Sergiu NEDEVSCHI
Florin ONIGA
Ion GIOSAN

Robert VARGA
Raluca BREHAR
Andra PETROVAI

Pattern Recognition Systems

Laboratory Works

1st Edition

UTPRESS
Cluj-Napoca, 2023
ISBN 978-606-737-637-1

Editura UTPRESS
Str. Observatorului nr. 34
400775 Cluj-Napoca
Tel.: 0264-401.999
e-mail: utpress@biblio.utcluj.ro
www.utcluj.ro/editura

Director: ing. Dan COLȚEA

Recenzia: Prof.dr.ing. Dorian Gorgan
Prof.dr.ing. Vasile Dădârlat

Pregătire format electronic on-line: Gabriela Groza

ISBN 978-606-737-637-1

Table of contents

Preface

Introduction

The present document is the intended to support the laboratory sessions for the Pattern Recognition Systems course. It presents important methods from the fields of Image Processing, Pattern Recognition and Machine Learning. The main goal is to develop systems capable of perception and automatic scene representation from visual input in the form of images.

Electronic support

To access the additional files required for the practical work use the following link:

https://cv.utcluj.ro/index.php/teaching.html

The website contains the following: the starting project (for multiple versions of Visual Studio); an introduction to the OpenCV library; and the additional data files required for the programming assignments.

Required software

The recommended software to complete the assignments is Visual Studio 2013 or above. Visual Studio solution files along with sample functions are provided for multiple versions. For newer versions an automatic upgrade of the latest one should work. It is also possible to setup a project manually and link the OpenCV library to it. For this, consult the documentation of the library. Any other IDE with C++ support can be used such as Eclipse or CLion. The assignments can be programmed in other languages with OpenCV support, such as Python, although this document assumes C++ is used.

Prerequisites

The following are required to understand and to successfully complete the presented material:

- Linear algebra – matrix operations, linear systems, eigenvalue decomposition;
- Analytic geometry and trigonometry – parametric equations for curves/surfaces;
- Real analysis – multivariate functions, partial derivatives, local and global minima;
- Statistics and probability theory – characterization of random variables;
- Data structures and algorithms – point lists/vectors, sorting arbitrary objects;
- C++ programming – text file input/output, functions and arguments, memory access and dynamic allocation

1 Least Mean Squares

1.1 Objectives

In this assignment a line is fitted to a set of points using the Least Mean Squares method (linear regression). Both the iterative solution (gradient descent) and the closed form are presented. This laboratory work also introduces the OpenCV-based framework used throughout the course.

1.2 Theoretical Background

Consider the following problem: Given a set of data points of the form (x_i, y_i) where $i = \{1,2, \dots, n\}$, find the equation of the line which best fits the data. The solution to this problem is obtained via linear regression. In this setting, the set of points is considered the training set and the goal is to find a line model that best fits the data. We will consider three different model types.

1.2.1 Model 1 – Slope-intercept form

When trying to fit a model to data the first step is to establish the form of the model. Linear regression adopts a model that is linear in terms of the parameters (including a constant term). In this first part, we will adopt a simple model that expresses y in terms of x:

$$f(x) = \theta_0 + \theta_1 x$$

This is the usual way this problem is solved. However, this representation cannot treat vertical lines, since then $\theta_1 \to \infty$. Nonetheless, it provides a good introduction to the method. A vector can be formed that contains all the parameters of the model $\boldsymbol{\theta} = [\theta_0, \theta_1]^{\mathrm{T}}$ (the intercept term θ_0 and the linear coefficient for θ_1).

The Least Squares approach for determining the parameters states that the best fit to the model will be obtained when the following quadratic cost function is at its minimum:

$$J(\boldsymbol{\theta}) = \frac{1}{2}\sum_{i=1}^{n}(f(x_i) - y_i)^2$$

The squared differences can be motivated by the assumption that the error in the data follows a normal distribution – see reference [1]. Note that, this minimizes the error only along the y-axis and not the actual distances of the points from the line. In order to minimize the cost function, we take its partial derivatives with respect to each parameter.

$$\frac{\partial}{\partial \theta_0} J(\boldsymbol{\theta}) = \sum_{i=1}^{n} (f(x_i) - y_i)$$

$$\frac{\partial}{\partial \theta_1} J(\boldsymbol{\theta}) = \sum_{i=1}^{n} (f(x_i) - y_i)\, x_i$$

The cost function attains its minimum when the gradient becomes zero. One general approach to find the minimum is to use **gradient descent**. Since the gradient shows the direction in which the function increases the most, if we take steps in the opposite direction we decrease the value of the function. By controlling the size of the step we can arrive at a local minimum of the function. Since the objective function in this case is quadratic, the function has a single minimum and so gradient descent will find it.

To apply gradient descent start from an initial non-zero guess $\boldsymbol{\theta}$ chosen randomly. Find the gradient in that point:

$$\nabla J(\boldsymbol{\theta}) = \left[\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_0}, \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_1}\right]^T$$

Then apply the following update rule until convergence:

$$\boldsymbol{\theta}_{new} = \boldsymbol{\theta} - \alpha \nabla J(\boldsymbol{\theta}),$$

where α is the learning rate and it is chosen appropriately to ensure the cost function decreases at each iteration. When the change between the parameter values is small enough, the algorithm stops.

The gradient descent approach is appropriate when the roots of the gradient are hard to find. But in this case an explicit solution can be deduced. By setting the gradient components equal to 0 we obtain the following system:

$$\begin{cases} \theta_0 n + \theta_1 \sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i \\ \theta_0 \sum_{i=1}^{n} x_i + \theta_1 \sum_{i=1}^{n} x_i^2 = \sum_{i=1}^{n} x_i y_i \end{cases}$$

which is a linear system with two equations and two unknowns and can be solved directly to obtain the values for $\boldsymbol{\theta}$:

$$\begin{cases} \theta_1 = \dfrac{n \sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{n \sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2} \\ \theta_0 = \dfrac{1}{n}\left(\sum_{i=1}^{n} y_i - \theta_1 \sum_{i=1}^{n} x_i\right) \end{cases}$$

In general, for higher dimensional data, the minimization problem for this model can be written in matrix form:

$$\|A\boldsymbol{\theta} - \boldsymbol{b}\|^2 = (A\boldsymbol{\theta} - \boldsymbol{b})^T (A\boldsymbol{\theta} - \boldsymbol{b})$$

In our case, for two-dimensional data, the matrix A is of size $n\ x\ 2$, with each row i containing the value 1 followed by the value x_i and $\boldsymbol{b}$ is an $n\ x\ 1$ column vector containing the values y_i. In this case the closed form solution is given directly by:

$$\theta_{opt} = (A^T A)^{-1} A^T \boldsymbol{b}$$

For more details and derivation consult [2].

1.2.2 Model 2 – Normal form

In order to address the issue of vertical lines we introduce another model that is capable of dealing with every possible line orientation. Consider the following parameterization of a line in 2D:

$$x\cos(\beta) + y\sin(\beta) = \rho$$

This describes a line with unit normal vector $[\cos(\beta), \sin(\beta)]$ which is at a distance of ρ from the origin. The cost function we wish to minimize in this case is the sum of squared distances of each point from the line. This is given by:

$$J(\beta,\rho)=\frac{1}{2}\sum_{i=1}^{n}(x_i cos(\beta)+y_i sin(\beta)-\rho)^2$$

Note, that this is the actual error term that we want to minimize and that in the previous section we have considered only the error along the y-axis, which is incorrect.

The components of the gradient need to be evaluated to perform gradient descent:

$$\frac{\partial J}{\partial \beta}=\sum_{i=1}^{n}(x_i cos(\beta)+y_i sin(\beta)-\rho)\big(-x_i sin(\beta)+y_i cos(\beta)\big)$$

$$\frac{\partial J}{\partial \rho}=-\sum_{i=1}^{n}(x_i cos(\beta)+y_i sin(\beta)-\rho)$$

A closed form solution can be obtained, although not as easily as in the previous case. The solution is given as:

$$\beta=-\frac{1}{2}atan2\left(2\sum_{i=1}^{n}x_i y_i - \frac{2}{n}\sum_{i=1}^{n}x_i\sum_{i=1}^{n}y_i\,,\sum_{i=1}^{n}(y_i^2-x_i^2)+\frac{1}{n}\left(\sum_{i=1}^{n}x_i\right)^2-\frac{1}{n}\left(\sum_{i=1}^{n}y_i\right)^2\right)$$

$$\rho=\frac{1}{n}\left(\cos(\beta)\sum_{i=1}^{n}x_i+sin(\beta)\sum_{i=1}^{n}y_i\right)$$

1.2.3 Model 3 – Standard form

There is a third possibility for the form of the model. If we adopt a parameterization with 3 free parameters for a line:

$$ax+by+c=0$$

The cost function can be defined as:

$$J(a,b,c) = \frac{1}{2}\sum_{i=1}^{n}(ax_i + by_i + c)^2$$

which can be written in matrix form as the squared norm of a vector:

$$J(a,b,c) = (A\boldsymbol{\theta})^T A\boldsymbol{\theta}$$

where A is a matrix with $n\,x\,3$ elements, each row containing $(x_i, y_i, 1)$ and $\boldsymbol{\theta} = [a,b,c]^T$ is the parameter vector (column vector with 3 elements).

We need to minimize this norm to obtain the parameter values. Working with this model which has 3 parameters has two important consequences. First, all possible lines can be modeled. Second, we will have a family of parameter values which correspond to the same line. To solve the second issue we will seek the parameter vector with unit norm. Finding the null-space of a matrix A with unit norm is a classical problem and it is solved with Singular Value Decomposition. We have:

$$A = USV$$

where U and V are orthogonal matrices and S contains values only on the main diagonal (called singular values). From here the optimal value for the parameter vector will correspond to the last column of the matrix V (which is the eigenvector of A^TA corresponding to the smallest eigenvalue). The interested reader can consult [2] for a demonstration and further details.

1.3 Practical Background

Download the Visual Studio project which is provided. It contains the some sample functions and includes the OpenCV library for image processing. Consider using the following code snippets in your work.

Reading from a text file:

```
FILE* f = fopen("filename.txt","r");
float x,y;
fscanf(f,"%f%f", &x,&y);
fclose(f);
```

Creating a color image – 8 unsigned bits with 3 channels:

```
Mat img(height, width, CV_8UC3);
```

Accessing the pixel at position row *i* and column *j*:

```
Vec3b pixel = img.at<Vec3b>(i,j);
//byte vector with 3 elements
```

Modifying the pixel at row *i* and column *j*:

```
img.at<Vec3b>(i,j)[0] = 255; //blue channel
img.at<Vec3b>(i,j)[1] = 255; //green channel
img.at<Vec3b>(i,j)[2] = 255; //red channel
```

Draw a line between two points:

```
line(img, Point(x1, y1), Point(x2, y2),
Scalar(Blue,Green,Red));
```

Viewing the image:

```
imshow("title", img);
waitKey();
```

1.4 Practical Work

1. Read the input data from the given file. The first line contains the number of points. Each line afterwards contains an (x, y) pair.
2. Plot the points on a white 500x500 background image. For better visibility draw circles, crosses or squares centered at the points. Be careful to consider how the coordinate system in the image is defined. Some points may have negative coordinates. Either do not plot them at all or shift the whole graph. The fitting method itself is not affected by points having negative coordinates.
3. Optionally, use *model 1* and gradient descent to fit a line to the points. Visualize the line at each k-th step. Output and visualize the value of the cost function at each step. Choose the learning rate so that the cost function is decreasing.
4. Use *model 1* and the closed form equation to calculate the parameters θ_0 and θ_1. Visualize both the final line from step 3 and this one and compare the parameter values.
5. Optionally, use *model 2* and gradient descent to fit a line to the points. Visualize the line at each k-th step. Output and visualize the value of the cost function at each step. Choose the learning rate so that the cost function is decreasing.
6. Use *model 2* and the closed form to calculate the parameters β and ρ. Compare the results with those from step 5.
7. Optionally, find the parameters with *model 3* and SVD.

1.5 Example Results

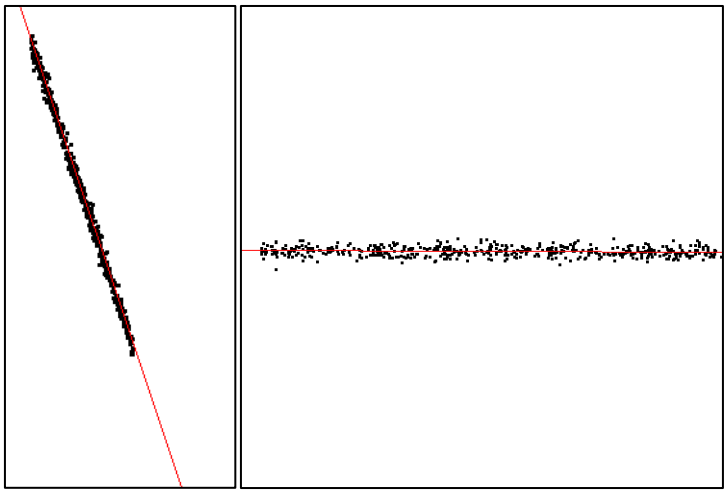


Figure 1.1 – Example Results using model 2 on data from files points1 and points2

1.6 References

[1] Stanford Machine Learning - course notes 1
http://cs229.stanford.edu/notes/cs229-notes1.pdf

[2] Tomas Svoboda - Least-squares solution of Homogeneous Equations
http://cmp.felk.cvut.cz/cmp/courses/XE33PVR/WS20072008/Lectures/Supporting/constrained_lsq.pdf

2 RANSAC

2.1 Objectives

This laboratory work discusses the RANSAC method and applies it to the problem of fitting a line to a set of 2D data points.

2.2 Theoretical Background

Consider the following problem: given *S* a set of 2D data points, find the line which minimizes the sum of distances of the points to the line (orthogonal regression). The data may be contaminated by outliers (noisy or incorrect points), thus fitting a line with the Least Mean Squares on all the points would lead to incorrect results – see Figures 2.1.a-b.

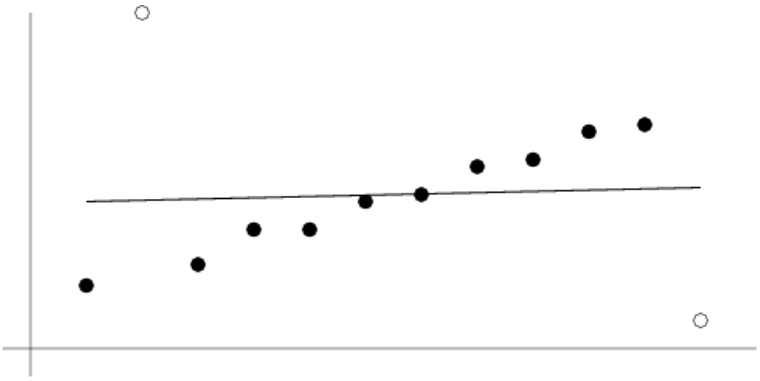


Figure 2.1.a – Line obtained via Least Mean Squares fit on the whole data

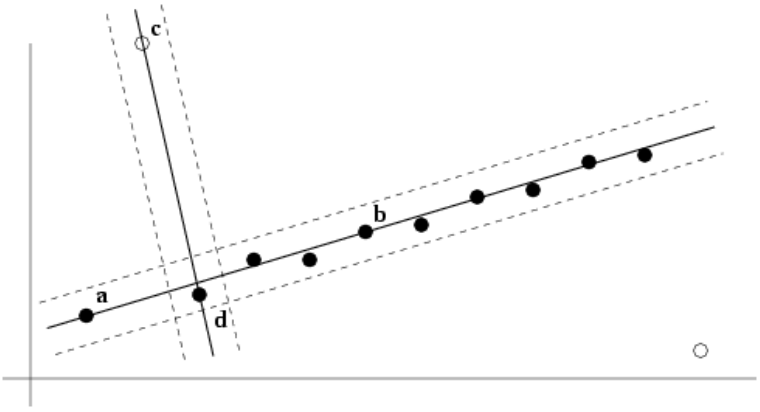


Figure 2.1.b – Two possible lines considered by the RANSAC approach

Random Sample Consensus (RANSAC) is a paradigm for fitting a model to experimental data, introduced by Martin A. Fischler and Robert C. Bolles in [1]. RANSAC addresses the previous issue and automatically determines the set of inlier points and proceeds to fit a model only on this subset. As stated by the authors: "The RANSAC procedure is opposite to that of conventional smoothing techniques: Rather than using as much of the data as possible to obtain an initial solution and then attempting to eliminate the invalid data points, RANSAC uses as small an initial data set as feasible and enlarges this set with consistent data when possible".

The general RANSAC algorithm based on [2] is given below:

Algorithm RANSAC

1. Randomly select a sample containing a number of s data points from S and instantiate the model from this subset.
2. Determine the set of data points S_i which is within a distance threshold t of the model. The set S_i is the consensus set of the sample and defines the inliers for model i.
3. If the size of S_i (the number of inliers) is greater than some threshold T, re-estimate the model using all the points in S_i and terminate.
4. If the size of S_i is less than T, select a new subset and repeat from step 1.
5. After N trials the largest consensus set S_i is selected, and the model is re-estimated using all the points in the subset S_i.

The definition of the parameters appearing in the previous algorithm is given next:

- s – the size of the subset selected for model fitting, i.e. the number of points;
- S – the whole input point set;
- S_i – the subset of the inlier points for the i-th trial, or support set;
- t – threshold value for maximum admissible distance from the model;
- T – threshold value for signaling a sufficiently good data fit;
- N – maximum number of trials;

The general algorithm can be adapted to the problem of line fitting on 2D points. The first step is to select $s = 2$ points randomly, these points define a line. The *support* or *consensus set* S_i for this line consists of the points that lie closer than a distance threshold t to the line. This random selection is repeated a number of times and the line

with biggest support set is retained. The points within the threshold distance are denoted as the inliers (and constitute the eponymous *consensus* set).

The method works one supposition that if one of the points is an outlier then the line will not gain much support. Scoring a line by its support set size has the advantage of favoring better fits. For example, the line passing through points *a* and *b* from Figure 1-b has a support of 10, whereas the line passing through points *c* and *d* has a support of only 2. We can probably deduce from this that *c* or *d* is an outlier point.

Next we address some questions regarding the approach and parameter selection:

- **Why is the method randomized?** Exhaustively trying all subsets is possible only for a small dataset. For example, for a dataset of size $|S| = n$ we have $n(n-1)/2$ point pairs to check for possible lines. This quickly becomes intractable for values of n on the order of 10^5. If the model is fitted using more than 2 points the possible subsets is even larger. By repeatedly selecting two random points we avoid checking all possible subsets.
- **How many trials should we perform?** The value for the number of trials N should be chosen such that there is a high enough probability p that at least one from the N trials is outlier-free. Consider the following:
    - q – is the estimated probability that a point is an inlier
    - q^s – is the probability the all s points are inliers
    - $1 - q^s$ – is the probability that at least one point is an outlier
    - $(1 - q^s)^N$- is the probability that there is at least an outlier in each of the N trials
    - $p = 1 - (1 - q^s)^N$ – is the probability that at least one trial is outlier-free
    - The value of N can be calculated based on a fixed value for the desired p
    - $N = \log(1-p) / \log(1 - q^s)$
- **How to choose the distance threshold t?** We would like to choose the distance threshold t, such that a point is an inlier with a given probability q. For this we require the probability distribution for the distance of an inlier from the model (measurement error model). In practice, the distance threshold is usually chosen empirically. However, if it is assumed that the

measurement error is Gaussian with zero mean and standard deviation σ, then a value for t may be set to 3σ.

- **How large is an acceptable consensus set?** A rule of thumb is to terminate if the size of the consensus set is similar to the number of inliers believed to be in the data set, given the assumed proportion of outliers, i.e. for n data points $T = q \cdot n$

2.2.1 Line model

The equation of a line through two distinct points (x_1, y_1) and (x_2, y_2) is given by:

$$(y_1 - y_2)x + (x_2 - x_1)y + x_1y_2 - x_2y_1 = 0$$

The distance from a point (x_0, y_0) to a line given by the equation $ax + by + c = 0$ is:

$$d = \frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}}$$

2.3 Practical Background

Opening an image with automatic grayscale conversion:

```
Mat img = imread("filename",
CV_LOAD_IMAGE_GRAYSCALE);
```

Creating a grayscale image named `dst`:

```
Mat dst(height, width, CV_8UC1);
//8bit unsigned 1 channel
```

Accessing the pixel at row *i* and column *j*:

```
uchar pixel = img.at<uchar>(i,j);
//unsigned char type
```

A black point from the image at row `i` and column `j` corresponds to a point at coordinates `(x=j, y=i)` – this is the OpenCV library's convention for point coordinates

```
if (img.at<uchar>(i,j)==0){
     Point p; p.x = j; p.y = i;
}
```

Modifying the pixel at row *i* and column *j*:

```
img.at<uchar>(i,j) = 255; //white
```

Selecting a random point from an array of *n* points (requires stdlib.h):

```
Point p = points[rand()%n];
```

Draw a line between two points:

```
line(img, Point(x1, y1), Point(x2, y2),
Scalar(B,G,R));
```

Viewing the image:

```
imshow("title", img);
waitKey();
```

2.4 Practical Work

1. Open the input image and construct the input point set by finding the positions of all black points.
2. Calculate the parameters N and T starting from the recommended values:
3. $t = 10$, $p = 0.99$, $q = 0.7$ and $s = 2$. For points1.bmp use $q = 0.3$.
4. Apply the RANSAC method:
   a. Choose two different points;
   b. Determine the equation of the line passing through the selected points;
   c. Find the distances of each point to the line;
   d. Count the number of inliers;
   e. Save the line parameters (a, b, c) if the current line has the highest number of inliers so far;
5. Write the correct termination conditions based on the size of the consensus set and the maximum number of iterations.
6. Optionally, estimate the line parameters using Least Mean Squares on the best consensus set.
7. Draw the optimal line found by the method.

2.5 References

[1] Fischler, Martin A., and Robert C. Bolles. "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography." Communications of the ACM 24, no. 6 (1981): 381-395.

[2] Hartley, Richard, and Andrew Zisserman. Multiple view geometry in computer vision. Cambridge university press, 2003.

3 Hough Transform

3.1 Objectives

The main objective of this laboratory session is to implement the Hough Transform for line detection from edge images.

3.2 Theoretical Background

The classical Hough transform is a method that solves an important image processing problem: finding lines in an image that contains a set of interest points. The straightforward method of computing lines from each pair of points has an increased computational complexity of $O(n^2)$, and is unusable for a large number of points.

The Hough transform was first proposed and patented by Peter Hough in [1]. It proposes to count how many points are placed on each possible line in the image. The original method relies on the representation of the lines in the slope-intercept form ($y = ax + b$), and on the building of the line parameter space, also called Hough accumulator. For each image interest point, all possible lines are considered, and the corresponding elements in the line parameter space are incremented. Relevant image lines are found at the locations of the local maxima in the line parameter space.

The initial proposal was focused on the detection of lines from video sequences, based on a slope and free-term line representation. This representation is not optimal because it is not bounded: in order to represent all the possible lines in an image, the slope and the intercept terms should vary between -∞ and +∞. The work of Duda and Hart from [2] made the Hough transform more popular in the computer vision field. The main problem of the original Hough transform (unbounded parameters) is solved by using the so-called normal parameterization. The normal parameterization of a line consists of representing the line by its normal vector and the distance from origin to the line. The normal representation (1) is sometimes referred to as the $\rho - \theta$ representation (Figure 3.1).

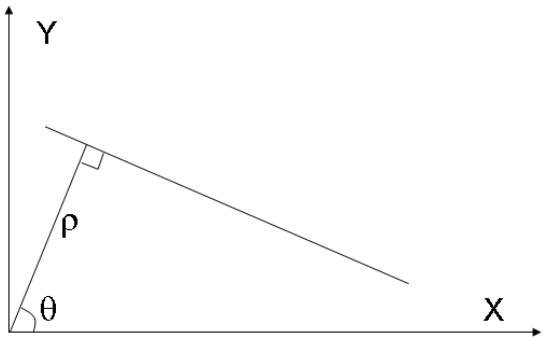


Figure 3.1 – Line represented by its normal vector, at angle $\boldsymbol{\theta}$, and its distance $\boldsymbol{\rho}$ from the origin

The equation satisfied by a point on the line *(x,y)* is then given by:

$$\rho = x\,cos(\theta) + y\,sin(\theta)$$

Parameter quantization plays an important role in decreasing the computational complexity of the method. Quantization determines the size of the Hough accumulator. For each of the two line parameters, a quantization level must be established, depending on the desired accuracy. The accuracy of ρ can be of 10, 1 or 0.5 pixels etc, and the accuracy of θ can be of 10, 1 or 0.5 degrees etc. The parameters ρ and θ have a limited range because the image has a limited size. The maximum value for ρ is the diagonal of the image. Depending on the interval selected for θ, there are mainly two equivalent configurations for the line parameter range. The first one is proposed in the original work and we will employ the second one:

$$\theta \in [0°, 180°), \rho \in [-\rho_{max}, \rho_{max}]$$

$$\theta \in [0{,}360°), \rho \in [0, \rho_{max}]$$

Let us assume that the Hough accumulator H represents the quantized line parameter space. The quantization steps for ρ and θ are $\Delta\rho$ and $\Delta\theta$, respectively. Their maximum values are ρ_{max} and θ_{max}. The accumulator will have a size of $(\rho_{max}\ /\Delta\rho, \theta_{max}/\Delta\theta)$. H is built with the following simple steps:

Algorithm Hough Transform

Initialize H with 0
For each edge point $P(x, y)$
For each θ from 0 to θ_{max} (with a step of $\Delta\theta$)

Compute $\rho = x\ cos(\theta) + y\ sin(\theta)$
If $\rho \in [0, \rho_{max}]$ increment $H(\rho, \theta)$

The increment operation of a Hough location can also be weighted, if different weights are desired for each interest point. Once the accumulator is built, relevant lines are extracted as the local peaks of the accumulator. Local peaks are found at places where the accumulator is larger than all neighboring cells.

An example of line detection based on the Hough transform is presented below (Figure 3.2). The parameter ranges for this example are [0, 360) degrees for θ, [0, 144] pixels for ρ. The parameter accuracy is 1 degree for θ and 1 pixel for ρ.

Choosing the correct level of quantization is important. If the quantization is too fine, then the resolution increases, but so does the processing time. A high resolution and also raises the chances of collinear points falling into different accumulator bins (this might cause multiple detections and the fragmentation of certain lines).

Although the Hough transform is widely used for line detection, it can also work with more complex curves, as long as an adequate parameterization is available. Duda and Hart [2] also proposed the detection of circles based on the Hough transform. In this case a 3D parameter space is needed and each interest point is transformed into a right circular cone in the parameter space (all the possible circles containing the point). Later, Ballard generalized the Hough transform to detect arbitrary non-analytical shapes in [3].

3.3 Practical Background

Use the simplest configuration for parameter quantization: 1 pixel for ρ and 1 degree for θ. Use the second option for the parameter range from formula (2). The size of the Hough accumulator will have *D + 1* rows and *360* columns, where *D* is the image diagonal rounded to the nearest integer.

```
Mat Hough(D+1, 360, CV_32SC1); //matrix with
int values
```

Initialize the accumulator to zero using:

```
Hough.setTo(0);
```

Modify the accumulator like:

```
Hough.at<int>(ro, theta)++;
```

The accumulator needs to be normalized to have values in the range 0-255 to be viewed as a grayscale image. Use the `normalize` method or find the maximum from the accumulator and call the following function. Note that the subsequent operations must be performed on the original accumulator.

```
Mat houghImg;
Hough.convertTo(houghImg, CV_8UC1,
255.f/maxHough);
```

In order to locate peaks in the accumulator, you will test for each Hough element if it is a local maximum in a squared window (*n x n*) centered on that element. Use the following custom structure to store and sort the local maxima. The < operator of the structure has been overwritten to use the > operator between the peak values specifically to sort descending when the `sort` method from the `algorithm` library is called.

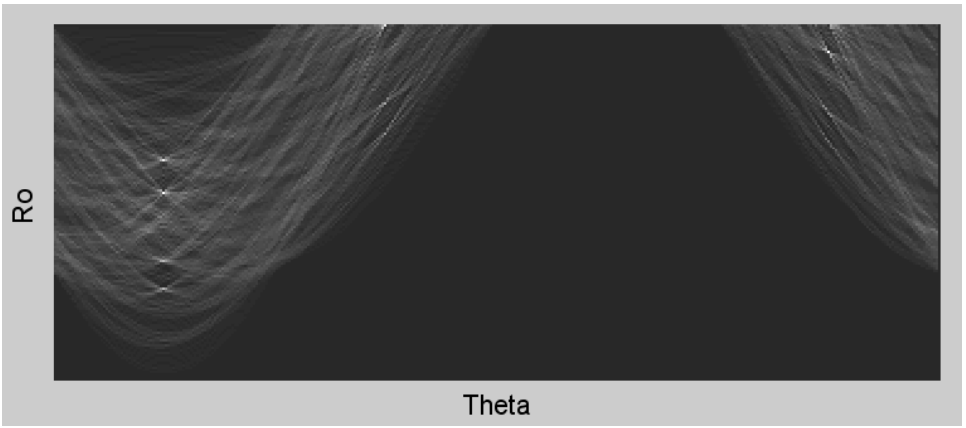
```
struct peak{
     int theta, ro, hval;
     bool operator < (const peak& o) const {
          return hval > o.hval;
     }
};
```

3.4 *Practical Work*

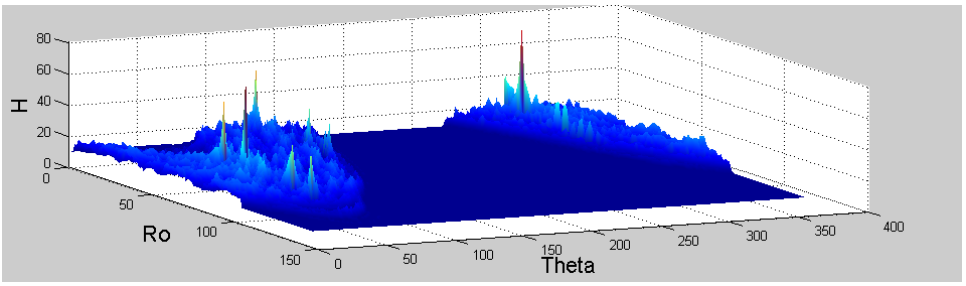
1. Compute the Hough accumulator based on the edge image and display it as a grayscale image.
2. Locate the *k* largest local maxima from the accumulator. Try using different window sizes such as: 3x3, 7x7 or 11x11.
3. Draw lines that correspond to the peaks found. Use both the original image and the edge image for visualization.

3.5 *Example Results*

a. b. c.



d.



e.

Figure 3.2 - **a.** An image containing a pattern with straight borders corrupted by salt-and-pepper like noise, **b.** The edges detected with the Canny edge detector, **c.** The most relevant image lines are displayed with green, associated to the most relevant 8 peaks from the Hough accumulator, **d.** The Hough accumulator displayed using an intensity encoding, **e.** The Hough accumulator displayed in 3D, using color encoding.

3.6 References

[1] P. Hough, “Method and means for recognizing complex patterns”, US patent 3,069,654, 1962.

[2] R. O. Duda and P. E. Hart, "Use of the Hough Transformation to Detect Lines and Curves in Pictures," Comm. ACM, Vol. 15, pp. 11–15, 1972.

[3] D. H. Ballard, "Generalizing the Hough Transform to Detect Arbitrary Shapes", Pattern Recognition, Vol.13, No.2, p.111-122, 1981.

4 Distance Transform

4.1 Objectives

In this laboratory session we will study an algorithm that calculates the Distance Transform of a binary image (object and background). Our goal is to evaluate the pattern matching score between a known template object (e.g. a pedestrian contour) and an unknown object (e.g. the contour of a different object) in order to decide if the unknown object is similar or not to the template object. The less the pattern matching score is, the more similar is the unknown object is to the template.

4.2 Theoretical Background

4.2.1 The Distance Transform

A distance transform, also known as distance map or distance field, is a representation of a digital image. The term transform or map is used depending on whether the initial image is transformed, or it is simply endowed with an additional map or field. The map will contain at each pixel the distance to the nearest obstacle pixel. The most common type of obstacle pixel is a boundary pixel from a binary image.

The distance transform is an operator normally only applied to binary images. The result of the transform is a grayscale image that looks similar to the input image, except that the intensities at each point show the distance to the closest boundary point.

One way to think about the distance transform is to first imagine that foreground regions in the input binary image are made of some uniform slow burning flammable material. Then consider simultaneously starting a fire at all points on the boundary of a foreground region and letting the fire burn its way into the interior. If we then label each point in the interior with the amount of time that the fire took to first reach that point, then we have effectively computed the distance transform of that region.

See the next image for an example of a chessboard distance transform on a binary image containing a simple rectangular shape. In the left image the pixels with value “0” represent object pixels (boundary pixels) and those with value “1” background pixels. In the right image is the result of applying the Distance Transform using the chessboard

metric, where each value encodes the minimum distance to an object pixel (boundary pixel):

0	0	0	0	0	0	0
0	1	1	1	1	1	0
0	1	1	1	1	1	0
0	1	1	1	1	1	0
0	1	1	1	1	1	0
0	1	1	1	1	1	0
0	0	0	0	0	0	0

0	0	0	0	0	0	0
0	1	1	1	1	1	0
0	1	2	2	2	1	0
0	1	2	3	2	1	0
0	1	2	2	2	1	0
0	1	1	1	1	1	0
0	0	0	0	0	0	0

Figure 4.1 – On the left: binary input image; on the right: distance transform of the image, every position shows the checkerboard distance to the closest boundary point (0 values in the input image) [1]

Usually the transform is qualified with the chosen metric. For example, one may speak of Manhattan Distance Transform, if the underlying metric is Manhattan distance. Common metrics are:

- Euclidean distance;
- Taxicab geometry, also known as *City block distance* or *Manhattan distance*;
- Chessboard distance.

There are several algorithms for implementing DT:

- Chamfer based DT;
- Euclidian DT;
- Voronoi diagram based DT.

We will present the **Chamfer based DT** which is a simple and very fast method (it requires only two scans of the binary image) and it is an approximation of the Euclidian DT. The sketch of the algorithm is the following:

Algorithm Chamfer Distance Transform

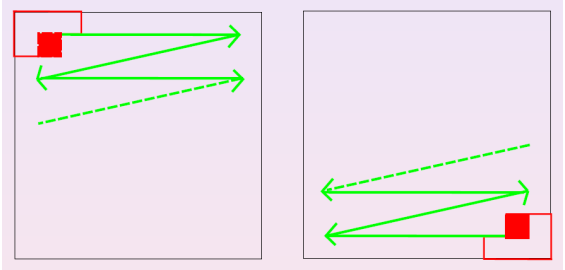
- A 3x3 weight mask is chosen which is has values proportional to the Euclidean distances from the middle element. The simplest and smallest such values are 2 for lateral displacement and 3 for diagonal displacement. In this way the distances obtained will be equal to approximately twice the actual Euclidean distances.

$$weights = \begin{bmatrix} 3 & 2 & 3 \\ 2 & 0 & 2 \\ 3 & 2 & 3 \end{bmatrix}$$

- The distance transform map has the same size as the input image and is initialized with zeroes and large values based on the positions of edge points:

$$DT(i,j) = \begin{cases} 0 & img(i,j) \in Object \\ \infty & img(i,j) \notin Object \end{cases}$$

- A double scan (first top-down, left-right and second bottom-up, right-left) of the image (with the corresponding two parts of the mask – see the figure below) is required to update the minimum distance. On the first traversal the central element is compared to the neighbors corresponding yellow elements, and on the second traversal, with green elements:



- The next update operation should be performed on the DT image while scanning (forward and then backward) the source image. Only the a subset of neighbors are used and so the minimum is calculated based on the previous values with regards to traversal direction:

$$DT(i,j) = \min_{(k,l)\in Mask} DT(i+k-1, j+l-1) + weights(k,l)$$

It is worth noting, that the center element with weight 0 belongs to both masks, and so, the minimum is always compared to the existing value from the DT map.

If we apply DT on a binary image, where the value 0 signifies object pixels and the value 255 encodes the background, and we want to obtain a grayscale DT image (8 bits/pixel), the value ∞ from the algorithm should be substituted with the value 255.

4.2.2 Pattern Matching using DT

Our goal in this part is to compute the pattern matching score between a template, which represents a known object, and an unknown object. The score can be used to quantify the similarity between the contours of the template and that of unknown object. We consider that both the template and the unknown object images have the same dimensions.

The steps for computing the pattern matching score:
- compute the DT image of the template object;
- superimpose the unknown object image over the DT image of the template;
- the pattern matching score is the average of all the values from the DT image that lie under the unknown object contour.

Example: Consider that the template object is a leaf contour and the unknown object is a pedestrian contour:
- Compute the DT image of the leaf:
- Superimpose the pedestrian image over the DT image of the leaf;
- Evaluate the matching score as the average values from the DT image from the positions indicated by the pedestrian contour

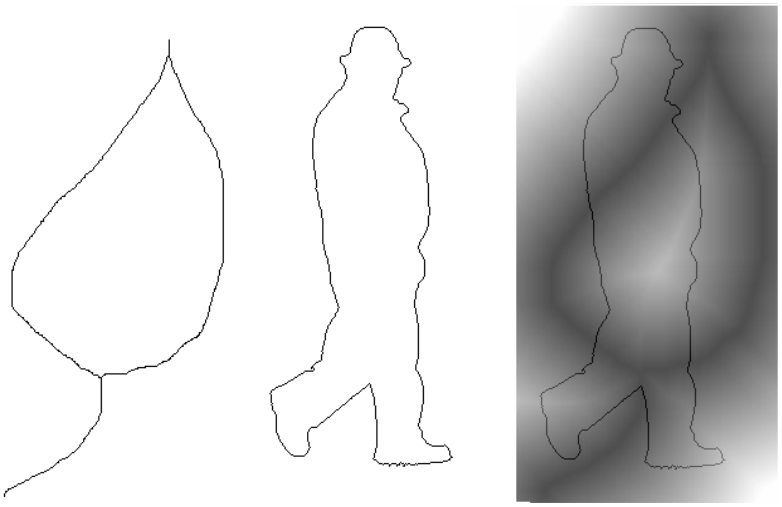


Figure 4.2 – From left to right: contour of a leaf; contour of a pedestrian; the pedestrian superimposed on the distance transform of the leaf

4.3 Practical Background

Opening the source image as a grayscale image:

```
Mat img = imread("filename",
IMREAD_GRAYSCALE);
```

Initializing the DT image as a copy of the source image:

```
Mat dt = src.clone();
```

Convenient 8-neighborhood access at position *(i,j)*:

```
int di[9] = {-1,-1,-1,0,0,0,1,1,1};
int dj[9] = {-1,0,1,-1,0,1,-1,0,1};
int weight[9] = {3,2,3,2,0,2,3,2,3};
for(int k=0; k<9; k++)
     uchar pixel = img.at<uchar>(i+di[k],
                                 j+dj[k]);
```

4.4 Practical Work

1. Implement the Chamfer Distance Transform algorithm. Compute and visualize the DT images for: *contour1.bmp, contour2.bmp, contour3.bmp*. Results should coincide with the ones presented in the text. Object pixels are black and background pixels are white.
2. Compute the DT image for template.bmp. Evaluate the matching score between the template and each of the two unknown objects (unknown_object1.bmp – pedestrian contour, unknown_object2.bmp – leaf contour). The matching score is given as the average of the DT image values that lie under the contour points from the object image.
3. Compute the matching score by switching the roles of template and unknown object. Are the scores the same?
4. Before calculating the matching score, translate the unknown object such that its center of mass coincides with template's center of mass. Estimate the center of mass based on the contour points only.
5. Optionally, implement the true Euclidean Distance Transform. Why is the Chamfer Distance Transform different?

4.5 Example Results

Examples of DT image results using Chamfer method using the suggested weight matrix.

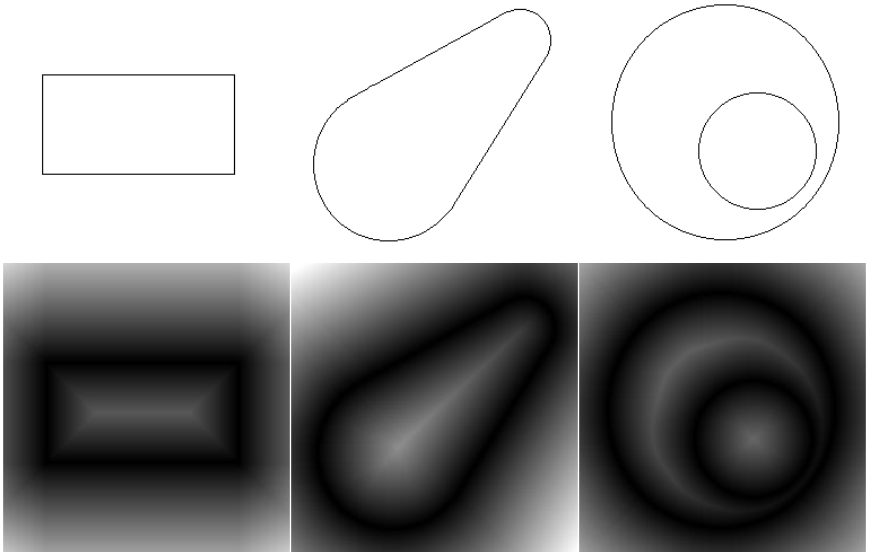


Figure 4.3 – Upper row: input binary images; lower row: corresponding DT images

4.6 References

[1] Wikipedia The Free Encyclopedia – *Distance Transform*, http://en.wikipedia.org/wiki/Distance_transform

[2] Compendium of Computer Vision – *Distance Transform*, http://homepages.inf.ed.ac.uk/rbf/HIPR2/distance.htm

5 Statistical Data Analysis

5.1 Objectives

The purpose of this laboratory session is to explore methods for analyzing statistical data. We will study the mean, standard deviation, covariance and the Gaussian probability density function. The experiments will be performed on a set of images containing faces. The covariance matrix will be used to establish the correlations between different pixels.

5.2 Theoretical Background

5.2.1 Definitions

A *random variable* X is a function that assigns a real number $X(\zeta)$ to each outcome ζ in the sample space of a random experiment (see figure below). This function $X(\zeta)$ is performing a mapping from all the possible elements in the sample space onto the real line (real numbers). Random variables can be:

- Discrete: the resulting number after rolling a dice;
- Continuous: the weight of a person.

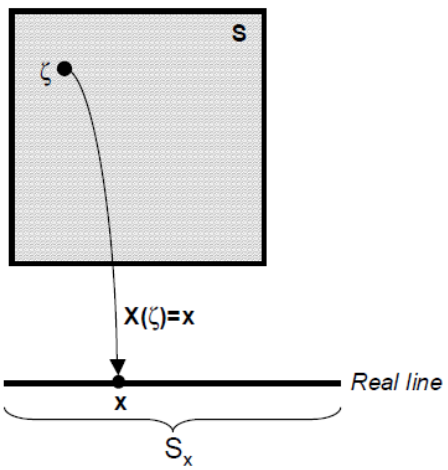


Figure 5.1 Random variable example

A *random variable vector* X is a function that assigns a vector of real numbers to each outcome $X(\zeta)$ in the sample space S. The notion of a random vector is an extension to that of a random variable:

$$\boldsymbol{X} = [X_1, X_2, \dots, X_N]^T$$

5.2.2 Statistical Characterization of Random variables

A random variable with probability density function $f_x(x)$ can be partially characterized by:

1. Expectation: represents the center of mass.

$$E[X] = \mu = \int_{-\infty}^{\infty} x f_x(x) dx$$

2. Variance: represents the spread about the mean.

$$VAR[X] = E[(X - E[X])^2] = \int_{-\infty}^{\infty} (x - \mu)^2 f_x(x) dx$$

3. Standard deviation: The square root of the variance. It has the same units as the random variable.

$$STD[X] = VAR[X]^{1/2}$$

5.2.3 Statistical Characterization of Random Vectors

We can describe a random vector with the following measures:

1. Mean vector:

$$E[\boldsymbol{X}] = \left[E[X_1], E[X_2], \dots, E[X_N]\right] = [\mu_1, \mu_2, \dots, \mu_N] = \boldsymbol{\mu}$$

2. Covariance matrix:

$$COV[\boldsymbol{X}] = \boldsymbol{\Sigma} = E[(\boldsymbol{X} - \boldsymbol{\mu})(\boldsymbol{X} - \boldsymbol{\mu})^T]$$

$$= \begin{bmatrix} E[(X_1 - \mu_1)(X_1 - \mu_1)^T] & \dots & E[(X_1 - \mu_1)(X_N - \mu_N)^T] \\ \dots & \dots & \dots \\ E[(X_N - \mu_N)(X_1 - \mu_1)^T] & \dots & E[(X_N - \mu_N)(X_N - \mu_N)^T] \end{bmatrix}$$

$$= \begin{bmatrix} \sigma_1^2 & \dots & c_{1N} \\ \dots & \dots & \dots \\ c_{N1} & \dots & \sigma_N^2 \end{bmatrix}$$

The covariance matrix indicates the tendency of each pair of random variables (dimensions in a random vector) to vary together, i.e., to co-vary. The covariance has several important properties:

- If X_i and X_k tend to increase together, then $c_{ik} > 0$
- If X_i tends to decrease when X_k increases, then $c_{ik} < 0$
- If X_i and X_k are **uncorrelated**, then $c_{ik} = 0$
- $|c_{ij}| < \sigma_i \sigma_j$, where σ_i is the standard deviation of X_i
- $c_{ii} = VAR[X_i]$
- $c_{ij} = c_{ji}$

The elements of the covariance matrix can be expressed as:

$$c_{ik} = E[(X_i - \mu_i)(X_k - \mu_k)]$$

$$c_{ii} = \sigma_i^2$$
$$c_{ik} = \rho_{ik}\sigma_i\sigma_k$$

where ρ_{ik} is called the **correlation coefficient.**

The next figures represent the correlation charts between two features, X_i and X_k.

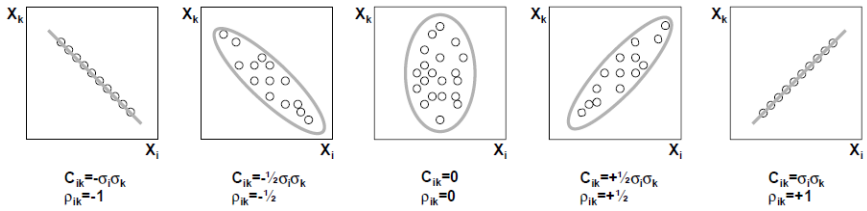


Figure 5.2 – Examples of 2D samples with different correlation coefficients

5.3 Practical Background

You are given *p=400* images that contain human faces. The figure below shows a montage of all the input images:

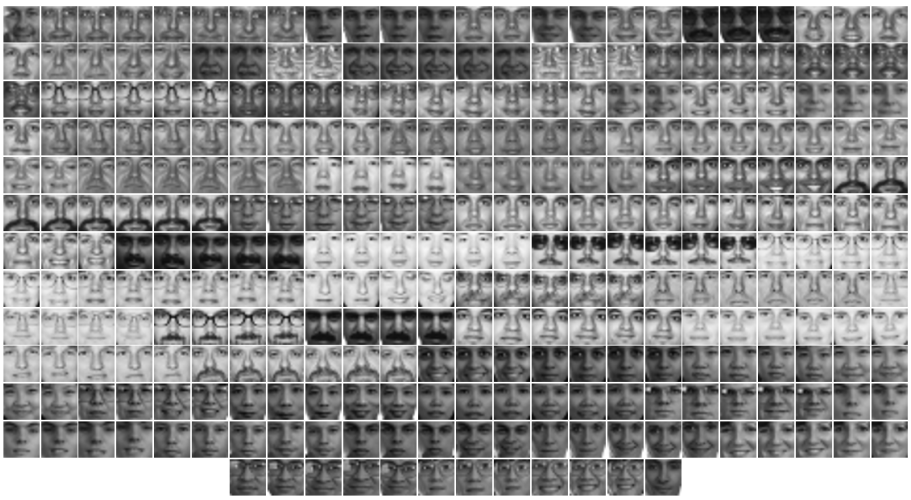


Figure 5.3 – Input image dataset containing portraits

Let ***I*** be the feature matrix which will hold all the intensity values from the image set. ***I*** is of dimension $p \times N$, where p is the number of images and N is the number of pixels in each image. The k^{th} row contains all the pixel values from the k^{th} image in row-major order. The row-major order for a 3x3 matrix is:

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix} \rightarrow [A_{00}, A_{01}, A_{02}, A_{10}, A_{11}, A_{12}, A_{20}, A_{21}, A_{22}]$$

Each image in the set has the dimension of *19x19* pixels. The interpretation of the feature matrix ***I*** is that each row holds a sample for the *N* dimensional random variable ***X*** which is drawn from the distribution underlying the dataset.

Your task will be to compute the covariance matrix of the given set of images and to study how different features vary with respect to each other.

The mean value of a feature located at position *i* in the image is:

$$\mu_i = \frac{1}{p}\sum_{k=1}^{p} I_{ki}$$

Where I_{ki} represents the value of feature *i* in image *k*. The standard deviation of a feature *i* is:

$$\sigma_i = \sqrt{\frac{1}{p}\sum\nolimits_{k=1}^{p}(I_{ki} - \mu_i)^2}$$

The elements of the covariance matrix, c_{ij} can be computed by:

$$c_{ij} = \frac{1}{p}\sum_{k=1}^{p}(I_{ki} - \mu_i)(I_{kj} - \mu_j)$$

The correlation coefficient is:

$$\rho_{ij} = \frac{c_{ij}}{\sigma_i \sigma_j}$$

Note that $c_{ii} = \sigma_i^2$ and $\rho_{ii} = 1$.

5.4 Practical Work

1. Load the 400 images and store the intensity values as rows in the feature matrix ***I***. The code that loads several images from a folder is:

```
char folder[256] = "faces";
char fname[256];
```

```
for(int i=1; i<=400; i++){
   sprintf(fname,"%s/face%05d.bmp", folder, i);
   Mat img = imread(fname,0);
}
```

2. Compute mean values for each feature and save them to a csv text file (comma separated values). Write the components in a text file separated by commas and save it with a csv extension. Csv files are viewable in Microsoft Excel as tables.
3. Compute the covariance matrix and save it to a csv text file.
4. Compute the correlation coefficients matrix and save it to a csv text file.
5. Compute the correlation coefficient and display the correlation chart between selected intensity feature pairs. The correlation chart between the ith and jth features is a 256x256 white image with black points at locations (I_{kj}, I_{ki}), for each possible k. Use the following coordinate pairs (row, column) which must be linearized (transformed to a single value using the row-major order presented above) to find the correct column index from $\boldsymbol{I}$:
   a. (5,4) and (5,14). These points correspond to pixels belonging to left eye and right eye. Your result should resemble the one in figure below having the correlation coefficient ~ 0.94.
   b. (10,3) and (9, 15). These points correspond to pixels belonging to left cheek and right cheek. Your result should resemble the one in figure below having the correlation coefficient ~ 0.84.
   c. (5,4) and (18,0). These points correspond to pixels belonging to left eye and the left bottom corner of the face images (notice these points are not highly correlated). Your result should resemble the one in figure below having the correlation coefficient ~ 0.07.
6. Plot the probability density function for a selected feature having the form of a one dimensional Gaussian probability density function:

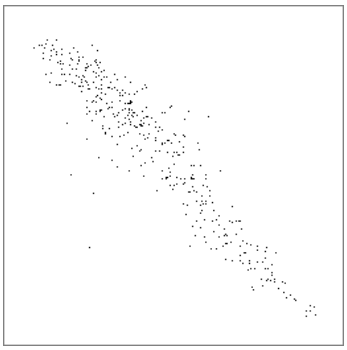
$$f_x(x) = \frac{1}{\sqrt{2\pi}\sigma} exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

   where μ is the mean and σ is the standard deviation for the selected feature. Normalize the density values so that the peak reaches the height of the image.
7. Optionally, plot the 2D probability density function as a grayscale image for two selected features using the 2D Gaussian probability density function:

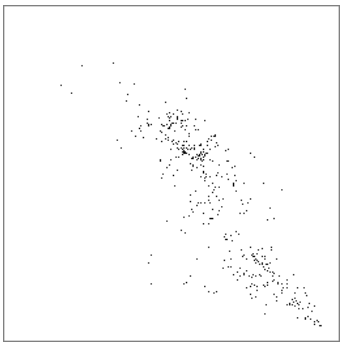
$$p(x_i, x_j) = \frac{1}{2\pi\sqrt{\det(C_{ij})}} exp\left(-0.5\left([x_i - \mu_i, x_j - \mu_j]C_{ij}^{-1}\begin{bmatrix} x_i - \mu_i \\ x_j - \mu_j \end{bmatrix}\right)\right)$$

where μ_i is the mean for feature *i* and C_{ij} is the covariance matrix between features *i* and *j*. Normalize the density values to fit inside the range 0:255.

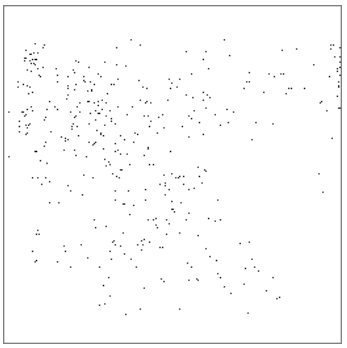
5.5 Example Results



a



b



c

Figure 5.3 – From left to right, Example Results for tasks 5 a, b and c

5.6 References

[1] MIT CBCL FACE dataset
http://www.ai.mit.edu/courses/6.899/lectures/faces.tar.gz

6 Principal Component Analysis

6.1 Objectives

This laboratory work describes the Principal Component Analysis method. It is applied as a means of dimensionality reduction, compression and visualization. A library capable of providing the eigenvalue decomposition of a matrix is required.

6.2 Theoretical Background

You are given a set of data points lying in a high dimensional space. Each point can represent the features of a training instance. Our goal is to reduce the dimensionality of the data points while preserving as much information as possible.

We begin with a simple 2D example. We plot the points corresponding to data collected about how different people enjoy certain activities and their skill in the respective domain. Figure 1 shows a cartoon example, from [2].

Consider now the two vectors u_1 and u_2. If we project the 2D points onto the vector u_2 we obtain scalar values with a small spread (standard deviation). If instead, we project it onto u_1 the spread is much larger. If we had to choose a single vector we would prefer to project onto u_1 since the points can still be discerned from each other.

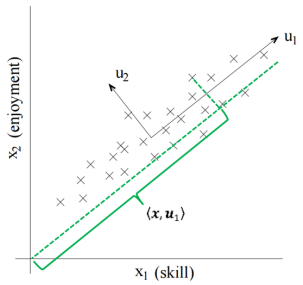


Figure 6.1 – Visualizing the projection of $\boldsymbol{x}$ along the $\boldsymbol{u}_1$ axis

More formally, each 2D point can be written as:

$$\boldsymbol{x} = \langle \boldsymbol{x}, \boldsymbol{u}_1 \rangle \boldsymbol{u}_1 / \|\boldsymbol{u}_1\| + \langle \boldsymbol{x}, \boldsymbol{u}_2 \rangle \boldsymbol{u}_2 / \|\boldsymbol{u}_2\|$$

Here we have projected $\boldsymbol{x}$ onto each vector and then added the corresponding terms. The dot product $\langle \boldsymbol{x}, \boldsymbol{u}_i \rangle$ gives the magnitude of

the projection, it needs to be normalized by the length of the vector $\|\boldsymbol{u}_i\|$ and the two vectors give the directions. This is possible since $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ are perpendicular. If we impose that they be also unit vectors then the normalization term disappears. See [4] for a better visualization.

The idea behind reducing the dimensionality of the data is to use only the largest projections. Since the projections onto $\boldsymbol{u}_2$ will be smaller we can approximate $\boldsymbol{x}$ with only the first term:

$$\tilde{\boldsymbol{x}}_{\mathbf{1}} = \langle \boldsymbol{x}, \boldsymbol{u}_1 \rangle \, \boldsymbol{u}_1 / \|\boldsymbol{u}_1\|$$

In general, given an orthonormal basis for a *d*-dimensional vector space called $\boldsymbol{B}$ with basis vectors $\boldsymbol{b}_i$ we can write any vector as:

$$\boldsymbol{x} = \sum_{i=1}^{d} \langle \boldsymbol{x}, \boldsymbol{b}_i \rangle \boldsymbol{b}_i = \sum_{i=1}^{d} (\boldsymbol{x}^T \boldsymbol{b}_i) \, \boldsymbol{b}_i$$

The question arises of how to determine the basis vectors onto which to perform the projections. Since we are interested in maximizing the preserved variance the covariance matrix could offer useful information. The covariance of two features is defined as:

$$C(i,j) = \frac{1}{n-1} \sum_{k=1}^{n} (x_{ki} - \mu_i)\big(x_{kj} - \mu_j\big)$$

where μ_i is the mean for feature *i* and x_{ki} is the *i*-th feature of the *k*-th point. The covariance matrix contains covariance values for all pairs. It can be shown that it can be expressed as a simple matrix product:

$$C = \frac{1}{n-1} (X - \boldsymbol{\mu}\mathbf{1}_{1xn})^T (X - \boldsymbol{\mu}\mathbf{1}_{1xn})$$

where $\boldsymbol{\mu}$ is a vector containing all mean values and $\mathbf{1}_{1xn}$ is a row vector containing only ones. If we extract the mean from the data as a preprocessing step the formula simplifies further:

$$C = \frac{1}{n-1} X^T X$$

The next step is to find the axes along which the covariance is maximal. Eigenvalue decomposition of a matrix offers such

information. Intuitively, (almost) any matrix can be viewed as a rotation followed by a stretching along the axes and the inverse rotation. The eigenvalue decomposition returns such a decomposition for the matrix:

$$C = Q\Lambda Q^T = \sum_{i=1}^{d} \lambda_i Q_i Q_i^T$$

where Q is a *dxd* rotation matrix (orthonormal) and Λ contains elements only on the diagonal representing stretching along each axis. The elements are called eigenvalues and each corresponding column from Q is its eigenvector. Since we want to preserve the projections with the largest variance we order the eigenvalues according to magnitude and pick the first k corresponding eigenvalues. In this way C can be approximated as:

$$\tilde{C}_k = Q_{1:k}\Lambda_{1:k}Q_{1:k}^T = \sum_{i=1}^{k} \lambda_i Q_i Q_i^T$$

where $Q_{1:k}$ is a *dxk* matrix with the first k eigenvectors and $\Lambda_{1:k}$ is a *kxk* diagonal matrix with the first k eigenvalues. If k equals d we obtain the original matrix and as we decrease k we get increasingly poorer approximations for C.

Thus we have found the axes along which the variance of the projections is maximized. Then, for a general vector its approximate using k vectors can be evaluated as:

$$\tilde{\boldsymbol{x}}_{\boldsymbol{k}} = \sum_{i=1}^{k} \langle \boldsymbol{x}, Q_i \rangle Q_i = \sum_{i=1}^{k} (\boldsymbol{x}^T Q_i)\, Q_i$$

where Q_i is the i^{th} column of the rotation matrix Q.

The PCA coefficients can be calculated as:

$$X_{coef} = XQ$$

PCA approximation can be performed on all the input vectors at once (if they are stored as rows in X) using the following formulas:

$$\tilde{\boldsymbol{X}}_{\boldsymbol{k}} = \sum_{\boldsymbol{i=1}}^{\boldsymbol{k}} XQ_iQ_i^t = \sum_{i=1}^{k} X_{coef_i}Q_i^t = XQ_{1:k}Q_{1:k}^T = X_{coef_{1:k}}Q_{1:k}^T$$

where $Q_{1:k}$ signifies the first k columns from the matrix Q. It is important to distinguish the approximation from the coefficients; the approximation is the sum of coefficients multiplied by the principal components.

We will end the theoretical description by giving several applications of PCA:

- reducing the dimensionality of features - in some cases a large feature vector may prohibit fast prediction;
- visualizing the data - we can inspect only data in 3D and 2D, for higher dimensional data a projection is necessary;
- approximating the data vectors;
- detecting redundant features and linear dependencies between features;
- noise reduction - if the noise term has less variance then the data (high signal-to-noise ratio) PCA eliminates the noise from the input

6.3 Practical Background

Declare and allocate an *nxd* matrix with double precision floating point values:

```
Mat X(n,d,CV_64FC1);
```

Calculate the covariance matrix after the means have been subtracted:

```
Mat C = X.t()*X/(n-1);
```

Perform eigenvalue decomposition on the covariance matrix *C*, *Lambda* will contain the eigenvalues and *Q* will contain the eigenvectors along columns. The transpose operation is necessary to follow the notation from the theoretical discussion.

```
Mat Lambda, Q;
eigen(C, Lambda, Q);
Q = Q.t();
```

Dot product is implemented as normal multiplication. Note that due to 0 indexing the first row is row(0). The dot product between row *i* of *X* and column *i* of *Q* is given by:

```
Mat prod = X.row(i)*Q.col(i);
```

6.4 Practical Work

1. Open the input file and read the list of data points. The first line contains the number of points *n* and the dimensionality of the data *d*. The following *n* lines each contain a single point with *d* coordinates.
2. Calculate the mean vector and subtract it from the data points.
3. Calculate the covariance matrix as a matrix product.
4. Perform the eigenvalue decomposition on the covariance matrix.
5. Print the eigenvalues.
6. Calculate the PCA coefficients and k^{th} approximate $\tilde{X}_k$ for the input data.
7. Evaluate the mean absolute difference between the original points and their approximation using *k* principal components.
8. Find the minimum and maximum values along the columns of the coefficient matrix.
9. For the input data from *pca2d.txt* select the first two columns from X_{coef} and plot the data as black 2D points on a white background. To obtain positive coordinates subtract the minimum values.
10. For input data from *pca3d.txt* select the first three columns from X_{coef} and plot the data as a grayscale image. Use the first two components as x and y coordinates and the third as intensity values. To obtain positive coordinates subtract the minimum values from the first two coordinates. Normalize the third component to the interval 0:255
11. Automatically select the required *k* which retains a given percent of the original variance. For example, find *k* for which the k^{th} approximate retains 99% of the original variance. The percentage of variance retained is given by $\sum_{i=1}^{k} \lambda_i / \sum_{i=1}^{d} \lambda_i$.

6.5 Example Results

For pca2d

- First eigenvalue is approximately 8090
- Mean absolute difference using only one dimension: 22.43

For pca3d

- First eigenvalue is 5462.3301
- Mean absolute difference using only one dimension: 14.5

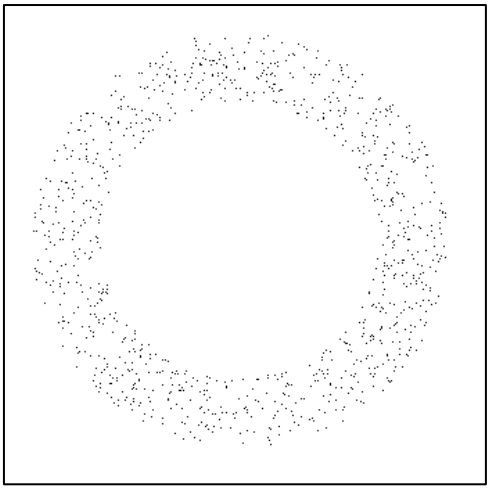


Figure 6.2 – Visualization of 2D points resulting from applying PCA on data from pca2d.txt

6.6 References

[1] Wikipedia article PCA - https://en.wikipedia.org/wiki/Principal_component_analysis

[2] Stanford Machine Learning course notes – http://cs229.stanford.edu/notes/cs229-notes10.pdf

[3] Lindsay Smith - PCA tutorial – http://faculty.iiit.ac.in/~mkrishna/PrincipalComponents.pdf

[4] PCA in R (animation of projection) - https://poissonisfish.wordpress.com/2017/01/23/principal-component-analysis-in-r/

7 K-means Clustering

7.1 Objectives

This laboratory session deals with the problem of clustering a set of points. This is a machine learning task that is unsupervised, i.e. the class labels of the points are not known and not required. Successful methods will identify the underlying structure in the data and group similar points together.

7.2 Theoretical Background

Cluster analysis or clustering is the task of grouping a set of objects in such a way that objects in the same group (also called a cluster) are more similar (in some sense or another) to each other than to those in other groups (clusters). It is a main task of exploratory data mining, and a common technique for statistical data analysis, used in many fields, including machine learning, pattern recognition, image analysis, information retrieval, bioinformatics, data compression, and computer graphics [1].

The input for the method is the set of data points: $X = \{x_i, i = 1{:}n\}$. Each point is d-dimensional $x_i = (x_{i1}, x_{i2}, \dots, x_{id})$. The goal of the k-means clustering method is to partition the points into *K* sets denoted by $S = \{S_k | k = 1{:}K\}$. The mean value of the points in each set is named m_k. The partitioning must minimize the following objective function:

$$J(X,S) = \sum_{k=1}^{K} \sum_{x \in S_k} dist(x, m_k)$$

where $dist(.,.)$ is the Euclidean distance function in d-dimensional space:

$$dist(x,y) = \sqrt{\sum_{i=1}^{d} (x_i - y_i)^2}$$

This is an NP-hard problem but there are several approximations that provide good results. Lloyd's method proposes to divide the problem into two parts. If we have the partitioning we can calculate the means, but we cannot know the partitions if the cluster centers are unknown.

The idea is to start with a random set of cluster centers and iteratively refine these. The algorithm is guaranteed to converge to a local minimum but it may not be the global minimum [2].

Let L denote the membership function for each point, so $L(i) \in 1{:}K, i = 1{:}n$. The membership function returns the cluster of the i[th] point. Start by assigning the cluster centers to random points from the dataset: $m_k = x_{r_k}$, where r_k is uniformly distributed random integer from *1:n*. In order to ensure better convergence more advanced initialization techniques can be applied. In [3] the authors define the k-means++ method. This relies on drawing the point with a given distribution that disfavors points that are close together.

Afterwards, perform several iterations of assignment steps and update steps. When the membership does not change or the maximum number of iterations is reached, the algorithm is halted. The steps of the method are given in the following algorithm.

K-means algorithm

Initialization – Randomly select the *K* centers from the set of input points. Let each r_k be a uniformly distributed random integer from *1:n*, then the initial means are chosen as:

$$m_k = x_{r_k}$$

Assignment - Assign each point from the input dataset to the closest cluster center found so far. The membership function will take the value of the index of the closest center:

$$L(i) = argmin_k dist(x_i, m_k)$$

Update – Recalculate the cluster centers based on the membership function. The new cluster centers are the means of the points from the cluster. In the following, summation is performed on all elements that are in cluster *k*, i.e. they have membership of *L(i) = k*.

$$m_k = \frac{\sum_{L(i)=k} x_i}{\sum_{L(i)=k} 1} = \frac{\sum_{x \in S_k} x}{|S_k|}$$

Halting condition - If there is no change in the membership function then the algorithm can be halted because further calculation will lead to no changes in the mean values. A maximum number of iterations can also be enforced. If none of the above conditions are met the algorithm continues with the assignment step.

7.3 Practical Background

Generating a random integer with uniform distribution between *a* and *b* (inclusive):

```
#include <random>

default_random_engine gen;
uniform_int_distribution<int> distribution(a,
b);
int randint = distribution(gen);
```

Creating a color image:

```
Mat img(height, width, CV_8UC3);
```

Assigning random colors to clusters:

```
const int K = 3;
Vec3b colors[K];
for(int i = 0; i<K; i++)
     colors[i] = { (uchar)distribution(gen),
                   (uchar)distribution(gen),
                   (uchar)distribution(gen) };
```

Assigning colors[k] to position (i,j):

```
img.at<Vec3b>(i,j) = colors[k];
```

7.4 Practical Work

1. Implement K-means on general input data (*d* dimensional points). Stop the algorithm once no change in the membership function is observed or after a certain number of maximum iterations. The number of clusters, *K*, is given by the user.
2. Apply K-means on a set of 2D points (input files points*.bmp) – in this case *d=2*.
   a. Choose random colors to visualize the clusters based on the resulting membership function.
   b. Color the neighborhood of points for better visualization.

c. Draw the Voronoi tessellation corresponding to the obtained cluster centers. For this picture you must color each image position (including the background) according to which is the closest center to it.

3. Apply K-means on a grayscale image. Use the intensity as the single feature for the input points – in this case *d=1*.
4. Recolor the input image based on the mean intensity of each cluster.
5. Apply K-means on a color image. Use the color components as the features for the input points – in this case *d=3*.
6. Recolor the input image based on the mean color of each cluster.
7. Optionally, implement the k-means++ initialization technique from [3].

7.5 Example Results

In the case of *d=2*, when K-means is run on a set of 2D points:

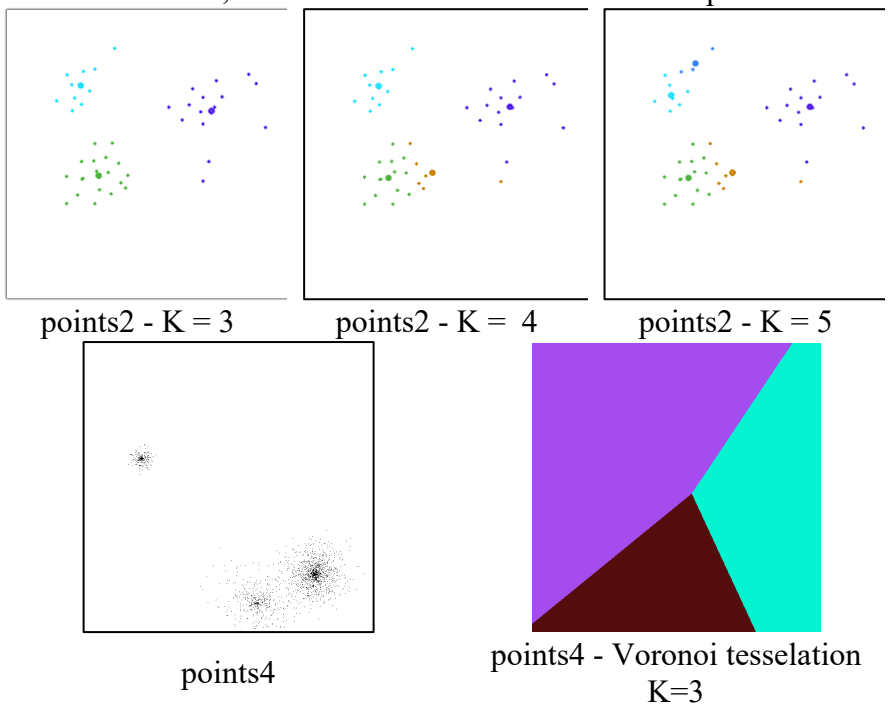


Figure 7.1 – Input 2D points and clusterization results with different K values

In the case of *d=1*, when K-means is run on a grayscale image:

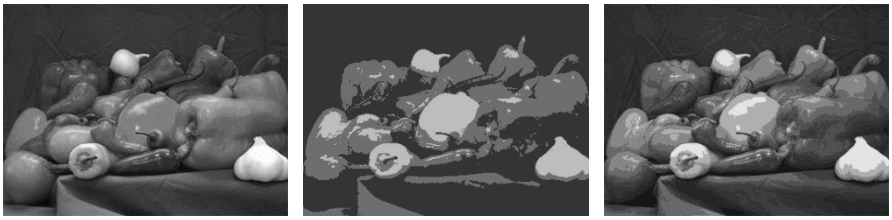


Figure 7.2 – Input grayscale image and its segmentation using K=3 and K=10 centers

In the case of *d=3*, when K-means is run on a color image:

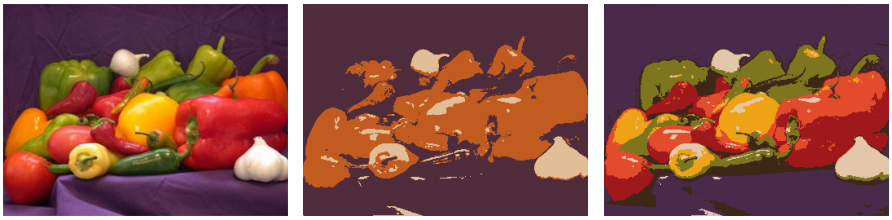


Figure 7.3 – Input color image and its segmentation using K=3 and K=10 centers

7.6 References

[1] Cluster analysis Wikipedia article - https://en.wikipedia.org/wiki/Cluster_analysis

[2] K-means Wikipedia article - https://en.wikipedia.org/wiki/K-means_clustering

[3] Arthur, David, and Sergei Vassilvitskii. "k-means++: The advantages of careful seeding." Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics, 2007.

[4] P. Arbelaez, M. Maire, C. Fowlkes and J. Malik. „Contour Detection and Hierarchical Image Segmentation", IEEE TPAMI, Vol. 33, No. 5, pp. 898-916, May 2011.

[5] Image segmentation dataset: https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/resources.html

8 K-Nearest Neighbor Classifier

8.1 Objectives

The purpose of this laboratory work is to introduce perhaps the simplest classifier: the k-Nearest Neighbor classifier. The classifier is applied on a small image dataset with multiple classes.

8.2 Theoretical Background

8.2.1 Introduction

The purpose of a classifier is to assign a class to an unknown sample. Each sample is described by a feature vector. Perhaps one the simplest classifiers is the k-NN classifier. It makes the decision about the input sample based on the K nearest neighbors from a labeled training dataset. The next figure illustrates this by showing the sample as a blue square among the labeled samples. A circle enclosing the 5 closest neighbors indicates the region which is used to infer the class of the test sample. The radius of the circle is **variable** and always encloses K neighbors.

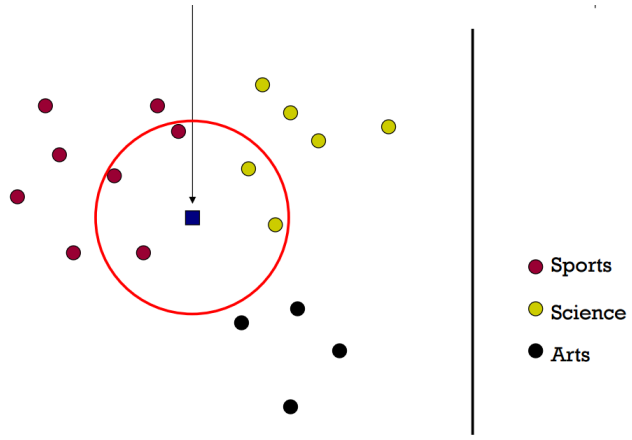


Figure 8.1 – 5-NN classifier example with three classes

The k-NN classifier is a non-parametric classifier, meaning that it does not construct a model for the classes it tries to distinguish. Instead it remembers the whole training set and at classification time the test instance is classified online. It can be labeled as a type of instance-based learning, or lazy learning, since the classifier function is only

approximated locally and all computation is deferred until classification.

8.2.2 Classification algorithm

Let the training dataset be defined in the form of a matrix of dimensions *nxd* denoted by *X*. Each line from *X* contains a single *d* dimensional feature vector called X_i, corresponding to a training instance. Also, let $\boldsymbol{y}$ denote the vector containing class labels. The dimension of $\boldsymbol{y}$ is *nx1*, each training instance having a class assigned to it. The elements of $\boldsymbol{y}$ are restricted to the set *{1,2,...,C}*, where *C* is the number of classes.

For an unknown test instance $\boldsymbol{x}$ the distance from each training example is calculated:

$$d_i = dist(x, X_i)$$

The distances are sorted in ascending order and the closest *K* instances are considered based on the distance. Each instance casts a vote for their class which is known from $\boldsymbol{y}$. The instance is classified as the class which has the most votes. A more formal description follows.

Let **p** be the permutation that sorts the distances in increasing order:

$$d_{p_1} < d_{p_2} < \cdots < d_{p_n}$$

The vote histogram is a *Cx1* vector constructed as:

$$\boldsymbol{h} = \sum_{k=1}^{K} \mathbf{1}(y_{p_k})$$

where $\mathbf{1}(y_{p_k})$ is a *Cx1* the indicator vector containing 1 only at the position $\boldsymbol{y}_{p_k}$ and 0 elsewhere. The sum accumulates the votes from the *K* closest neighbors. The class of the unknown instance is selected as:

$$c = argmax_i \boldsymbol{h}_i$$

There are multiple versions of the algorithm depending on the distance function used and the voting scheme. For example, votes can be weighed based on inverse distance using the following formula:

$$\boldsymbol{h} = \sum_{k=1}^{K} \frac{\mathbf{1}(y_{p_k})}{1 + d_{p_k}}$$

where we have added 1 to the distance to avoid division by 0 and to obtain a weight of 1 for distances equal 0.

The parameter *K* controls how many neighbors are considered. If *K=1*, only the nearest neighbor is considered. Increasing its value reduces the effect of noise on the result but makes the boundaries between the classes less distinct. In the extreme case when *K=n*, the whole training set is considered. If the votes are not weighted, this would classify an instance based on the prior distribution of the classes from the training set. In practice *K* is chosen to be an odd number to break ties when there are only two classes. Tests are performed on a validation set to obtain a proper value for *K* (hyper-parameter optimization).

The presented approach can also be used to perform regression if instead of choosing the class. In this case, we need to construct a weighted sum of the training instances as a response. The error rate of a k-NN classifier approaches that of the ideal Bayes error rate and is bounded by twice the Bayes error for two classes and for $n \to \infty$.

8.2.3 Global image features

Color images can be characterized by a global feature vector for the purpose of classification. A global feature vector of fixed dimension for any input image enables the process of classification. Global features usually describe certain relevant statistics of the image but lose information about the spatial layout of the image.

The image histogram can be viewed as a global feature vector for the image. The basic definition for a histogram of a grayscale image is that of a vector which counts the occurrences of each gray level intensity. It is a vector of dimension 256. In general, the histogram can be a vector of length *m* if we divide the [0,255] interval in *m* equal parts. In this case each bin in the histogram vector counts the number of gray level intensities falling in that particular bin. For example: if *m=8,* the first bin would count all intensities between *0* and *256/m - 1=31;* the second bin between *32* and *63*; and so on. The histogram for a color image can be formed by concatenating the individual histograms for the separate channels. The size of the resulting histogram is of *3 x m*.

8.2.4 Evaluation of classifiers

Multiple metrics can be calculated to evaluate the performance of the classifier. The **confusion matrix** for a labeled dataset can be defined as a matrix containing in each cell M_{ij} the number of instances classified by the classifier into class i while having true class j. The ideal classifier would assign all instances to their correct class and would have large entries on the diagonal of the confusion matrix M_{ii}. In general, the values show which classes are confused with each other and can help to improve the classifier performance by identifying specific features that aid the discrimination between the two classes.

	Real Class: *Positive*	**Real Class: *Negative***
Predicted class: *Positive*	**TP** (*True Positive*)	**FP** (*False Positive*)
Predicted class: *Negative*	**FN** (*False Negative*)	**TN** (*True Negative*)

Figure 8.2 – Confusion matrix

The accuracy for the classifier on a labeled test set is defined as the percentage of correctly classified instances. It is the complementary metric to the error rate. It does not offer relevant information if the class distribution is skewed. If the number of instances is unbalanced, a classifier that always predicts the most prevalent class will have a high accuracy. This is the typical situation, for example: pedestrian classifiers deal with a highly skewed distribution of much more background image samples than pedestrian samples. In this case, more relevant metrics are precision and recall for each class.

The accuracy can be calculated from the confusion matrix as:

$$Acc = \frac{\sum_{i=1}^{C} M_{ii}}{\sum_{i=1}^{C} \sum_{j=1}^{C} M_{ij}}$$

8.2.5 Scene Recognition Dataset Statistics

The dataset for this session is for scene recognition. It contains 6 different classes: beach, city, desert, forest, landscape and snow.

Images for each class are stored in subfolders and named as six digit numbers. The dataset is slightly imbalanced, the number of examples for each class ranging from 35 to 277. The training set contains 672 files, and the test set contains 85 files. Sample images from each class are given below:

Figure 8.3 – Sample images

8.3 Practical Background

Suggestion for the histogram function header (the *hist* array is allocated previously):

```
void calcHist(Mat img, int nr_bins, int* hist)
```

Define the class names:

```
const int nrclasses = 6;
char classes[nrclasses][10] =
{"beach", "city", "desert", "forest", "landscape",
"snow"};
```

Allocate the feature matrix and the label vector:

```
Mat X(nrinst, feature_dim, CV_32FC1);
Mat y(nrinst, 1, CV_8UC1);
```

Read all images from class *c*, calculate the histogram and insert the values in *X*:

```
int c = 0, fileNr = 0, rowX = 0;
while(1){
      sprintf(fname, "train/%s/%06d.jpeg", classes[c],
fileNr++);
      Mat img = imread(fname);
      if (img.cols==0) break;

      //call function to calculate the histogram in
hist

      for(int d=0; d<hist_size; d++)
            X.at<float>(rowX, d) = hist[d];
```

```
        y.at<uchar>(rowX) = c;
        rowX++;
}
```

Allocate the confusion matrix:

```
Mat C(nrclasses, nrclasses, CV_32FC1); //or CV_32SC1
```

8.4 Practical Work

1. Implement a function for extracting the color histogram of an image.
2. Read all the images from the training set. For each image compute the color histogram with general bin size *m* and save it as a row in the feature matrix ***X***. Save the corresponding class label in the label vector ***y***.
3. Implement the k-NN classifier for an unknown image and for a general *K* value.
4. Evaluate the classifier on the test set by calculating the confusion matrix and the overall accuracy.
5. Try out different values for the number of bins for the histogram and the parameter *K* to see which feature attains the best performance. Aim for over 65% accuracy.
6. Convert the input image into Luv or HSV color-space before histogram calculation.
7. Optionally, try out more complex features (such as histograms on image regions) or other distance metrics (Manhattan distance, weighted Euclidean).

8.5 References

[1] Wikipedia article - k-NN classifier
https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm
[2] Andrew Ng - Machine Learning: Nonparametric methods & Instance-based learning
http://www.cs.cmu.edu/~epxing/Class/10701-08s/Lecture/lecture2.pdf

9 Naive Bayes Classifier

9.1 Objectives

In this laboratory session we will study the Naive Bayes Classifier and we will apply it to a specific recognition problem: we will learn to distinguish between images of handwritten digits.

9.2 Theoretical Background

The Naive Bayes classifier takes as input a list of features, applies the Bayes rule and assumes that the features are independent to calculate the posterior probability of each class. The class with the highest probability is chosen as the output.

Due to the independence assumption, it can handle an arbitrary number of independent variables whether continuous or categorical. Given a set of random variables, $\boldsymbol{x} = \{x_1, x_2, \dots, x_d\}$, we want to construct the posterior probability for the random variable *C,* having the set of possible outcomes $C = \{c_1, c_2, \dots, c_J\}$. Using different terminology, the elements $\boldsymbol{x}$ are the predictors or features and $\boldsymbol{C}$ is the set of categorical levels or classes present in the dependent variable. Using Bayes' rule we can write:

$$p(c|x_1, x_2, \dots, x_d) \propto P(c)p(x_1, x_2, \dots, x_d|c)$$

where $p(c|x_1, x_2, \dots, x_d)$ is the posterior probability of class membership, i.e., the probability that $\boldsymbol{x}$ belongs to *C* given the features; $p(x_1, x_2, \dots, x_d|c)$ is the likelihood and $P(c)$ is the prior. Naive Bayes assumes that the feature values are independent given the class so we can decompose the likelihood into a product of terms:

$$p(\boldsymbol{x}|c) = \prod_{k=1}^{d} p(x_k|c)$$

and rewrite the posterior probability as:

$$p(c|\boldsymbol{x}) \propto P(c) \prod_{k=1}^{d} p(x_k|c)$$

Using Bayes' rule from above, we can implement a classifier that predicts the class based on the input features $\boldsymbol{x}$. This is achieved by selecting the class c that achieves the highest posterior probability.

$$c^* = argmax_j p(c_j|\boldsymbol{x})$$

Although the assumption that the predictor variables (features) are independent is not always accurate, it does simplify the classification task dramatically, since it allows the class conditional densities $p(x_k|c)$ to be calculated separately for each variable. In effect, Naive Bayes reduces a high-dimensional density estimation task to multiple one-dimensional kernel density estimations. Furthermore, the assumption does not seem to affect the posterior probabilities, especially in regions near decision boundaries, thus, it leaves the classification task unaffected. The probability density functions can be modeled in several different ways including normal, log-normal, gamma and Poisson density functions and discrete versions.

9.3 Practical Background

9.3.1 MNIST handwritten dataset

We will use a standard benchmark for digit recognition to evaluate the performance of the classifier. The MNIST dataset was assembled by Yann LeCun from multiple datasets. The training set contains 60000 images of handwritten digits from approximately 250 writers. The test set contains 10000 instances. The distribution of the digits is roughly uniform. For more details visit the link from [2]. We will use binomial probability distributions to model the probability density functions.

9.3.2 Training algorithm

Let *X* denote the feature matrix for the training set, as usual. In this case *X* contains on every row the binarized values of each training image to either 0 or 255 based on a selected threshold. *X* has the dimension *n x d*, where *n* is the number of training instances and *d=28 x 28* is the number of features which is equal to the size of an image. The class labels are stored in the vector *y* of dimension *n*.

The prior for class *i* is calculated as the fraction of instances from class *i* from the total number of instances:

$$P(C = i) = {}^{n_i}/_{n}$$

The likelihood of having feature *j* equal to 255 given class *i* is given by the fraction of the training instances which have feature *j* equal to 255 and are from class *i*:

$$p(x_j = 255|C = i) = \frac{count(X_{kj} = 255 \wedge y_k = i)}{n_i}$$

The likelihood of having feature *j* equal to 0 is the complementary event so:

$$p(x_j = 0|C = i) = 1 - p(x_j = 255|C = i)$$

To avoid multiplication by zero in the posterior probability, likelihoods having the value of 0 need to be treated carefully. A simple solution is to change all values smaller than 10^{-5} to 10^{-5}. Another alternative is to use Laplace smoothing, where $|C|$ signifies the number of classes:

$$p(x_j = 255|C = i) = \frac{count(X_{kj} = 255 \wedge y_k = i) + 1}{n_i + |C|}$$

9.3.3 Classification algorithm

Once the likelihood values and priors are calculated classification is possible. The values for the likelihood are in the interval [0,1] and the posterior is a product of 784 numbers each less than 1. To avoid precision problems, it is recommended to work with the logarithm of the posterior. Denote the test vector as *T* and its elements as T_j. These are the binarized values from the test image in the form of a vector. The log posterior of each class can be evaluated as:

$$\log(p(C = i|T)) \propto \log(P(C = i)) + \sum_{j=1}^{d} \log\left(p(x_j = T_j|C = i)\right)$$

Since the ordering of the posteriors does not change when the log function is applied, the predicted class will be the one with the highest log posterior probability value. The log of the total probability can be ignored since it is a constant.

9.3.4 Implementation details

Load the first 100 images from class c:

```
char fname[256];
int c = 1;
int index = 0;
```

```
while(index<100){
	sprintf(fname, "train/%d/%06d.png", c, index);
	Mat img = imread(fname, 0);
	if (img.cols==0) break;
	//process img
	index++;
}
```

The prior is a Cx1 vector:

```
const int C = 3; //number of classes
Mat priors(C,1,CV_64FC1);
```

The likelihood is a Cxd vector (we only store the likelihood for 255):

```
const int d = 28*28;
Mat likelihood(C,d,CV_64FC1);
```

Header suggestion for the classifier:

```
int classifyBayes(Mat img, Mat priors, Mat likelihood);
```

9.4 Practical Work

1. Load each image from the training set, perform binarization and save the values in the training matrix *X*. Save the class label in the label vector *y*. For the initial version use only the first 100 images from the first two classes.
2. Implement the training method.
   a. Compute and save the priors for each class.
   b. Compute and save the likelihood values for each class and each feature. Apply Laplace smoothing to avoid zero values.
3. Implement the Naive Bayes classifier for an unknown image.
4. Display the log posterior for each class. Optionally, convert the values to proper probabilities.
5. Evaluate the classifier on the test images and calculate the confusion matrix and the error rate. The error rate is the fraction of misclassified test instances (the complementary metric to the accuracy).
6. Train and evaluate on the full dataset

9.5 References

[1] *Electronic Statistics Textbook* – http://www.statsoft.com/textbook/stnaiveb.html
[2] LeCun, Yann, Corinna Cortes, and Christopher JC Burges. "The MNIST database." (1998) http://yann.lecun.com/exdb/mnist

10 Perceptron Classifier

10.1 Objectives

This laboratory session presents the perceptron learning algorithm for the linear classifier. We will apply gradient descent and stochastic gradient descent procedure to obtain the weight vector for a two-class classification problem.

10.2 Theoretical Background

The goal of classification is to separate items into different classes or groups. A linear classifier achieves this goal via a discriminant function that is the linear combination of the features.

10.2.1 Definitions

Define a training set as the tuple *(X,Y)*, where $X \in M_{n\times m}(R)$ and *Y* is a vector $Y \in M_{n\times 1}(D)$, where *D* is the set of class labels. *X* represents the concatenation the feature vectors for each sample from the training set, where each row is an *m* dimensional vector representing a sample. *Y* is the vector the desired outputs for the classifier. A classifier is a map from the feature space to the class labels: $f: R^m \to D$.

Thus a classifier partitions the feature space into |*D*| **decision regions.** The surface separating the classes is called **decision boundary.** If we have only two dimensional feature vectors the decision boundaries are lines or curves. In the following we will discuss binary classifiers. In this case the set of class labels contains exactly two elements. We will denote the labels for classes as *D={-1,1}*.

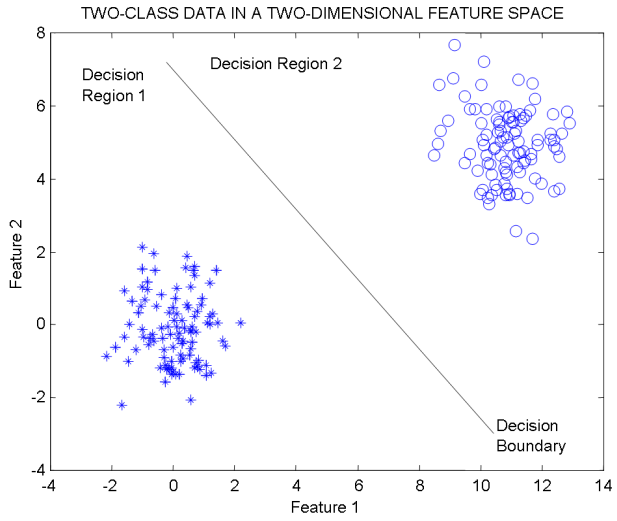


Figure 10.1. Example of a linear classifier for a two-class classification problem. Each sample is characterized by two features. The decision boundary is a line.

10.2.2 General form of a linear classifier

The simplest classifier is a linear classifier. A linear classifier outputs the class labels based on a linear combination of the input features. Considering $\boldsymbol{x} \in M_{m\times 1}(R)$ as a feature vector we can write the linear decision function as:

$$g(\boldsymbol{x}) = w_0 + \sum_{i=1}^{m} w_i x_i = w_0 + \boldsymbol{w}^T \boldsymbol{x}$$

Where

- $\boldsymbol{w}$ is the $m \times 1$ weight column vector
- w_0 is the bias or the threshold weight

A schematic view of the linear classifier is given in the next figure.

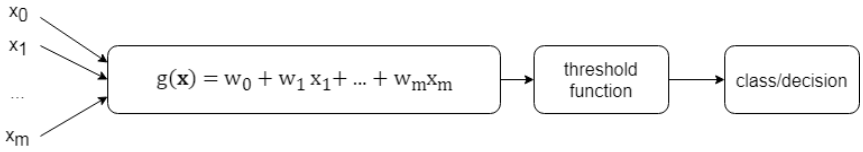


Figure 10.2 – Schematic view of a linear classifier for multidimensional data

For convenience, we will absorb the intercept w_0 into $\boldsymbol{w}$ by augmenting the feature vector $\boldsymbol{x}$ with an additional constant dimension (let the bar over a variable denote the augmented version of the vector):

$$w_0 + \boldsymbol{w}^T\boldsymbol{x} = [w_0 \quad \boldsymbol{w}^T]\begin{bmatrix}1\\ \boldsymbol{x}\end{bmatrix} = \bar{\boldsymbol{w}}^T\bar{\boldsymbol{x}}$$

A two-category linear classifier (or binary classifier) implements the following decision rule:

$$\begin{cases} if\ g(x) > 0\ decide\ that\ sample\ x\ belongs\ to\ class\ +1 \\ if\ g(x) < 0\ decide\ that\ sample\ x\ belongs\ to\ class\ -1 \end{cases}$$

or

$$\begin{cases} if\ w^T x > -w_0\ decide\ that\ sample\ x\ belongs\ to\ class\ +1 \\ if\ w^T x < -w_0\ decide\ that\ sample\ x\ belongs\ to\ class\ -1 \end{cases}$$

If $g(\boldsymbol{x}) = 0, \boldsymbol{x}$ can ordinarily be assigned to either class.

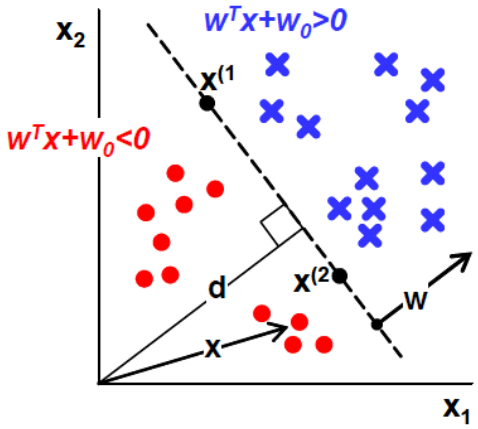


Figure 10.3 – Image for 2D case depicting: decision regions (red and blue), linear decision boundary (dashed line), weight vector ($\boldsymbol{w}$) and bias ($\boldsymbol{w_0} = \boldsymbol{d} \cdot \|\boldsymbol{w}\|$).

10.2.3 Learning algorithms for linear classifiers

We will present two learning algorithms for linear classifiers. In order to perform learning we transform the task into an optimization problem. For this we define a loss function L. The loss function applies a penalty for every instance that is classified into the wrong class. The perceptron algorithm adopts the following form for the loss function:

$$L(\bar{\boldsymbol{w}}) = \frac{1}{n}\sum_{i=1}^{n} max(0, -y_i\bar{\boldsymbol{w}}^T \cdot \bar{\boldsymbol{x}}_i) = \frac{1}{n}\sum_{i=1}^{n} L_i\,(\bar{\boldsymbol{w}})$$

If an instance i is classified correctly, no penalty is applied because the expression $-y_i\bar{\boldsymbol{w}}^T \cdot \bar{\boldsymbol{x}}_i$ is negative. In the case of a misclassification, the previous expression will be positive and it will be added to the function value. The objective now is to find the weights that minimize the loss function.

Gradient descent can be employed to find the global minimum of the loss function. This relies on the idea that a differentiable multivariate function decreases fastest in the opposite direction of the gradient. The update rule according to this observation is:

$$\bar{\boldsymbol{w}}_{k+1} \leftarrow \bar{\boldsymbol{w}}_k - \eta \boldsymbol{\nabla L}(\bar{\boldsymbol{w}}_k)$$

where $\bar{\boldsymbol{w}}_k$ is the weight vector at time k, η is a parameter that controls the step size and is called the learning rate, and $\boldsymbol{\nabla L}(\bar{\boldsymbol{w}})$ is the gradient vector of the loss function at point $\bar{\boldsymbol{w}}_k$. The gradient of the loss function is:

$$\boldsymbol{\nabla L}(\bar{\boldsymbol{w}}) = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{\nabla L_i}(\bar{\boldsymbol{w}})$$

$$\boldsymbol{\nabla L_i}(\bar{\boldsymbol{w}}) = \begin{cases} 0, & if \;\; y_i\bar{\boldsymbol{w}}^T \cdot \bar{\boldsymbol{x}}_i > 0 \\ -y_i\bar{\boldsymbol{x}}_i, & otherwise \end{cases}$$

In the standard gradient descent approach we update the weights only after visiting all the training examples. This is also called the batch-update learning algorithm. We can use stochastic gradient descent instead. This entails updating the weights after visiting each training example resulting in the classical online perceptron learning algorithm from [1]. In this case the update rule becomes:

$$\bar{\boldsymbol{w}}_{k+1} \leftarrow \bar{\boldsymbol{w}}_k - \eta \boldsymbol{\nabla L_i}(\bar{\boldsymbol{w}})$$

Algorithm:
Batch Perceptron

```
init w, η, E_limit, max_iter
for iter=1:max_iter
  E = 0, L = 0
  ∇L = [0,0,0]
  for i=1:n
    z_i = Σ_{j=0}^{d} w_j X_ij
    if z_i · y_i ≤ 0
      ∇L ← ∇L − y_i X_i
      E ← E + 1
      L ← L − y_i z_i
    endif
  endfor
  E ← E/n
  L ← L/n
  ∇L ← ∇L /n
  if E < E_limit
    break
  w ← w − η∇L
endfor
```

Algorithm:
Online Perceptron

```
init w, η, E_limit, max_iter
for iter=1:max_iter
  E = 0
  for i=1:n
    z_i = Σ_{j=0}^{d} w_j X_ij
    if z_i · y_i ≤ 0
      w ← w + η X_i y_i
      E ← E + 1
    endif
  endfor
  E ← E/n
  if E < E_limit
    break
endfor
```

10.3 Practical Background

In this laboratory session we will find a linear classifier that discriminates between two sets of points. The points in class 1 are colored in red and the points in class 2 are colored in blue.

Each point is described by the color (that denotes the class label) and the two coordinates, x_1 and x_2.

The augmented weight vector will have the form $\bar{w} = [w_0\ w_1\ w_2]$.
The augmented feature vector will be $\bar{x} = [1\ x_1\ x_2]$.

10.4 Practical Work

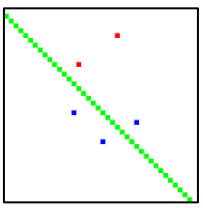
1. Read the points from a single file *test0*.bmp* and construct the training set *(X,Y)*. Assign the class label +1 to red points and -1 to blue points.
2. Implement and apply the online perceptron algorithm to find the linear classifier that divides the points into two groups. Suggestion for parameters:
   $\eta = 10^{-4}, w0 = [0.1, 0.1, 0.1], Elimit = 10^{-5}, max\_iter = 10^{5}$.
   Note: for a faster convergence use a larger learning rate only for w_0
3. Draw the final decision boundary based on the weight vector $\boldsymbol{w}$.
4. Implement the batch perceptron algorithm and find suitable parameters values. Show the loss function at each step. It must decrease slowly.
5. Visualize the decision boundary at intermediate steps, while the learning algorithm is running.
6. Change the starting values for the weight vector $\boldsymbol{w}$, the learning rate and terminating conditions to observe what happens in each case. What does an oscillating cost function signal?

10.5 Numerical example

Consider the points from the file points00 as (x,y) pairs or (column, row):

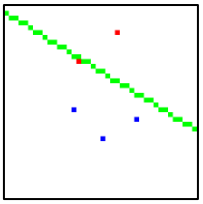
- Red points: (23, 5), (15,11) – class +1
- Blue points: (14, 21), (27,23), (20, 27) – class -1

The steps for the online perceptron algorithm are given below:

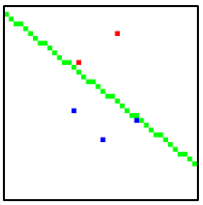


Learning rate = 0.01

Iteration 0
i=0: w=[1.000000 1.000000 -1.000000] xi=[1 23 5] yi = 1 zi=19.000000
i=1: w=[1.000000 1.000000 -1.000000] xi=[1 15 11] yi = 1 zi=5.000000
i=2: w=[1.000000 1.000000 -1.000000] xi=[1 14 21] yi = -1 zi=-6.000000
i=3: w=[1.000000 1.000000 -1.000000] xi=[1 27 23] yi = -1 zi=5.000000
wrong
update w0 = w0 - 0.01, w1 = w1 - 27*0.01, w2 = w2 – 23*0.01
i=4: w=[0.990000 0.730000 -1.230000] xi=[1 20 27] yi = -1 zi=-17.620000



Iteration 1
i=0: w= [0.990000 0.730000 -1.230000] xi= [1 23 5] yi = 1 zi=11.630000
i=1: w= [0.990000 0.730000 -1.230000] xi= [1 15 11] yi = 1 zi=-1.590000
wrong
update w0 = w0 + 0.01, w1 = w1 + 15*0.01, w2 = w2 + 11*0.01
i=2: w= [1.000000 0.880000 -1.120000] xi= [1 14 21] yi = -1 zi=-10.200000
i=3: w= [1.000000 0.880000 -1.120000] xi= [1 27 23] yi = -1 zi=-1.000000
i=4: w= [1.000000 0.880000 -1.120000] xi= [1 20 27] yi = -1 zi=-11.640000



Iteration 2
i=0: w= [1.000000 0.880000 -1.120000] xi= [1 23 5] yi = 1 zi=15.640000
i=1: w= [1.000000 0.880000 -1.120000] xi= [1 15 11] yi = 1 zi=1.880000
i=2: w= [1.000000 0.880000 -1.120000] xi= [1 14 21] yi = -1 zi=-10.200000
i=3: w= [1.000000 0.880000 -1.120000] xi= [1 27 23] yi = -1 zi=-1.000000
i=4: w= [1.000000 0.880000 -1.120000] xi= [1 20 27] yi = -1 zi=-11.640000
All classified correctly

10.6 References

[1] Rosenblatt, Frank (1957), The Perceptron - a perceiving and recognizing automaton. Report 85-460-1, Cornell Aeronautical Laboratory.

[2] Richard O. Duda, Peter E. Hart, David G. Stork: Pattern Classification 2nd ed.

[3] Xiaoli Z. Fern, Machine Learning and Data Mining Course, Oregon University - http://web.engr.oregonstate.edu/~xfern/classes/cs434/slides/perceptron-2.pdf

[4] Gradient Descent - http://en.wikipedia.org/wiki/Gradient_descent

[5] Avrim Blum, Machine Learning Theory, Carnegie Mellon University - https://www.cs.cmu.edu/~avrim/ML10/lect0125.pdf

11 AdaBoost Method

11.1 Objectives

In this laboratory session we will study a method for obtaining an ensemble classifier called AdaBoost (Adaptive Boosting). We will apply it for a binary classification problem on 2D points.

11.2 Theoretical Background

AdaBoost, short for Adaptive Boosting, is a machine learning meta-algorithm formulated by Yoav Freund and Robert Schapire in [1], who won the 2003 Gödel Prize for their work [2]. In this session the goal will be to separate 2D points into two classes, the class membership is given by the color of the points.

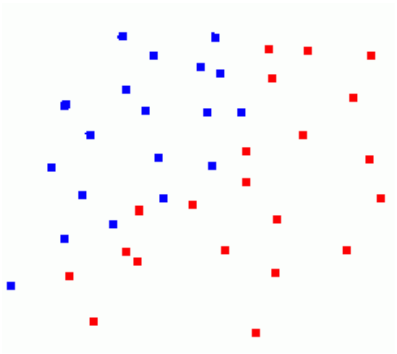


Figure 11.1 – Example of samples from two classes (red and blue points)

The general idea of the AdaBoost algorithm is to build a strong classifier $H_T(\boldsymbol{x})$ which is the sign of the linear combination of T weak classifiers (or weak learners) h_t:

$$H_T(\boldsymbol{x}) = sign\left(\sum_{t=1}^{T} \alpha_t h_t(\boldsymbol{x})\right)$$

Each weak learner returns either +1 or -1 and is weighted by α_t. The final class is given by the sign of the strong classifier $H_T(\boldsymbol{x})$. In this work, we will use **decision stumps** as weak learners. A decision stump classifies an instance by looking at a particular feature, if this feature is below a threshold, the instance is classified as class +1 and -1 otherwise.

We are given the training set in the following form: $\boldsymbol{X}$ is the feature matrix of dimension $n\,x\,m$ and contains n the training samples, each row being an individual sample of dimension m. In our case, $m = 2$ and the features are the rows and columns at which the points are found in the input image. The class vector $\boldsymbol{y}$ of dimension n contains +1 for each red point and -1 for each blue point.

For this method we will associate a weight with each example. We will store the weights in the weight vector $\boldsymbol{w}$ of dimension n. Initially all samples have an equal weight of $1/n$. The following algorithm describes the high level AdaBoost procedure which finds the strong classifier H_T.

Algorithm AdaBoost

```
init wᵢ=1/n
for t=1:T
  //also returns the weighted training error εₜ:
  [hₜ, εₜ] = findWeakLearner(X,y,w)
  αₜ = 0.5 ln((1−εₜ)/εₜ)
  s = 0
  for i=1:n
    //wrongly classified examples obey: yᵢhₜ(Xᵢ) < 0
    //their weights will become larger
    wᵢ ← wᵢ · exp(−αₜyᵢhₜ(Xᵢ))
    s += wᵢ
  endfor
  //normalize the weights
  for i=1:n
    wᵢ ← wᵢ/s
  endfor
endfor
//returns all the alpha values
//and the weak learners
return [α, h]
```

```
[h_t,ε_t] = findWeakLearner(X,y,w)
best_h = {}
best_err = ∞
for j=1:X.cols
  for threshold=0:max(img.cols, img.rows)
    for class_label={-1,1}
      e=0
      for i=1:X.rows
        if X(i,j)<threshold
          z_i=class_label
        else
          z_i=-class_label
        endif
        if z_i y_i < 0
          e += w_i
        endif
      endfor
      if e<best_err
        best_err = e
        best_h = {j, threshold, class_label, e}
      endif
    endfor
  endfor
endfor
return [best_h, best_err]
```

The underlying idea behind this algorithm is to find the best simple (weak) classifier and then to modify the importance of the examples. Missclassified examples will get a higher weight and correctly classified examples will get a lower weight. An example is classified as the wrong class if the sign of the expression $y_i h_t(X_i)$ is negative (the predicted and correct class labels have different signs).

In the following step, when we search for the next weak learner, it will be more important to correctly classify the examples which have higher weights since they contribute more to the weighted training error.

Each weak learner contributes to the final score of the classifier. The contribution is weighted by how well the weak learner performed in terms of the weighted training error.

11.3 Practical Background

Suggested structure for a single weak leaner (assumes *X* stores floats):

```
struct weaklearner{
  int feature_i;
  int threshold;
  int class_label;
  float error;
  int classify(Mat X){
    if (X.at<float>(feature_i)<threshold)
      return class_label;
    else
      return -class_label;
  }
};
```

Header for function that finds the best weak learner – note that the `weaklearner` structure stores the weighted error:

```
weaklearner findWeakLearner(Mat X, Mat y, Mat w)
```

Suggested structure for the strong classier (`MAXT` is a constant):

```
struct classifier{
  int T;
  float alphas[MAXT];
  weaklearner hs[MAXT];
  int classify(Mat X){
    return 0;
  }
};
```

Header for function which draws the decision boundary (keep the original image unmodified):

```
void drawBoundary(Mat img, classifier clf)
```

11.4 Practical Work

1. Read the training set from one of the input files (points*.bmp). Each row from the feature matrix $\boldsymbol{X}$ should contain the row and column of each colored point from the image. The class vector $\boldsymbol{y}$ contains +1 for red and -1 for blue points.
2. Implement the decision stump weak learner – the `weaklearner` structure.
3. Implement the `findWeakLearner` function.
4. Implement the `drawBoundary` function which colors the input image showing the decision boundary by changing the background color (white pixels) based on the classification result. Use yellow for +1 background and teal for -1 background pixels. Test the function with a strong classifier formed by a single weak learner.
5. Implement the AdaBoost algorithm to find the strong classifier with T weak learners. Visualize the decision boundary. For each input image find the value of T which results in zero classification error. What are the limitations of the presented method?

11.5 Example Results

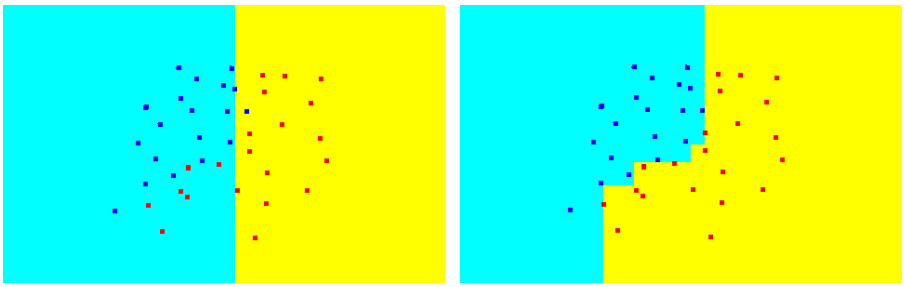


Figure 11.2 – Sample results on points1 with number of weak learners T=1 (left) and T=13 (right)

11.6 References

[1] Robert E. Schapire, The Boosting Approach to Machine Learning, An Overview, 2001

[2] AdaBoost - https://en.wikipedia.org/wiki/AdaBoost

12 Support Vector Machine

12.1 Objectives

In this lab session we will implement the simple linear classifier described in the previous lab and we will study the mechanisms of support vector classification based on soft margin classifiers.

12.2 Theoretical Background

12.2.1 Hard-margin classifiers

We will start the discussion from a simple problem of separating a set of points into two classes, as depicted in Figure 12.1igure 12.1:

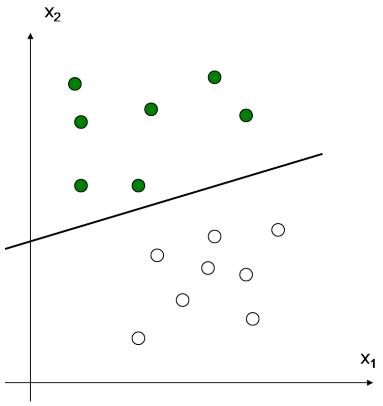


Figure 12.1 – A set of linearly separable points

The question here is how can we classify these points using a linear discriminant function in order to minimize the training error rate? We have an infinite number of answers, as shown in Figure 12.2:

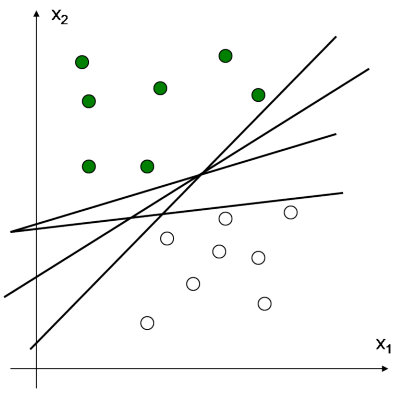


Figure 12.2 – Linear classifiers that correctly discriminate between the two classes

From the multitude of solutions we need to find which the best one is. One possible answer is given by the linear discriminant function with the maximum margin. Informally, the margin is defined as the width by which the boundary can be increased by before hitting a data point, see Figure 12.3.

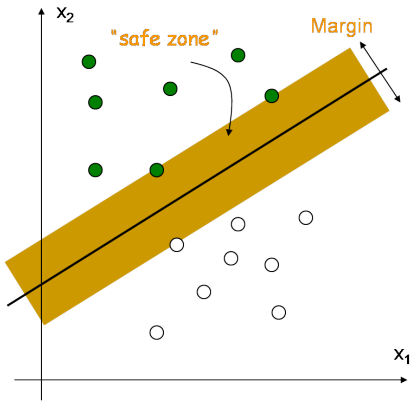


Figure 12.3 – The margin of a linear classifier

Such a classifier is robust to outliners and thus has strong generalization ability.

12.2.2 Optimization problem

Given a set of data points $\boldsymbol{x}_i$ with their class labels y_i for $i = 1,2,\dots,n$ our goal is to find $\boldsymbol{w}$ such that for any i:

$$\boldsymbol{w}^T\boldsymbol{x}_i + b > 0 \text{ if } y_i = 1$$
$$\boldsymbol{w}^T\boldsymbol{x}_i + b < 0 \text{ if } y_i = -1$$

With a scale transformation on both $\boldsymbol{w}$ and b, the above is equivalent to:

$$\boldsymbol{w}^T\boldsymbol{x}_i + b \geq 1 \text{ if } y_i = 1$$
$$\boldsymbol{w}^T\boldsymbol{x}_i + b \leq -1 \text{ if } y_i = -1$$

Choosing two points from the positive and negative sides of the boundary we know that:

$$\boldsymbol{w}^T\boldsymbol{x}^+ + b = 1$$
$$\boldsymbol{w}^T\boldsymbol{x}^- + b = -1$$

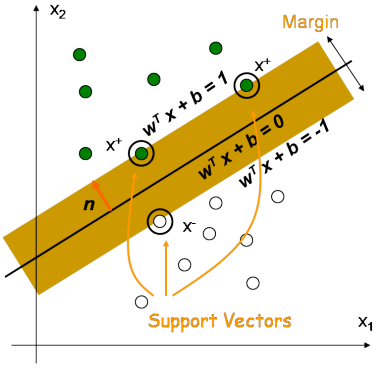


Figure 12.4 – Positive and negative samples nearest to the separation boundary – support vectors

Based on this the width of the margin is:

$$M = (\boldsymbol{x}^+ - \boldsymbol{x}^-) \cdot \boldsymbol{n} = (\boldsymbol{x}^+ - \boldsymbol{x}^-) \cdot \frac{\boldsymbol{w}}{||\boldsymbol{w}||} = \frac{2}{||\boldsymbol{w}||}$$

This margin should be maximized. The maximization problem is difficult to solve because it depends on $||\boldsymbol{w}||$, the norm of $\boldsymbol{w}$, which involves a square root. Fortunately it is possible to alter the equation by substituting $||\boldsymbol{w}||$ with $\frac{1}{2}||\boldsymbol{w}||^2$ without changing the solution (the minimum of the original and the modified equation have the same $\boldsymbol{w}$ and b).

The resulting problem is a quadratic programming (QP) optimization problem. It can be stated as:

$$\text{minimize } \frac{1}{2}||\boldsymbol{w}||^2 \text{ such that:}$$
$$\boldsymbol{w}^T\boldsymbol{x}_i + b \geq 1, \text{ if } y_i = 1$$
$$\boldsymbol{w}^T\boldsymbol{x}_i + b \leq -1, \text{ if } y_i = -1$$

Which can be written more succinctly as:

$$\text{minimize } \frac{1}{2}||\boldsymbol{w}||^2 \text{ such that:}$$
$$y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) \geq 1\,.$$

The solution to this optimization problem is found by Lagrangian multipliers, but its derivation is not the purpose of this work.

12.2.3 Soft-margin classifiers

In 1995, Corinna Cortes and Vladimir Vapnik suggested a modified maximum margin idea that allows some mislabeled examples. If there exists no hyperplane that can split the "yes" and "no" examples, the Soft Margin method will choose a hyperplane that splits the examples as cleanly as possible, while still maximizing the distance to the nearest correctly classified examples. The method introduces slack variables, ξ_i, which measure the degree of misclassification on the samples $\boldsymbol{x}_i$.

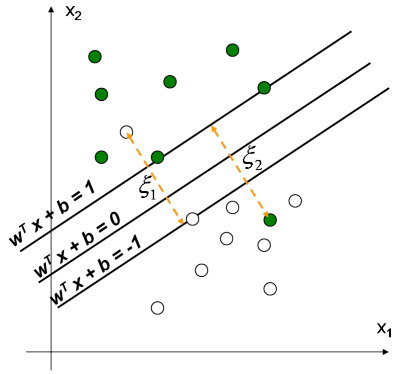


Figure 12.5 – Classification using soft margin

Using the slack variables we define the following optimization problem:

$$\boldsymbol{w}^T\boldsymbol{x}_i + b \geq 1 - \xi_i \text{ if } y_i = 1$$
$$\boldsymbol{w}^T\boldsymbol{x}_i + b \leq -1 + \xi_i \text{ if } y_i = -1$$
$$\xi_i \geq 0$$

Which is equivalent to:

$$\text{minimize } \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{n}\xi_i \text{ such that}$$
$$y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) \geq 1 - \xi_i \text{ and } \xi_i \geq 0.$$

Parameter *C* can be viewed as a tradeoff parameter between error and margin.

12.2.4 Kernel trick

If the data points are not linearly separable a transformation can be applied to each sample $\boldsymbol{x}_i$.which performs a mapping into a higher dimensional space where they are linearly separable. Denoting this transformation by $\boldsymbol{\phi}$ we can write the following optimization problem:

$$\text{minimize } \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{n}\xi_i \text{ such that}$$
$$y_i(\boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x}_i) + b) \geq 1 - \xi_i \text{ and } \xi_i \geq 0.$$

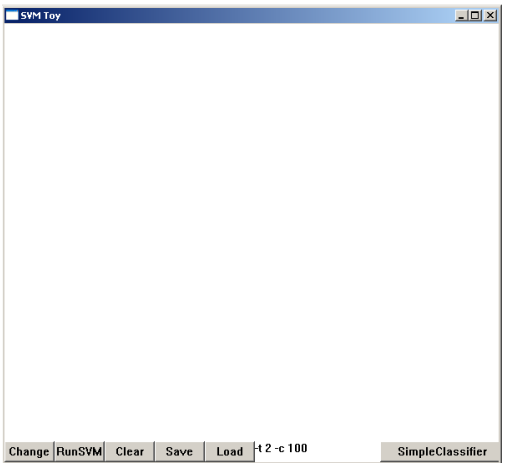
Since the solution for the SVM requires only dot products between instances the usage of the transformation ϕ can be avoided if we define the following kernel function:

$$K(x_i, x_j) = < \phi(x_i), \phi(x_j) >$$

12.3 Practical Work

For the Practical Work you will be given a framework called *SVM-toy* that provides a C++ implementation of soft-margin classifiers using different types of kernels.

1. Download *TestSVM.zip*. Compile *SVM-toy* and run it. Its interface should look like:



The buttons of the interface have the following meaning:

- 'Change' button: the application allows the user to add points in the classification space (the white window) by mouse left click; this button allows to change the color of the points (each color corresponds to a class). A maximum number of three colors is allowed (hence three classes)
- 'RunSVM' button – runs the SVM classifier with the parameters specified in the edit box
- 'Clear' button – clears the classification space
- 'Save' button – saves the points (normalized coordinates) from the classification space to a file
- 'Load' button – loads a bitmap image (loads and draws the points into the classification space)
- The Edit box where parameters are specified, the default values are
  '–t 2 –c 100'

The application allows several parameters, but we will use two of them:

- '-t kernel_type' specifies the kernel type: set type of kernel function (default 2); 'kernel_type' can be one of the following:
  0 – linear kernel: $< \boldsymbol{u}, \boldsymbol{v} >= \boldsymbol{u}^T \boldsymbol{v}$
  1 –polynomial kernel: $(\gamma < \boldsymbol{u}, \boldsymbol{v} > +c)^d$
  2 – radial basis function: exp $(-\gamma |\boldsymbol{u} - \boldsymbol{v}|^2)$
  3 – sigmoid: tanh $(\gamma < \boldsymbol{u}, \boldsymbol{v} > +c)$
- '-c cost' specifies the parameter C from the soft margin classification problem

- 'SimpleClassifier' button – implements the simple classifier.

2. For each image in *svm_images.zip* run the default SVM classifier (with different kernels and costs)
3. Implement the 'SimpleClassifier' code and compare it to the SVM classifier that uses a linear kernel.

   Write the code in the file *svm-toy.cpp* for the case branch:

```
case ID_BUTTON_SIMPLE_CLASSIFIER:
{
/* ************************************
      TO DO:
      WRITE YOUR CODE HERE FOR THE SIMPLE CLASSIFIER
************************************** */
}
```

   For implementing the simple classifier you should know that in the *svm_toy.cpp* file the coordinates of the points are stored in the structure

```
list<point> point_list;
```

   and a point is defined by the structure:

```
struct point {
      double x, y;
      signed char value;
};
```

   The variable 'value' represents the class label.

   The coordinates of the points are normalized between 0 and 1 and the (0,0) point is located in the top left corner.

Notice that the dimension of the classification space is XLEN x YLEN. Hence to a normalized point (x,y) we have other coordinates in the classification space (drawing space) which are (x*XLEN, y*YLEN).

The drawing of a segment between two points is done by the method:

```
DrawLine(window_dc,x1, y1, x2, y2, RGB(255,0,0));
```

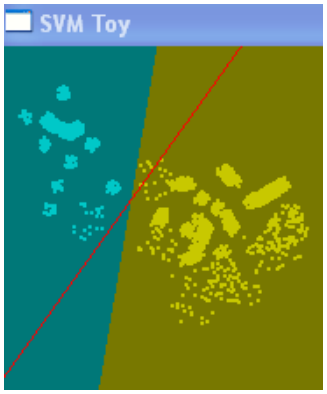
In order to iterate over all the points and count how many points are in class '1' and in class '2' you should do the following:

```
//declare an iterator
list<point>::iterator p;
int nrSamples1=0;
int nrSamples2=0;
double xC1=0,xC2=0,yC1=0,yC2=0;

for(p = point_list.begin(); p != point_list.end(); p++)
{
	if ((*p).value==1) //point from class '1'
	{
		nrSamples1++;
		xC1 =(*p).x;
		//normalized x coordinate of the current point
		yC1 =(*p).y;
		//normalized y coordinate of the current point

	}
	if ((*p).value==2)  //point from class '2'
	{
		nrSamples2++;
		xC2 =(*p).x;
		//normalized x coordinate of the current point
		yC2 =(*p).y;
		//normalized y coordinate of the current point
	}
}
```

12.4 Sample result:



Details:

- 2D points to be classified
- 2 classes, 2 features (x_1 and x_2)
- Red line separation obtained by implementing the 'Simple Classifier' algorithm
- Cyan/Brown line separation obtained by SVM linear kernel (-t 0) and cost C=100 (-c 100)

Observe:

- The maximized margin obtained with SVM
- The points incorrectly classified by simple classifier

12.5 References

[1] J. Shawe-Taylor, N. Cristianini: *Kernel Methods for Pattern Analysis.* Pattern Analysis (Chapter 1)

[2] B. Scholkopf, A. Smola: *Learning with Kernels*. A Tutorial Introduction (Chapter 1), MIT University Press.

[3] LIBSVM: http://www.csie.ntu.edu.tw/~cjlin/libsvm/

REFERENCES

Global references:

1. Richard O. Duda, Peter E. Hart , David G . Stork, "Pattern Clasification", John Wiley and Sons, 2001.

2. Kevin P. Murphy, “Machine Learning: A Probabilistic Perspective”, The MIT Press, 2012

3. Kevin P. Murphy, “Probabilistic Machine Learning: An Introduction”, The MIT Press, 2022

4. C.M. Bishop, “Pattern Recognition and Machine Learning”, second edition, Springer, 2016

5. S. Nedevschi, “Pattern Recognition Systems – Lecture Notes”, TUCN 2022

References per lab works:

1.1 Stanford Machine Learning CS229 - lecture notes 1 – https://see.stanford.edu/materials/aimlcs229/cs229-notes1.pdf

1.2 Tomas Svoboda - Least-squares solution of Homogeneous Equation http://cmp.felk.cvut.cz/cmp/courses/XE33PVR/WS20072008/Lectures/Supporting/constrained_lsq.pdf

2.1 Robert C. Bolles, Martin A. Fischler: *A RANSAC-Based Approach to Model Fitting and Its Application to Finding Cylinders în Range Data*, 1981

2.2 Richard Hartley, Andrew Zisserman: *Multiple View Geometry în Computer Vision,* 2003

3.1 P. Hough, “Method and means for recognizing complex patterns”, US patent 3,069,654, 1962.

3.2 R. O. Duda and P. E. Hart, "Use of the Hough Transformation to Detect Lines and Curves in Pictures," *Comm. ACM*, Vol. 15, pp. 11–15, 1972.

3.3 D. H. Ballard, "Generalizing the Hough Transform to Detect Arbitrary Shapes", *Pattern Recognition*, Vol.13, No.2, p.111-122, 1981.

3.4 https://en.wikipedia.org/wiki/Hough_transform

4.1 Wikipedia The Free Encyclopedia – *Distance Transform*, http://en.wikipedia.org/wiki/Distance_transform

4.2 Compendium of Computer Vision – *Distance Transform*, http://homepages.inf.ed.ac.uk/rbf/HIPR2/distance.htm

5.1 MIT CBCL FACE dataset, http://www.ai.mit.edu/courses/6.899/lectures/faces.tar.gz

6.1 Wikipedia article PCA - https://en.wikipedia.org/wiki/Principal_component_analysis

6.2 Stanford CS229 Machine Learning course notes - https://cs229.stanford.edu/main_notes.pdf

6.3 Lindsay Smith - PCA tutorial - http://faculty.iiit.ac.in/~mkrishna/PrincipalComponents.pdf

6.4 PCA in R (animation of projection) - https://poissonisfish.wordpress.com/2017/01/23/principal-component-analysis-in-r/

7.1 Cluster analysis Wikipedia article - https://en.wikipedia.org/wiki/Cluster_analysis

7.2 K-means Wikipedia article - https://en.wikipedia.org/wiki/K-means_clustering

7.3 Arthur, David, and Sergei Vassilvitskii. "k-means++: The advantages of careful seeding." Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics, 2007.

7.4 P. Arbelaez, M. Maire, C. Fowlkes and J. Malik. „Contour Detection and Hierarchical Image Segmentation", IEEE TPAMI, Vol. 33, No. 5, pp. 898-916, May 2011.

7.5 Image segmentation dataset: https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/resources.html

8.1 Wikipedia article - k-NN classifier https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm

8.2 Andrew Ng - Machine Learning: Nonparametric methods & Instance-based learning http://www.cs.cmu.edu/~epxing/Class/10701-08s/Lecture/lecture2.pdf

9.1 *Data Science Textbook* – https://docs.tibco.com/data-science/textbook

9.2 LeCun, Yann, Corinna Cortes, and Christopher JC Burges. "The MNIST database." URL http://yann. lecun. com/exdb/mnist (1998).

10.1 Rosenblatt, Frank (1957), The Perceptron - a perceiving and recognizing automaton. Report 85-460-1, Cornell Aeronautical Laboratory.

10.2 Richard O. Duda, Peter E. Hart, David G. Stork: Pattern Classification 2nd ed.

10.3 Xiaoli Z. Fern, Machine Learning Course, Oregon University – http://web.engr.oregonstate.edu/~xfern/classes/cs434/slides/perceptron-2.pdf

10.4 Gradient Descent - http://en.wikipedia.org/wiki/Gradient_descent

10.5 Avrim Blum, Machine Learning Theory, Carnegie Mellon University - https://www.cs.cmu.edu/~avrim/ML10/lect0125.pdf

11.1 Robert E. Schapire, The Boosting Approach to Machine Learning, An Overview, 2001

11.2 AdaBoost - https://en.wikipedia.org/wiki/AdaBoost

12.1 J. Shawe-Taylor, N. Cristianini: *Kernel Methods for Pattern Analysis.* Pattern Analysis (Chapter 1)

12.2 B. Scholkopf, A. Smola: *Learning with Kernels*. A Tutorial Introduction (Chapter 1), MIT University Press.

12.3 LIBSVM: http://www.csie.ntu.edu.tw/~cjlin/libsvm/