes	Yes	Yes	Yes
	SVM	Yes	Yes	Yes	Yes
Classifier Image 04	No reduction	Yes	Yes	Yes	Yes
	PCA	Yes	Yes	Yes	Yes
	LDA	Yes	Yes	Yes	Yes

	SVM	Yes	Yes	Yes	Yes
Classifier Image 05	No reduction	Yes	Yes	Yes	Yes
	PCA	Yes	Yes	Yes	Yes
	LDA	Yes	Yes	Yes	Yes
	SVM	Yes	Yes	Yes	Yes
Classifier Image 06	No reduction	Yes	Yes	Yes	Yes
	PCA	Yes	Yes	Yes	Yes
	LDA	Yes	Yes	Yes	Yes
	SVM	Yes	Yes	Yes	Yes
Classifier Image 07	No reduction	Yes	Yes	Yes	Yes
	PCA	Yes	Yes	Yes	Yes
	LDA	Yes	Yes	Yes	Yes
	SVM	Yes	Yes	Yes	Yes
Classifier Image 08	No reduction	Yes	Yes	Yes	Yes
	PCA	Yes	Yes	Yes	Yes
	LDA	Yes	Yes	Yes	Yes
	SVM	Yes	Yes	Yes	Yes
Classifier Image 09	No reduction	Yes	Yes	Yes	Yes
	PCA	Yes	Yes	Yes	Yes
	LDA	Yes	Yes	Yes	Yes
	SVM	Yes	Yes	Yes	Yes
Classifier Image 10	No reduction	Yes	Yes	Yes	Yes
	PCA	Yes	Yes	Yes	Yes
	LDA	Yes	Yes	Yes	Yes
	SVM	Yes	Yes	Yes	Yes
Classifier Image 11	No reduction	Yes	Yes	Yes	Yes
	PCA	Yes	Yes	No	Yes
	LDA	Yes	Yes	No	Yes
	SVM	Yes	Yes	Yes	Yes
Classifier Image 12	No reduction	Yes	Yes	Yes	Yes
	PCA	Yes	Yes	Yes	Yes
	LDA	Yes	Yes	Yes	Yes
	SVM	Yes	Yes	Yes	Yes
Non-classifier 01	No reduction	No	No	No	No
	PCA	No	No	No	No
	LDA	No	Yes	No	No
	SVM	No	No	No	No
Non-classifier 02	No reduction	No	No	No	No
	PCA	No	No	No	No
	LDA	No	No	Yes	No
	SVM	No	No	No	No
Non-classifier 03	No reduction	No	No	No	No
	PCA	No	No	No	No
	LDA	No	No	No	No
	SVM	No	No	No	No
Non-classifier 04	No reduction	No	No	No	No

	PCA	No	Yes	No	No
	LDA	No	Yes	No	No
	SVM	No	No	No	No
Non-classifier 05	No reduction	No	No	No	No
	PCA	No	No	No	No
	LDA	No	No	No	No
	SVM	No	No	No	No
Non-classifier 06	No reduction	No	No	No	No
	PCA	No	Yes	No	No
	LDA	No	No	Yes	No
	SVM	No	No	No	No
Non-classifier 07	No reduction	No	No	No	No
	PCA	No	No	No	No
	LDA	No	No	No	No
	SVM	No	No	No	No
Non-classifier 08	No reduction	No	No	No	No
	PCA	No	No	No	No
	LDA	No	No	No	No
	SVM	No	No	No	No
Non-classifier 09	No reduction	No	No	No	No
	PCA	No	No	No	No
	LDA	No	No	No	No
	SVM	No	No	No	No
Non-classifier 10	No reduction	No	No	No	No
	PCA	No	No	No	No
	LDA	No	No	No	No
	SVM	No	No	No	No
Non-classifier 11	No reduction	No	No	No	No
	PCA	No	No	No	No
	LDA	No	No	No	No
	SVM	No	No	No	No
Non-classifier 12	No reduction	No	No	No	No
	PCA	No	No	No	No
	LDA	No	No	No	No
	SVM	No	No	No	No

Table 1.3.3 indicates samples for which classification was wrong. It can be seen that error rate was not too high and was not existent in every testing sample. Since none of the methods except for PCA and LDA had errors, we can calculate the error rate for LDA & PCA in case of two classifiers: one is Leather vs. non-leather where the error rate is around 8% for both and the other is Sand vs non-sand where the error is 12% for LDA and 4% for PCA. This error rate indicates that there were only two or three errors at maximum for each classifier. It is to be added that 10 choice of random samples is not sufficient enough to do a statistical study. However, it is roughly good enough to get an idea of error rate.

1.4 Discussion

Based on the performance above, it can be said that all the methods did a good job in classification relatively in both cases. It can be argued that since the testing data set is smaller compared to training data set (1:3 ratio), not much discrepancy can not be observed in results. Apart from that, if we visually look at the texture of grass and straw, we can see that there are certain distinctive features in both textures such as levels, which can be used to test the texture classification. Certain errors are obtained in LDA and PCA where multiple classes are added. It can be argued that since both LDA and PCA reduces the dimension to a significant amount, it is possible to loose out certain feature information which was crucial in making the classification work. However, an 8-12% is not a huge error rate.

If one method had to be chosen out of all methods, the method I would go with SVM. The method with no feature reduction is also an option. However, the time consumption in the method makes it harder for the user to stay with it. In our case, there were little data to deal with; but in real world we deal with more datasets in the range of thousands to the least. So, the procedure of pattern with no feature reduction makes it quite difficult for the programmer to deal with. This is what we call the ***curse of dimensionality***. SVM is an optimized tool that can deal with huge amount of datasets and computational time can be decreased by reducing the dimensions. Although reducing dimensions at times may occur in providing wrong result as seen in the above problem, it helps us to avoid over fitting by minimizing the error in estimation of parameter ***(curse of dimensionality)***.

Are all features equally important? There are 25 different combinations of features obtained from Law's filters. However, it is computationally hard to go calculate for each feature. Apart from that, it is not necessary to consider all the features in a classifier based on the type of problem. As mentioned above, for the first case it is easy to deduce the type of features that will help in distinguishing the class type of each texture. For the second case, we need more information since it involves multiple classes. In short, the answer to the question is negative.

What is the dominant feature? In the first case, the dominant feature was found to be in Levels **(L^TL).** For this, a code was written which will distinguish between all the features in a surface plot for 36 grass samples and 36 straw sample and give us a 3D plot showing the dominant portion of the features. The surface plot for both sample types are shown in Figure 1.4.1.

Here we can observe the first value for each texture of 36 numbers is higher in values compared to all other features. Hence, Levels is the dominant feature for both grass and straw images. This particular feature can also be distinguished to differentiate between two textures. Plotting this particular feature element in a 2-D graph, we find that the values are far apart from each other in Figure 1.4.2.

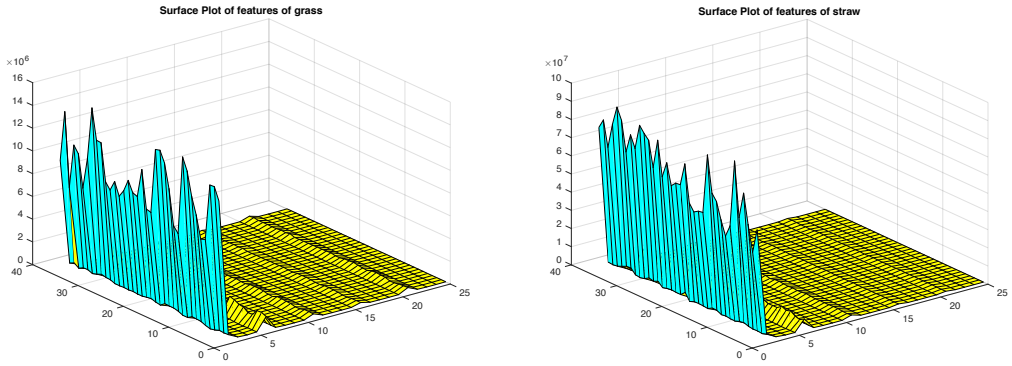

Figure 1.4.1: Surface plot of feature vectors of Grass [Left] & Straw [Right]

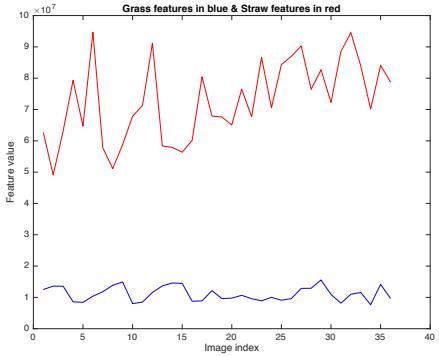

Figure 1.4.2: Plot of Levels feature for Straw [Top] & Grass [Bottom]

It was seen from plotting all the feature vectors that they were distinctively apart from each other, which actually argues for why the error rate is too low or there is zero error rate for the first case. The classification accuracy was found to be 100% when class labels for first 12 was represented with +1 and last 12 with -1.

PCA vs LDA It is to be added that LDA may sound superior to PCA; but there are cases where PCA can outperform LDA if the number of samples per class is relatively small [4]. In our case, the number of samples per class are smaller than a minimum statistical sample requirement. Hence, it is not necessary that LDA is the right solution for every problem. While discussing about PCA and LDA, it has to be added that there has to be higher correlation between dataset to get a better PCA and LDA feature reduction since both of them mathematically deal with covariance.

2 Problem 2

2.1 Abstract and Motivation

Edge detection is one of the most fundamental tasks in image processing and computer vision. It is an important pre-processing step in every sort of processing tasks aimed to achieve various objectives such as, image segmentation, object recognition, finding active contours and so on. Although we have been able to define edges in mathematical term to a certain extent, there is not a single edge detection algorithm which have been able to solve the problem for once and all. Hence, we have various edge detection algorithms – few of which will be discussed here.

The first algorithm that will be discussed is known as Sobel Edge detector. The basic concept of this algorithm is based on the gradient of image intensity, which detects directional change in intensity. In mathematical terms, a gradient is a two-dimensional equivalent of the first derivative also known as gradient vector. In case of image, we obtain gradient vectors in two directions: one in horizontal direction (x) and the other in vertical direction (y). From the gradient vectors, we obtain two important information related to gradient: gradient magnitude and gradient direction:

$$\nabla f(x,y) = g = \begin{bmatrix} g_x \\ g_y \end{bmatrix}$$

$$|g| = \sqrt{g_x^2 + g_y^2} \text{ (Gradient magnitude)}$$

$$\theta = \tan^{-1}\left(\frac{g_y}{g_x}\right) \text{ (Gradient direction)}$$

The magnitude provides information about the strength of the edge. The direction of gradient is orthonormal to the direction of the edge. All of these parameters are applied in finding information related to edges in an image. It is to be mentioned that we are dealing with discrete values of images. Hence, we should think of every differential operators as difference operators (used in discrete cases).

The second algorithm that will be discussed is known as Canny Edge detector. The detection uses multi-stage algorithm to detect edges in an image. It was first developed by John Canny, who took the problem of edge detection aiming to achieve two of the most common criterions in edge detection. The first criterion is to detect all edges in an image with low error rate such that there are no spurious responses. The second criterion is to localize edge points well such that the distance between edges marked by detector and "center" of the true edge must be minimized. He also added a third criterion which ensures that the detector has only one response to a single edge [5]. Based on these criterions, the detection algorithm is developed which ensures more reliability as well as simplicity of calculation following procedural steps. The steps are jotted down in Approach and Procedure [2.2.2].

The final algorithm that will be discussed is known as Structured Edge (SE) detector developed by Piotr and Zitnik [6]. The algorithm is implemented based on an advanced classifier known as random forests, which functions on decision trees during the training phase via multiple learning

algorithms (called ensemble learning) and provides a classification based on the nodal results. In our case, the output of the classification provides us with the output whether a particular pixel is an edge or not. The result provided by SE detector seems to have done a better performance with lesser computational time compared to the other state-of-the art algorithms. We will observe the performance of the algorithm on our provided images [Farm & Cougar].

2.2 Approach and Procedure

Before we dive deep into procedures used to implement the algorithms, our experimental results are applied on two images called Farm and Cougar. Two of the images are provided below for our reference. Along with images, the histogram of both images are attached as well [Fig. 2.2.1 & 2.2.2]

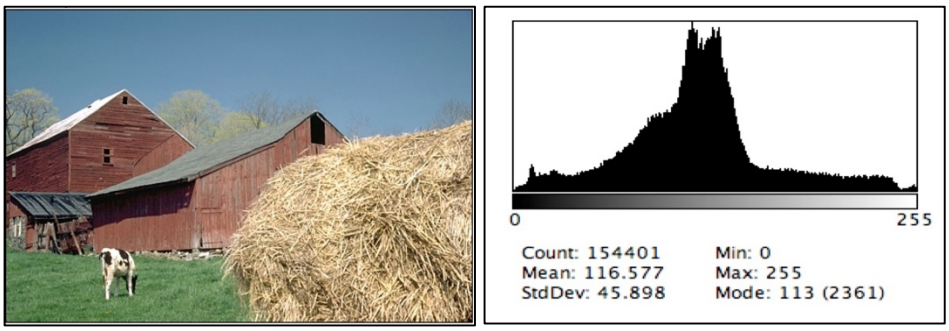

Figure 2.2.1: [Left] Farm (Original Image); [Right] Histogram of Farm image

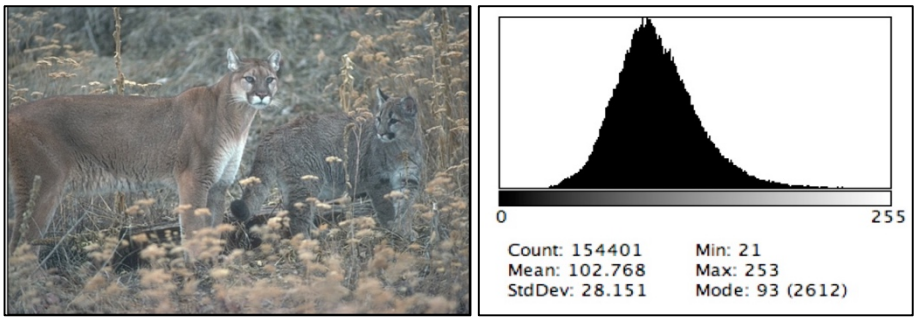

Figure 2.2.2: [Left] Cougar (Original Image); [Right] Histogram of Cougar image

2.2.1 Sobel Edge Detector & Non Maximal Suppression

Although implementation of Sobel edge detector may sound complex on a conceptual basis, its implementation is quite straightforward. The basic implementation of the detector is using two kernels called G_x and G_y passed through the entire image. The described implementation can be thought as convolution between image and kernel. The kernels used for Sobel operation are as follows:

$$G_x = \begin{vmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{vmatrix}; G_y = \begin{vmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{vmatrix}$$

To implement this method, we have to convert the image into a grayscale using using proportions of different channels. For this, we used the generally accepted grayscale conversion formula, which uses

21% of red channel, 72% of green channel and 7% of blue channel. This is equivalent of **rgb2gray** in MATLAB. After converting into a grayscale image, we apply the aforementioned Sobel masks in both directions to get the gradient vectors. While the kernel is applied, it is divided by 4 to keep the values of image intensities normalized to 0-255. Then, we find the gradient magnitudes using the gradient vectors with normalization between 0 & 255. Hence, we obtained our edge detected image. Our obtained images produce white edges with black background. However, we can create black edges with white background by subtracting our obtained image from 255 (maximum value) and converting it into unsigned 8-bit integer *('uint8')*.

Thresholding We can get better performance by applying proper thresholds such that certain percentage of pixels in image are edge points. To implement this, we need to find the threshold intensity values above which the edges will only exist and the intensity value goes to zero below the threshold value. One way to solve for the threshold value is to find the intensity value from the cumulative histogram of the image [Fig. 2.3]. In our case, both of our images has size of 321x481 (WxH) with one dimension, summing to 154401 pixels. For 10% and 15% threshold, the number of pixels that should convert to zero are approximately 138961 and 131241 pixels respectively. From the cumulative histogram of both images, we observe that the threshold values are as follows:

Farm: 10% threshold- 56 & 15% threshold – 45
Cougar: 10% threshold- 30 & 15% threshold – 25

To be noted, the cumulative histograms in Figure 2.2.3 are plotted from a function written by me and the intensity values are shifted to the right by 1 since MATLAB indexing starts from 1.

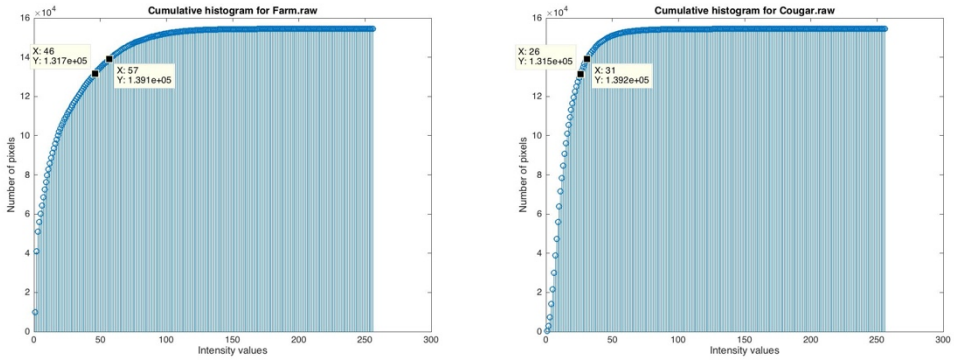

Figure 2.2.3: Cumulative Histogram of [Left]: Farm.raw; [Right] Cougar.raw

It is worth mentioning that the threshold values are obtained from the Sobel edge applied image; not from the original black and white image.

Non Maximal Suppression (NMS) Non Maximal Suppression (NMS) is a technique widely used for thinning edges such that local maxima (sharp changes in a gradient direction) are identified and all other gradient values are set to 0. The implementation of this technique is purely based on the usage of gradient directions found in all neighboring pixels relative to the central pixel. Hence it can be identified that there are eight cases we need to solve [Fig. 2.2.4].

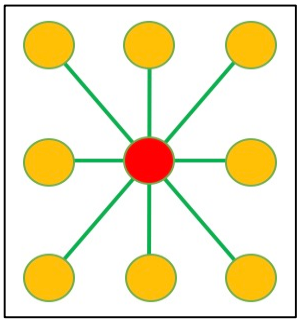

Figure 2.2.4: Visualization of Non Maximal Suppression (NMS)

The red circle in the figure represents the central pixel and the other circles are neighboring pixels. We solve for eight 45-degree blocks in a clockwise manner with angles valued from 0 to pi. To implement this, we first find the gradient directions of the images whose edges we are going to thin. Then, we create a mesh grid of width and height indices of the image so that we can access each and every coordinate of the image. Along with that, we also create a mask of the size of image. Next, we compare the gradient magnitude of each edge point with its neighbors along the dominant gradient direction. If it is greater than both neighbors, we preserve the value; otherwise we discard it. If it is cutting in the middle, we do linear interpolation as follows:

$$m(r, c) = a * (1 - d) + b * d$$

where a and b are the neighboring pixels; d is the gradient direction and m is the mask on which the value is stored. Finally, we multiply the image with the masks on which all evaluations are stored.

2.2.2 Canny Edge Detector

Canny Edge detector uses multi-stage algorithms to find wide range of edges in an image. The procedural steps used in detecting edge points fulfill three criterions set by John Canny as mentioned above [5]. The steps used by the detector are as follows:

Step 1: We apply a low-pass filter like Gaussian filter to remove the noise from the image. This is a pre-processing step. However, it does not guarantee complete removal of noise.

Step-2: We obtain gradient magnitude of the image using any derivative-based operator such as Sobel operator.

Step-3: We apply Non Maximal Suppression (NMS) to thin edges in the output to one-pixel wide edges such that spurious response to edge detection are removed.

Step-4: We apply double threshold since the image obtained from NMS still contains edges created due to noise. In this case, we filter out the edge pixel with the weak gradient value and preserve the edge with high gradient value. This step is where we fine tune our threshold to get proper edge detected image.

Step-5: In the final step, we do edge tracking by hysteresis aimed to connect strong edge points and throw away isolated weak edge pixels.

There is a built-in function in MATLAB that can be used to apply Canny Detector to an image. The function is called ***edge*** which requires a grayscale image and supports six different edge-finding methods: Sobel, Prewitt, Roberts, Laplacian of Gaussian, zero-cross method and Canny method. It

returns a binary image with 1's where there are edge points and 0's elsewhere. In the ***edge*** function, we need to provide our parameters as follows:

```
I = edge(BW,'canny',[low_threshold high_threshold]);
```

where *I* represents the binary-returned image, *BW* is the input image, low and high thresholds are parameters that we use to fine tune our edge detection. MATLAB can also find threshold which provides better performance as canny detector. The concept of low threshold and high threshold is that it assigns 1 to any value higher than high threshold and 0 to any value lower than low threshold. If the value is between low threshold and high threshold, the pixel is assigned 1 only if it can be connected to a pixel larger than high threshold through a stream of pixels which has values larger than low threshold. It is assumed that the image is converted to a probability edge map (normalized between 0 & 1).

2.2.3 Structured Edge (SE) Detector

Before we discuss the Structured Edge Detector algorithm, we need to have some basic idea about the construction of decision tree and how Random Forest classifier works. So, let us get started.

Decision Tree Construction A decision tree is used as a tool to reach a conclusion on a particular object (in our case image patches) based on criterions set by the algorithm. Given several features, the algorithm predicts the target value of a testing variable based on the tree created. The decision tree has essentially two types of nodes. One is leaf node, which has a class label and the other is non-leaf (interior) node, which has a question on features branching to various answers. The answer can be either a simple binary (yes or no) or multiple class (e.g. color: red, green, blue).

The basic intuition behind the construction of decision tree lies on the questions asked on various non-leaf nodes and the split that provides the final target value of the testing variable. To measure the degree of purity induced by question, we use a quantity obtained from Shannon's basic information theory called **mutual information**. The calculation of mutual information requires other metrics to be calculated such as, entropy and conditional entropy. In information theory, entropy implies the average of information contained in an instance and conditional entropy implies the information that is required to know the outcome of a random instance provided that we have an instance or instances. Mutual information of random variables implies a measure of variable's mutual independence. Here, instance can be thought of as a dataset or image patches. Let's suppose we have k values of labels denoted by $Y = \{y_1, y_2, y_3, \ldots. y_k\}$ and X denotes a question with v answer., then we calculate the above mentioned parameters as follows [7]:

$$H(Y) = \sum -\Pr(Y = y_i)\log_2 \Pr(Y = y_i)$$

$$H(Y \mid X = v) = \sum_{i=1}^{\hat{}} -\Pr(Y = y_i \mid X = v)\log_2 \Pr(Y = y_i \mid X = v)$$

$$I(Y;X) = H(Y) - H(Y \mid X)$$

where $H(Y)$ is the entropy of Y class labels; $H(Y \mid X = v)$ is the conditional entropy of Y given that we know about X, $I(X,Y)$ is the mutual information between X and Y.

After calculating all the parameters, our objective is to find the optimum information gain, which will provide us with more information related to labels. In short, the decision construction tree accomplished for Random Forest classification greedily ask questions and reaches to pure child node (target value) by maximizing the information gain.

Random Forest Classifier The construction of decision tree learning is the preliminary basis of RF classifier. This classifier was first introduced by Leo Breiman. The classifier essentially considers multiple decision trees and averages the results with an aim to decrease the variance. After multiple decision trees are constructed, the classifier applies a technique of bagging, which will be described in mathematical terms in short. Let us suppose we have a training set $Y = \{y_1, y_2, y_3, \ldots. y_N\}$ with class labels $C = \{c_1, c_2, c_3, \ldots. c_N\}$. Then, we do a random selection of features from Y. After the random selection, we apply the following algorithm to get a random forest tree:

For $b = 1:B$

a) Create a sample with replacement of data size N from Y, C and let us call it Y_b, C_b (bootstrapping).
b) Train a decision tree T_b based on the bootstrapped data Y_b, C_b .

In this way, we create T trees all of which are trained independently. During testing, each test point $\boldsymbol{v}$ is passed through the trees until it reaches a target value for each. Finally, we take the average of all the target values and find our class labels for the training sets. It has to be mentioned that the procedure followed to obtain a random forest tree is based on bootstrap aggregating (bagging). However, it takes a substantial different scheme from the bagging procedure by selecting features included in Y randomly [8].

SE Detection Algorithm [3]

The flow chart for SE detection algorithm is shown in Figure 2.2.5 [9]. In the flow chart, the task of training datasets is not included since it is done at an earlier stage.

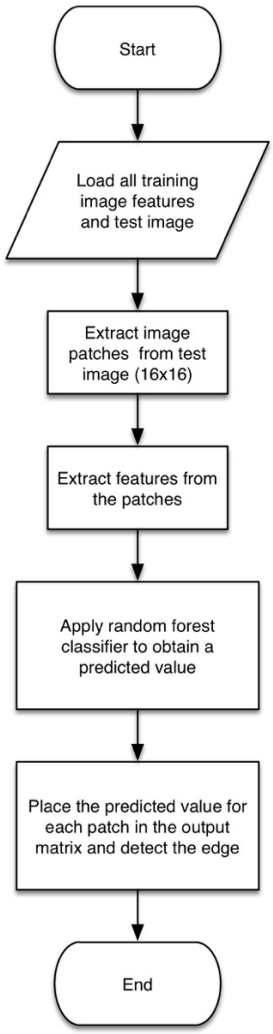

Figure 2.2.5: High level flow chart of SE detection algorithm

The flowchart above provides a high level operation of SE detection algorithm. Before the detection is executed, various training images are selected and patches (**P-matrix)** from each of them are extracted. After that, we extract features from the matrix to a **vector v**. Hence we have a training set which is used in Random forest classifier to provide classification for edges of test images. The parameters that we are dealing with in this algorithm are image and label patch size, channel and feature parameters, and decision forest parameters (stopping criteria, number of trees). In programming, training would take a lot of time to be implemented. So, they are done early so that we can directly input the image and get our desired edge outcome. In the program provided for SE detection, there is an option of adding Non-Maximal Suppression to the detected edge map. We obtain a probability edge map, which is converted to binary edge map by taking a threshold where the edges provide a better result.

2.2.4 Performance Evaluation

The performance of an edge detector can be measured using the concept of recall and precision, importantly used in machine learning to evaluate classifiers. The edges detected through programming are based on algorithms where more and more conditions are added to make the performance better. However, the edges detected by algorithm can never be as good as a human detected edges. These edge detected maps are called **ground truths**. We consider multiple ground truth images since different people may have different perception of edge in an image. While calculating the performance evaluation we take the mean of a certain performance measure with respect to each ground truth i.e. mean precision & mean recall. The calculation of precision and recall is done using the following formula:

$$Precision: P = \frac{True\ Positive}{True\ Positive + False\ Positive}$$

$$Recall: R = \frac{True\ Positive}{True\ Positive + False\ Negative}$$

where true positive means the edge pixels detected by algorithms (Sobel, Canny, SE detector) will map with ground truth, false positive means edge pixels in the edge map correspond to non-edge pixels in the ground truth and false negative means non-edge pixels in the edge map correspond to the true edge pixels in the ground truth. We don't consider true negative since it does not contribute to the evaluation of edge detection. Based on the recall and precision, we can formulate a performance evaluation matrix called F-measure:

$$F = 2 * \frac{P * R}{P + R}$$

To evaluate edge detection performance, we use the *edgeEvalImg* function of Piotr's code of SE detector algorithm:

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

The parameters that are given as an input to the function are the edge map detected by algorithms (**E**) and a ground truth *cell* (1x5) created using ground truth images provided for Farm and Cougar (`'G_Farm.mat'` & `'G_Cougar.mat'`). Ground Truth cell is created separately for each image. Inside the cell, we have five ground truth images in **struct** format. If we let the function run using its default function, it measures for a threshold of 99, which takes a lot of time implement. However, we can control the threshold by adding a parameter in struct format in `varargin.` For Sobel and Canny edge detection, it is enough to use 1 as threshold. For SE detection, we can either use probability edge with 99 as threshold or we can create a binary edge map and use 1 as threshold. To be added, the given ground truth images had black edges with white background. To keep it consistent for the function used, the ground truth image is inverted logically (0 for 1 and 1 for 0).

2.3 Experimental Results

Figure 2.3.1 shows Sobel Edge detection (Gradient Magnitude) on Cougar and Farm image.

Figure 2.3.1: Gradient Magnitude of Sobel operator on [Left] Farm & [Right] Cougar

Figure 2.3.2 shows the result after Thresholding for 10% and 15 % on Farm image.
Figure 2.3.3 shows the result after apply Non-Maximal Suppression on Farm image.

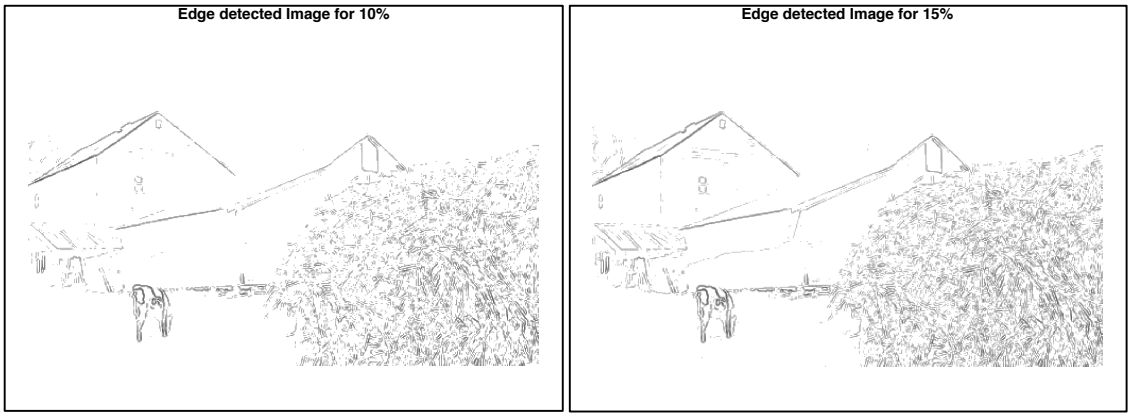

Figure 2.3.2: Farm Thresholding [Left] 10% [Right] 15%

Figure 2.3.3: Farm Image NMS: [Left] on 10% Thresholding [Right] 15% Thresholding

Figure 2.3.4 shows the effect of NMS on thinning edges

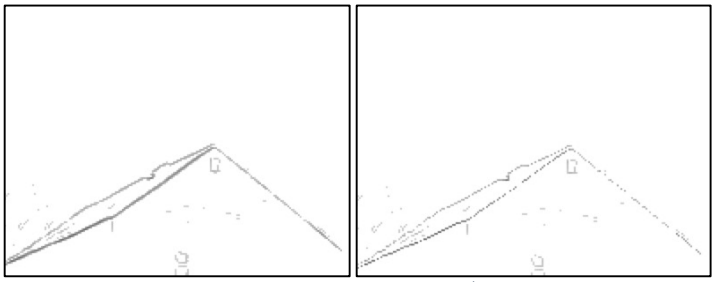

Figure 2.3.4: Effect of NMS on edge thinning. [Left] w/o thinning [Right] w. thinning

Figure 2.3.5 shows the result after Thresholding for 10% and 15 % on Farm image.
Figure 2.3.6 shows the result after apply Non-Maximal Suppression on Farm image.

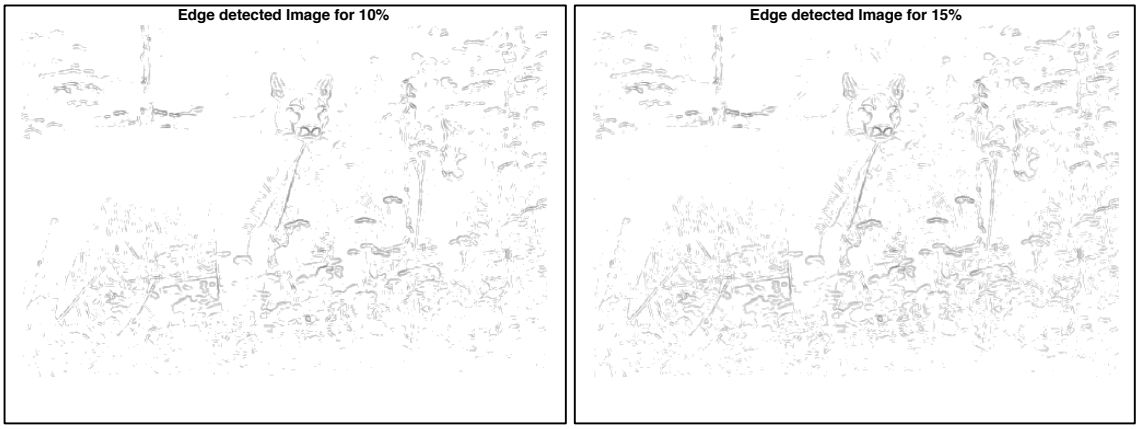

Figure 2.3.5: Cougar Thresholding [Left] 10% [Right] 15%

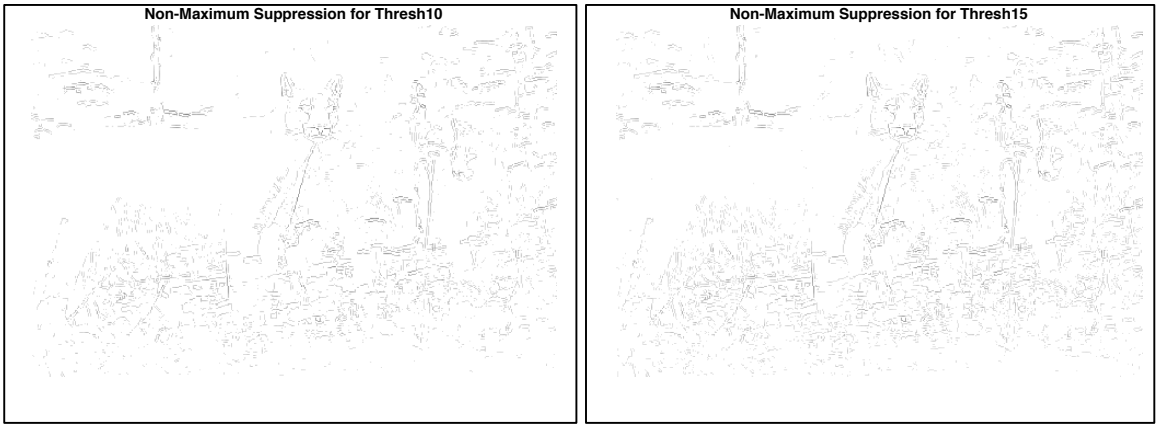

Figure 2.3.6: Cougar Image NMS: [Left] on 10% Thresholding [Right] 15% Thresholding

Figure 2.3.7 shows the effect of different thresholds from Canny Edge Detector

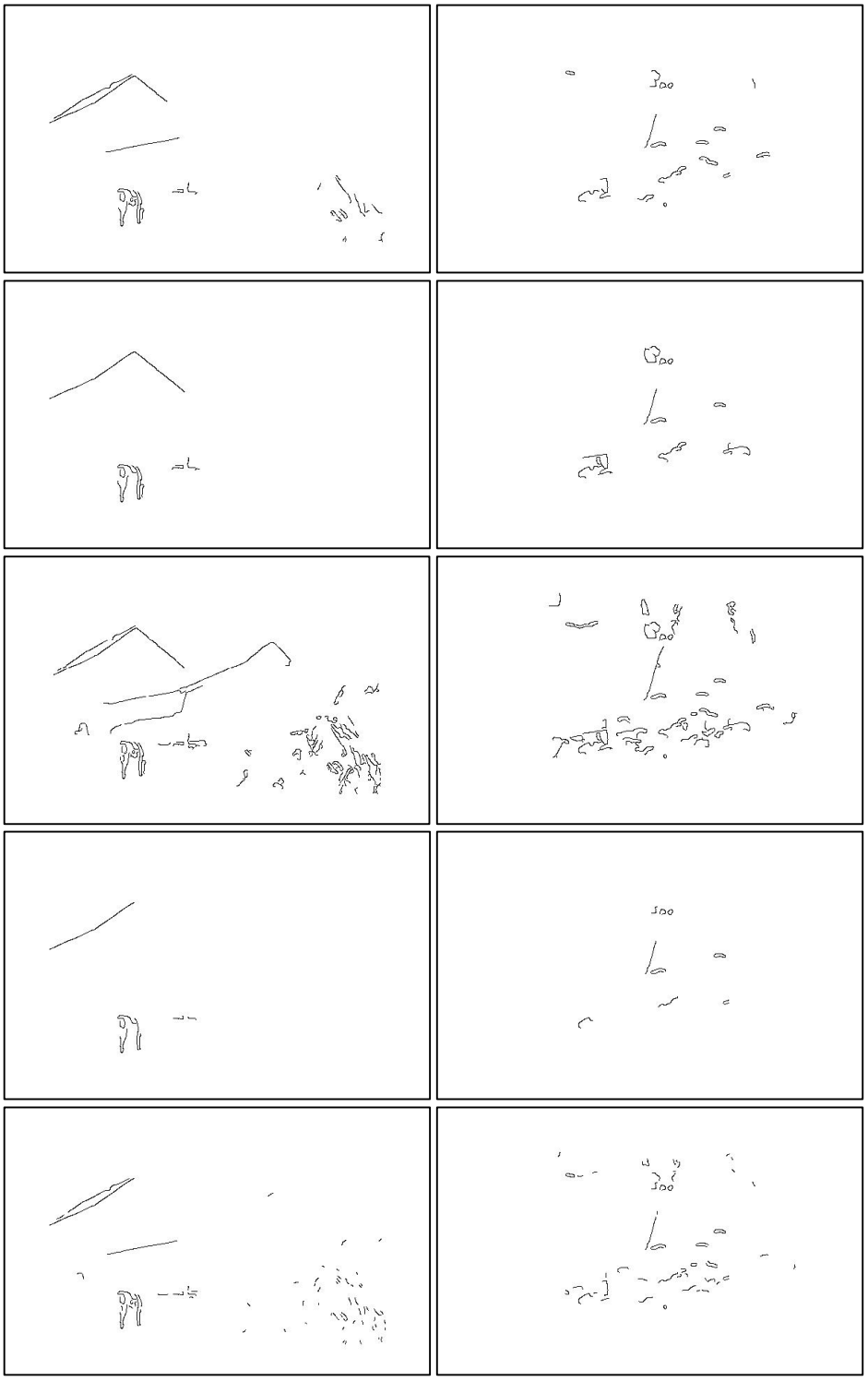

Figure 2.3.7: [Left] Farm [Right] Cougar image for various thresholds [Top to Bottom]: [.3,.6], [.2,.7], [.2,.5], [.4,.7], [.4,.5]

Figure 2.3.8 shows Canny Edge detector result with MATLAB's default choice of threshold

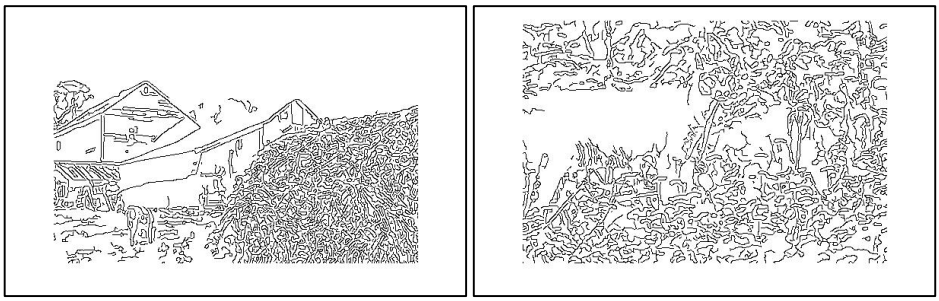

Figure 2.3.8: [Left] Farm [Right] Cougar image for MATLAB's default choice of threshold for Canny edge detector

Figure 2.3.9 shows the result of Structured Edge detector as probability edge map and binary edge map (p>.25) for Farm image.

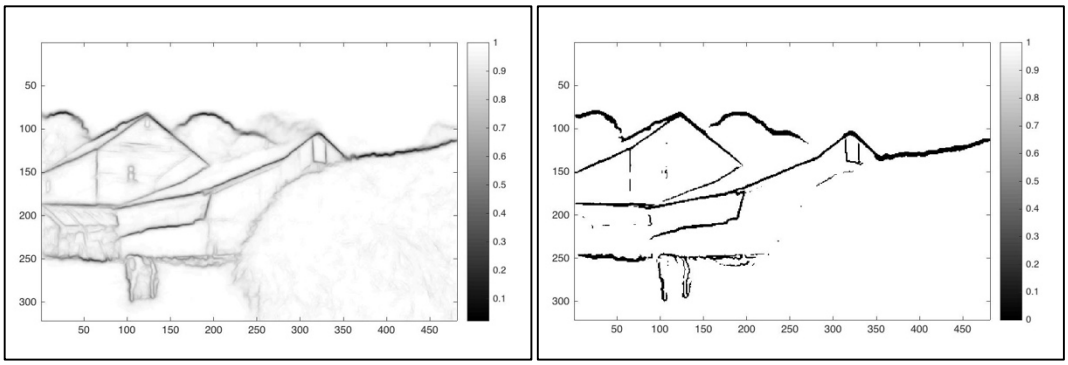

Figure 2.3.9: [Left] Probability edge map of Farm for SE detector [Right] Binary edge map with p>.25

Figure 2.3.10 shows the effect NMS on SE detector for Farm image.

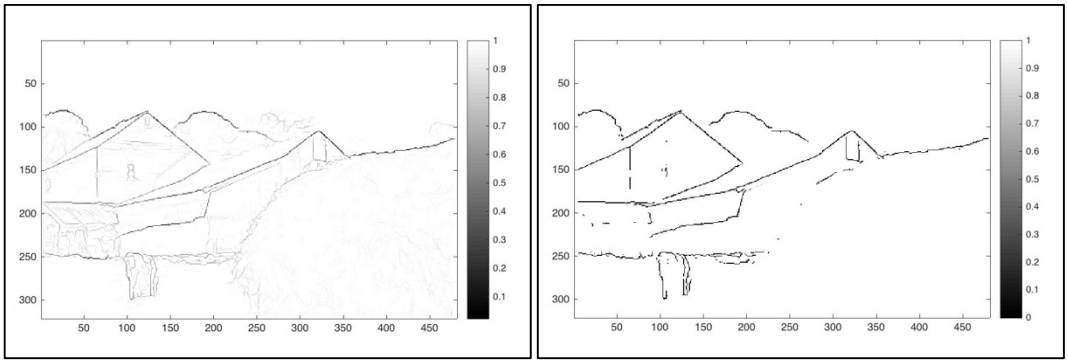

Figure 2.3.10: [Left] Structured Edge detector without NMS [Right] with NMS (Farm)

Figure 2.3.11 shows the result of Structured Edge detector as probability edge map and binary edge map (p>.07) for Cougar image.

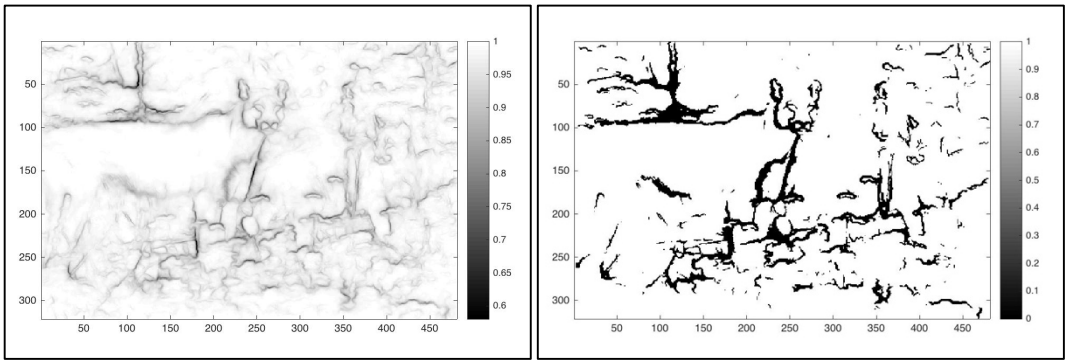

Figure 2.3.11: [Left] Probability edge map of Cougar for SE detector [Right] Binary edge map with p>.07

Figure 2.3.12 shows the effect NMS on SE detector for Cougar image.

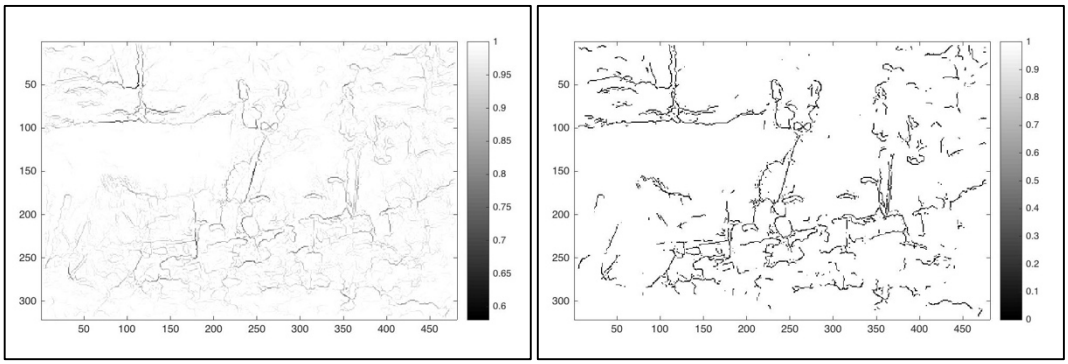

Figure 2.3.12: [Left] Structured Edge detector without NMS [Right] with NMS (Cougar)

Table 2.3.1 shows the F-measure evaluated for different edge images for both Farm and Cougar images.

Table 2.3.1: F-measure table for different edge maps

Detector	Image	Specification	Mean Precision	Mean Recall	F-measure
Sobel Edge Detector	Farm	Sobel Gradient	0.1015	0.5490	0.1714
		10% Threshold	0.1517	0.5557	0.2384
		15% Threshold	0.1472	0.6847	0.2423
		NMS on 10%	0.1512	0.5195	0.2342
		NMS on 15%	0.1554	0.6575	0.2514
Canny Edge Detector		[0.3,.06]	0.6775	0.2307	0.3442
		[0.2,0.7]	0.8832	0.1608	0.2720
		[0.2,0.5]	0.4562	0.3438	0.3921
		[0.4,0.7]	0.8834	0.1078	0.1922
		[0.4,0.5]	0.1716	0.8513	0.2856
SE Detector		w/o NMS	0.9694	0. 5870	0.7312
		w. NMS	0.9670	0.5944	0.7363

Sobel Edge Detector	Cougar	Sobel Gradient	0.3318	0.4318	0.3752
		10% Threshold	0.5508	0.7321	0.6286
		15% Threshold	0.4758	0.8486	0.6097
		NMS on 10%	0.5640	0.7295	0.6361
		NMS on 15%	0.4933	0.8519	0.6248
Canny Edge Detector		[0.3,.06]	0.8469	0.0914	0.1650
		[0.2,0.7]	0.7587	0.0757	0.1377
		[0.2,0.5]	0.8224	0.2418	0.3738
		[0.4,0.7]	0.8288	0.0402	0.0766
		[0.4,0.5]	0.8831	0.1181	0.2084
SE Detector		w/o NMS	0.7471	0.5870	0.6574
		w. NMS	0.7473	0.5871	0.6575

2.4 Discussion

It can be observed that out of all three edge detection algorithms, Structure Edge detection algorithm takes precedence in providing better images visually. Although Canny provides good performance provided that thresholds are chosen in a proper manner, the detection fails if the thresholds are not chosen properly. Hence, SE detection algorithm stays as state of the art edge detection algorithm. SE detection algorithm surpasses every other edge detector by providing better results in *Cougar* image, which has similar level intensity values all over the image. We can observe in Fig 2.4.1 that SE is able to detect the head of the cougar to a great extent whereas it fails in Canny.

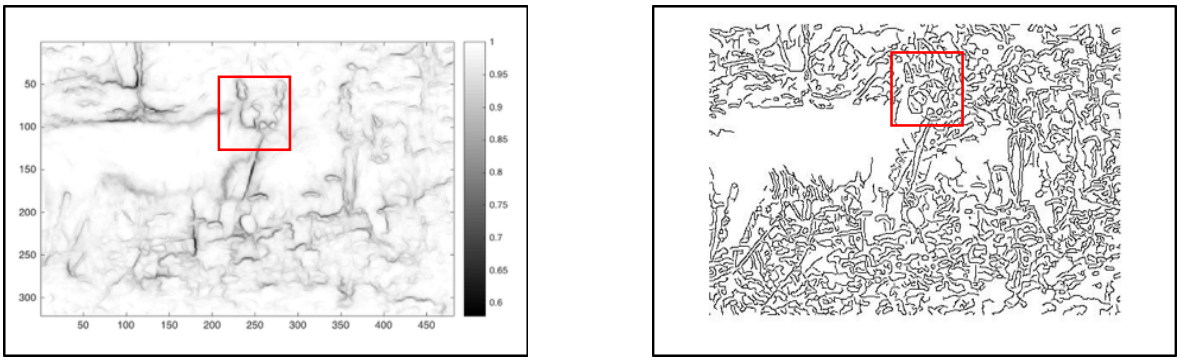

Figure 2.4.1: [Left]: Structured Edge Detector on Cougar, [Right] Canny Edge Detector on Cougar

Sobel Filter does a good job in getting edge map for Farm image; but fails in getting a proper edge map for Cougar image. A limitation of Sobel edge detector is that it can not detect fine details such as hair or grass/straw. Since the cougar image contains a lot of fine details, Sobel fails in providing a better result in this regard. Applying threshold to Sobel gradient image provides a better edge map. However, the boundaries between scene elements are not always sharp. Apart from that, we have to careful while using thresholds. We have to fine tune our threshold every time to get a proper output. It may be at times inefficient in a project where dynamic systems are involved e.g. quad copter. Applying NMS to the threshold images are seen to make the edges thin. However, since they do not consider for a continuous formation of edges, there are places in NMS images where discontinuous trend in certain places. It is mentioned that *Cougar* images failed in any case getting a proper edge map even though every algorithm seems to serve its purpose.

Choosing threshold It is seen from the first two edge detectors that choosing threshold is a trivial task in image processing. Although threshold can be chosen by the user, automation of threshold is better and efficient option. The choice of threshold usually involves analyzing the histogram. There have been many suggestions on automation of threshold choice. An algorithm that can be implemented to find out the threshold is known as Otsu's method. The algorithm executes the following to find out the threshold [10 & 11]:

- We plot the histogram and probability distribution of intensity levels in an image
- We set up initial parameter such as class probability $\omega_i(0)$ and class mean $\mu_i(0)$. We calculate class probabilities and class mean for intensity values starting from 1 till maximum intensity values present in the image. We accordingly calculate a parameter called intra-class variance $\sigma_b^2(t) = \omega_1(t)\omega_2(t)[\mu_1(t) - \mu_2(t)]^2$ for each t values.
- We find the maximum of intra-class variance $\sigma_b^2(t)$ which corresponds to the desired threshold.

The main idea behind the algorithm above is to find a threshold that minimizes the weighted intra-class variance. This can be achieved in MATLAB using **graytrash**( ).

Effect of threshold in Canny Edge Detector We observe from various choice of thresholds in canny images that they were not able to detect enough edge points based on the threshold chosen. There is a obvious relationship between threshold values and edge detection using Canny. The high threshold value affects selection of edge points. If it is set smaller, more pixels are selected as part of edge points which at times end up adding spurious and unnecessary edge points. On the other hand, setting it higher decreases the number of edge fragments. The low threshold value has an effect on the noisy edges to break up, which affects the final display of edges map. If low threshold is chosen close to higher threshold, it will show more localized edges with smooth results whereas choosing it with a bigger difference shows noisier edges. The threshold values that were chosen by MATLAB to get a supposedly better result of Canny Edge detector were [.0438, .1094] and [.0652, .1563]. It seems from the images that they were able to get more edge points detected than the other given threshold even though the outcome is satisfactory enough compared to SE detector.

Evaluation based on F-measure So far we have done our evaluation based on visual appearance of the edge map. F-measure is a good way to evaluate the performance of edge maps based on ground truth developed by human. From the F-measure values as seen in Table 2.3.1, the first observation that we can do is Structure Edge (SE) detector stays state of the art based on metric calculation as well. It can be confirmed from lower F-measure values that the effect of threshold holds in Canny edge detector in providing incomplete or errors in edge maps. It is seen that Sobel edge detection may have failed in providing a better edge map for Cougar image. However, thresholding to 10% with NMS seems to provide a better result which explains that similar intensity values in the image getting leveled out. Overall, the evaluation with F-measure shows that SE detector has provided better performance. An extreme example can be taken from the error map obtained for edge detection using Canny detector with thresholds [.0625, .1563] and Structured Edge with NMS [Figure 2.4.2] can be explained to show how SE detector stays state of the art

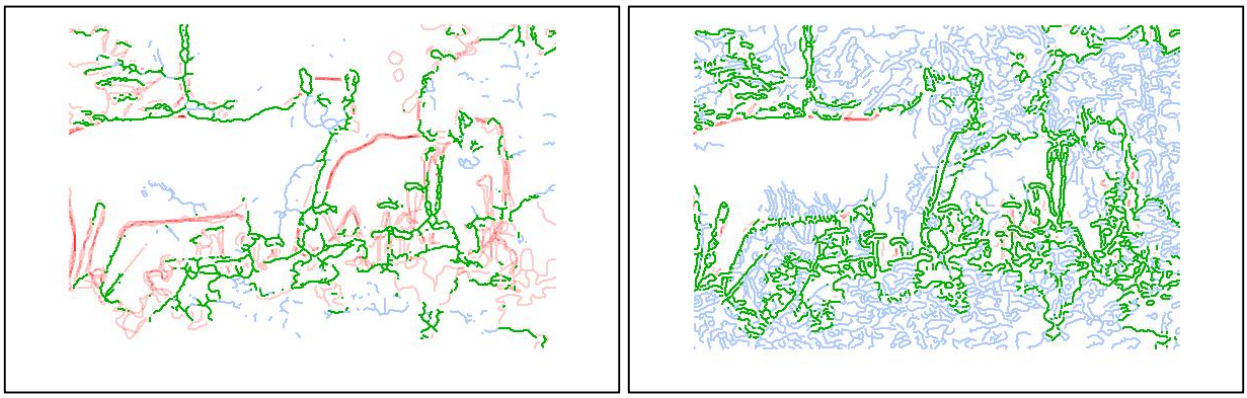

Figure 2.4.2: The error map of [Left] SE edge map, [Right] Canny edge map

Certain color labels are to be considered in the error maps. Green indicates true positive, blue indicates false positive and red indicates false negative. According to F-measure definition, both blue and red are considered as an error and green is considered a correct edge. Now, we observe from the SE error map that many edges are detected with green color labels (which are accurate) including certain edges are red. In case of Canny error map, we find more green labels than SE detector. However, number of blue labels are excessive, which implies that F-measure will go down. In terms of visual sense, it will be difficult to differentiate between edges. The measure F-value of Canny was found to be 0.5449, which is relatively better than other thresholds used for Canny. However, it fails with SE detector to stay as state of the art tool for edge detection.

Image dependence The F-measure depends on the image content as well. Here we are considering two images where the chrominance in one image (Farm.raw) can be distinctively separated whereas the other image (Cougar.raw) has a similar level of chrominance spread in every region (e.g. presence of two cougars, various standing flower petals). With natural eye, it will be easier to detect edges in Farm image than Cougar image. A similar trend is seen in the edge maps where comparing with F-measures we observe that Farm comparatively gets higher measures. The F-measure for SE detector is higher than Cougar images. A similar trend can be found in Canny edge maps.

Rationale behind F-measure The rationale behind F-measure is that it does not only measure the number of edges detected correctly, it also counts for any fake edges. Intuitively, the measure provides a better understanding of visually salient contours in an image. The value for precision and recall is between 0 and 1. Hence the maximum value F-measure can attain is 1. Getting both recall and precision as 1 would imply that there are false negatives and false positives detected such that all detected edges are correct. It is quite impractical to achieve in real world. If either one of the two measures (precision and recall) get significantly higher than the other, the value of F-measure will actually decrease. An empirical example of it can be observed from Table 2.3.1 in Canny edge map (Farm) with [0.2,0.7] and [0.2,.05] that as precision went significantly higher compared to Recall, the F-measure decreased. Similar empirical example can be seen in Canny edge (Cougar) with [0.4,.07].

Let's suppose that the sum of precision (P) and recall (R) is a constant C. It can be shown that, F-measure will reach its maximum when both recall and precise are equal i.e. P=R. We can verify it directly by looking at the mathematical formula of F-measure where the denominator stays constant as C. Since values of P and R are fractions and we need to keep them constant. It means that if we increase one, we have to decrease the other. We know from the basic of fractional multiplication that if we increase one value in turn decreasing the other value, their multiplication will decrease as we continue the process. Hence the highest value is obtained when both of them are i.e. P=R. Let's take an example: P+R =2 and P=R. So, P=R=1. Solving for F, we get F = 1, the maximum value it can reach. The underlying assumption of this example is that we can reach maximum once get all true positives. In a word, F-measure tries to check for edge performance by considering both true edges and fake edges.

Reference

[1] Sebastianraschka.com, 'PCA in 3 Steps', 2015. [Online]. Available: http://sebastianraschka.com/Articles/2015_pca_in_3_steps.html.

[2] Sebastianraschka.com, 'PCA in 3 Steps', 2015. [Online]. Available: http://sebastianraschka.com/Articles/2015_pca_in_3_steps.html.

[3] Csie.ntu.edu.tw, 'LIBSVM -- A Library for Support Vector Machines', 2015. [Online]. Available: https://www.csie.ntu.edu.tw/~cjlin/libsvm/

[4] A. Martinez and A. Kak, 'PCA versus LDA', IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 23, no. 2, pp. 228-233, 2001.

[5] Canny, John. "A computational approach to edge detection." Pattern Analysis and Machine Intelligence, IEEE Transactions on 6 (1986): 679-698.

[6] Dollár, Piotr, and C. Lawrence Zitnick. "Structured forests for fast edge detection." Computer Vision (ICCV), 2013 IEEE International Conference on. IEEE, 2013.

[7] J. Zhu, 'Machine Learning: Decision Trees', University of Wisconsin-Madison, 2015 [Online]. Available: http://pages.cs.wisc.edu/~jerryzhu/cs540/handouts/dt.pdf

[8] N. Freitas, 'Random Forests', University of British Columbia, 2015. Available: http://www.cs.ubc.ca/~nando/540-2013/lectures/l9.pdf

[9] [Online]. Available: http://www.cs.utoronto.ca/~fidler/slides/CSC420/lecture5.pdf

[10] B. Sankur and M. Sezgin, 'Survey over image thresholding techniques and quantitative performance evaluation', *J. Electron. Imaging*, vol. 13, no. 1, p. 146, 2004.

[11] [Online]. Available: http://www.math.tau.ac.il/~turkel/notes/otsu.pdf.