UNIT - II

Image to Image Mapping

This chapter describes transformations between images and some practical methods for computing them. These transformations are used for warping and image registration. Finally, we look at an example of automatically creating panoramas.

1.HOMOGRAPHIES:

A homography is a 2D projective transformation that maps points in one plane to another. In our case, the planes are images or planar surfaces in 3D. Homographies have many practical uses, such as registering images, rectifying images, texture warping, and creating panoramas. We will make frequent use of them. In essence, a homography H maps 2D points (in homogeneous coordinates) according to

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix} \quad \text{or} \quad \mathbf{x}' = H\mathbf{x}.$$

Homogeneous coordinates are a useful representation for points in image planes (and in 3D, as we will see later). Points in homogeneous coordinates are only defined up to scale so that $\mathbf{x} = [x, y, w] = [\alpha x, \alpha y, \alpha w] = [x/w, y/w, 1]$ all refer to the same 2D point. As a consequence, the homography H is also only defined up to scale and has eight independent degrees of freedom. Often points are normalized with w = 1to have a unique identification of the image coordinates x, y. The extra coordinate makes it easy to represent transformations with a single matrix.

The Direct Linear Transformation Algorithm:

The direct linear transformation (DLT) is an algorithm for computing H given four or more correspondences.

$$\begin{bmatrix} -x_1 & -y_1 & -1 & 0 & 0 & 0 & x_1x_1' & y_1x_1' & x_1' \\ 0 & 0 & 0 & -x_1 & -y_1 & -1 & x_1y_1' & y_1y_1' & y_1' \\ -x_2 & -y_2 & -1 & 0 & 0 & 0 & x_2x_2' & y_2x_2' & x_2' \\ 0 & 0 & 0 & -x_2 & -y_2 & -1 & x_2y_2' & y_2y_2' & y_2' \\ & \vdots & & \vdots & & \vdots & & \vdots & \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \\ h_7 \\ h_8 \\ h_9 \end{bmatrix} = \mathbf{0},$$

or A**h** = **0** where A is a matrix with twice as many rows as correspondences. By stacking all corresponding points, a least squares solution for H can be found using singular value decomposition (SVD).

Affine Transformations:

An affine transformation has six degrees of freedom and therefore three point correspondences are needed to estimate H. Affine transforms can be estimated using the DLT algorithm above by setting the last two elements equal to zero, $h_7 = h_8 = 0$. Here we will use a different approach, (Visualizing the Images on Principal Components). which computes the affine transformation matrix from point correspondences:

```
import numpy as np

# Define source and destination points
src_points = np.array([[1, 1], [2, 3], [4, 5]])
dst_points = np.array([[2, 2], [4, 6], [8, 10]])

# Compute the affine transformation matrix
affine_matrix = compute_affine_matrix(src_points, dst_points)

print(affine_matrix)
```

[[2. 0. 0.]
[0. 2. 0.]
[0. 0. 1.]]

2.WARPING IMAGES:

- A warp can easily be performed with SciPy using the ndimage package.
- The command

transformed_im=ndimage.affine_transform(im,A,b,size)

transforms the image patch im with A a linear transformation and **b** a translation vector as above. The optional argument size can be used to specify the size of the output image. The default is an image with the same size as the original. To see how this works, try running the following commands:

```
from scipy import ndimage
im = array(Image.open('empire.jpg').convert('L'))
H = array([[1.4,0.05,-100],[0.05,1.5,-100],[0,0,1]])
im2 = ndimage.affine_transform(im,H[:2,:2],(H[0,2],H[1,2]))
figure()
gray()
imshow(im2)
show()
```

This gives a result like the image to the right in Figure 3-1. As you can see, missing pixel values in the result image are filled with zeros.

Image in Image

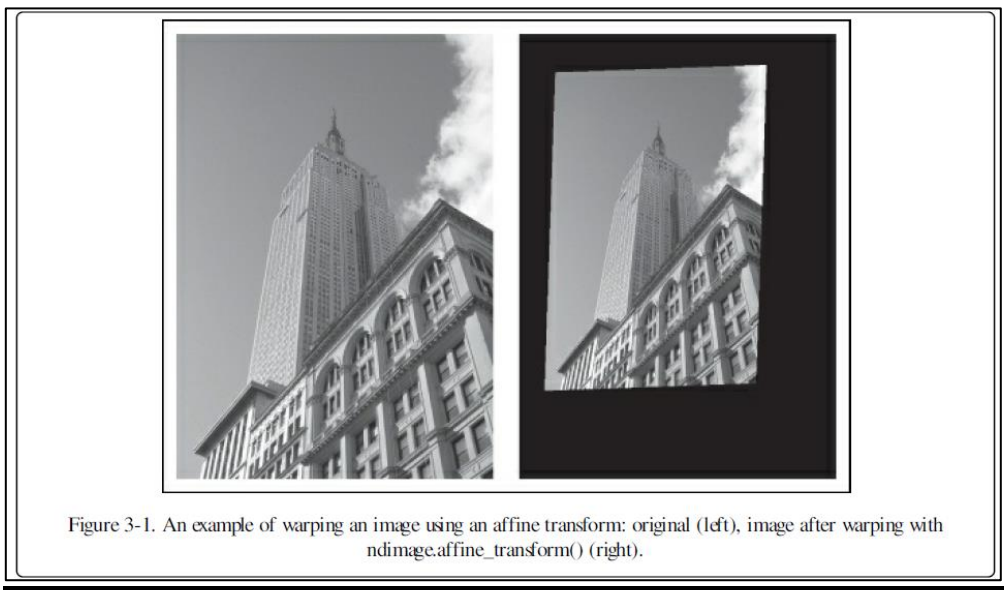
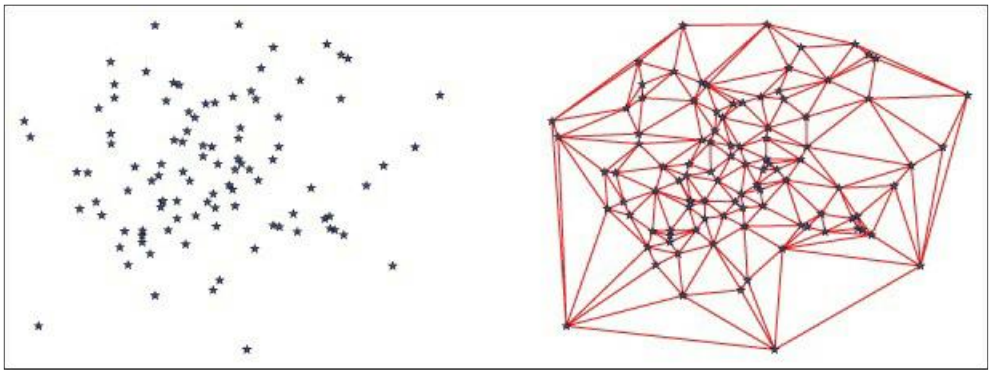


Figure 3-1. An example of warping an image using an affine transform: original (left), image after warping with ndimage.affine_transform() (right).

Piecewise Affine Warping

As we saw in the example above, affine warping of triangle patches can be done to exactly match the corner points.



An example of Delaunay triangulation of a set of random 2D points

Registering Images

Image registration is the process of transferring images so that they are aligned in a common coordinate frame. Registration can be rigid or non-rigid and is an important step in order to be able to do image comparisons and more sophisticated analysis.

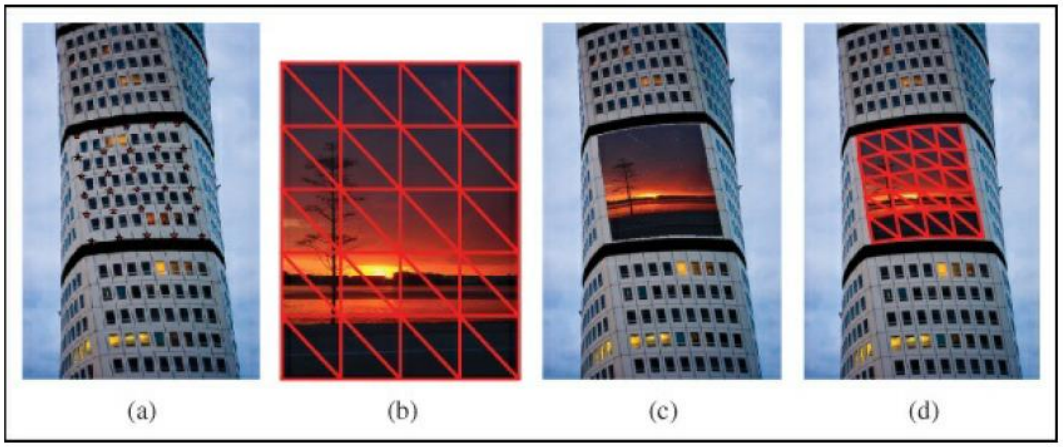


Figure 3-5. An example of piecewise affine warping using Delaunay triangulated landmark points: (a) the target image with landmarks; (b) image with triangulation; (c) with warped image; (d) with warped image and triangulation.

3. CREATION PANORMAS:

Two (or more) images that are taken at the same location (that is, the camera position is the same for the images) are homographically related (see Figure 3-9). This is frequently used for creating panoramic images where several images are stitched together into one big mosaic. In this section we will explore how this is done.

RANSAC

RANSAC, short for "RANdom SAmple Consensus," is an iterative method to fit models to data that can contain outliers.

- Given a model, for example a homography between sets of points, the basic idea is that the data contains inliers,
- the data points that can be described by the model, and outliers, those that do not fit the model.

Figure 3-9. Five images of the main university building in Lund, Sweden. The images are all taken from the same viewpoint.

The standard example is the case of fitting a line to a set of points that contains outliers. Simple least squares fitting will fail, but RANSAC can hopefully single out the inliers and obtain the correct fit.

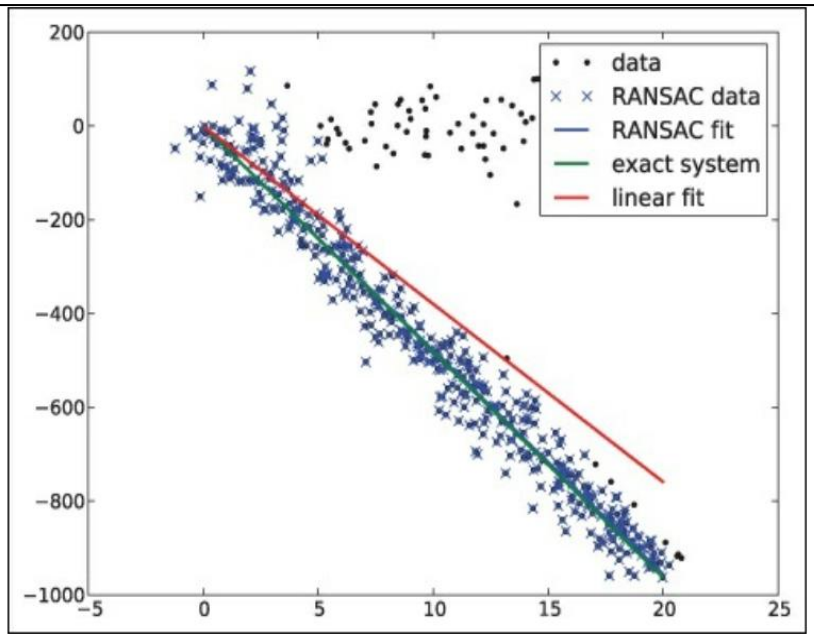


Figure 3-10. An example of using RANSAC to fit a line to points with outliers.

Camera Models and Augmented Reality

Here we show how to determine camera properties and how to use image projections for applications like augmented reality.

4.PINHOLE CAMERA MODEL

The pinhole camera model is a simplified mathematical representation of how light rays pass through a small aperture or hole in a camera to form an image. It is commonly used in computer vision and graphics to simulate the behavior of a camera.

In the pinhole camera model, we consider a camera with a small pin-sized hole (aperture) in the front and an image plane (sensor or film) placed behind it. When an object is placed in front of the camera, light rays from the object pass through the pinhole and form an inverted image on the image plane.

Here are the key components and assumptions of the pinhole camera model:

1. **Pinhole:** The small aperture or hole through which light enters the camera. It acts as a point where all incoming light rays converge.
2. **Image Plane:** The surface where the image is formed. It is positioned perpendicular to the optical axis and located at a certain distance from the pinhole. The size and position of the image on the image plane depend on the object's distance and its projection through the pinhole.
3. **Optical Axis:** The line passing through the pinhole, perpendicular to the image plane. It defines the camera's viewing direction.
4. **Focal Length:** The distance between the pinhole and the image plane. It determines the amount of magnification and affects the field of view. In the pinhole camera model, the focal length is typically assumed to be fixed.
5. **Perspective Projection:** The pinhole camera model assumes perspective projection, where objects that are closer to the camera appear larger in the image, while objects farther away appear smaller. This effect is due to the convergence of light rays through the pinhole.
6. **Inverted Image:** The pinhole camera model assumes that the formed image on the image plane is inverted. This means that the top of an object will appear at the bottom of the image and vice versa.
7. **No Lens Distortions**: The pinhole camera model does not consider lens distortions such as radial or tangential distortions, as it assumes a perfect pinhole without any lens elements.

By using the pinhole camera model, we can estimate various camera parameters and simulate the behavior of a camera system. This model forms the foundation for more complex camera models and computer vision algorithms, such as camera calibration, 3D reconstruction, and image formation.

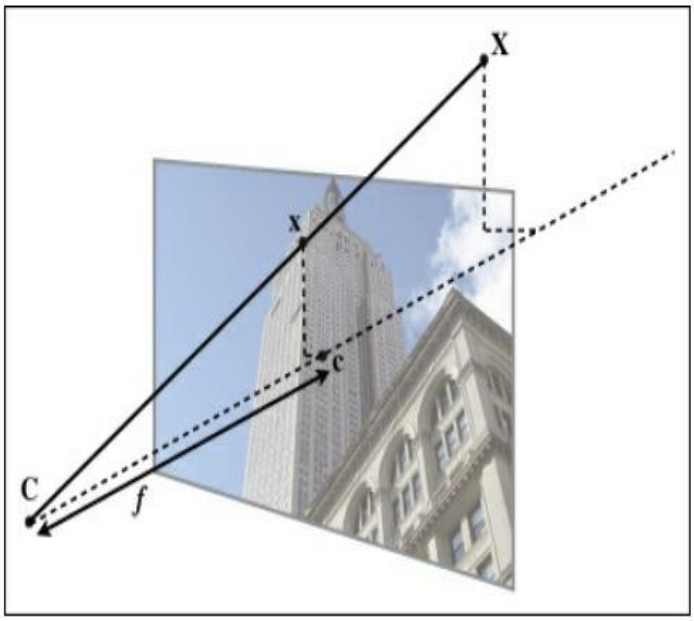


Figure 4-1. The pin-hole camera model. The image point x is at the intersection of the image plane and the line joining the 3D point X and the camera center C. The dashed line is the optical axis of the camera.

With a pin-hole camera, a 3D point **X** is projected to an image point **x** (both expressed in homogeneous coordinates) as

Equation 4-1.

$$\lambda \mathbf{x} = P\mathbf{X}.$$

Here, the 3 × 4 matrix P is called the camera matrix (or projection matrix). Note that the 3D point X has four elements in homogeneous coordinates, X = [X, Y, Z, W]. The scalar λ is the inverse depth of the 3D point

The Camera Matrix

The camera matrix can be decomposed as

Equation 4-2.

$$P = K\,[R \mid \mathbf{t}],$$

where

- R is a rotation matrix describing the orientation of the camera,
- **t** a 3D translation vector describing the position of the camera center,
- intrinsic calibration matrix K describing the projection properties of the camera.
  The calibration matrix depends only on the camera properties and is in a general form written as

$$K = \begin{bmatrix} \alpha f & s & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix}.$$

- The focal length, f, is the distance between the image plane and the camera center.

The skew, s, is only used if the pixel array in the sensor is skewed

Equation 4-3:

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix},$$

where we used the alternative notation fx and fy, with fx = αfy.
The aspect ratio, α is used for non-square pixel elements. It is often safe to assume α = 1

$$K = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix}.$$

5. CAMERA CALIBRATION:

Calibrating a camera means determining the internal camera parameters, in our case the matrix K

A Simple Calibration Method:

1. Measure the sides of your rectangular calibration object. Let's call these dX and dY.
2. Place the camera and the calibration object on a flat surface
3. Measure the distance from the camera to the calibration object. Let's call this dZ.
4. Take a picture and check that the setup is straight meaning that the sides of the calibration object align with the rows and columns of the image.
5. Measure the width and height of the object in pixels. Let's call these dx and dy.

$$f_x = \frac{dx}{dX} dZ, \qquad f_y = \frac{dy}{dY} dZ.$$

Figure 4-3. A simple camera calibration setup: an image of the setup used (left); the image used for the calibration (right). Measuring the width and height of the calibration object in the image and the physical dimensions of the setup is enough to determine the focal length.

For the particular setup in Figure 4-3, the object was measured to be 130 by 185 mm, so dX = 130 and dY = 185. The distance from camera to object was 460 mm, so dZ = 460. You can use any unit of measurement; only the ratios of the measurements matter. Using ginput() to select four points in the image, the width and height in pixels was 722 and 1040. This means that dx = 722 and dy = 1040. Putting these values in the relationship above gives

fx = 2555, fy = 2586.

6. POST ESTIMATION FROM PLANES AND MARKER

Introduction:

In computer vision, the estimation of planes plays a crucial role in various applications such as scene understanding, object detection, tracking, and 3D reconstruction. Estimating planes involves identifying and modeling planar surfaces in images or point cloud data. This topic explores different methods and techniques used to estimate planes in computer vision tasks.

1. Plane Detection:

- Plane hypothesis generation: Various algorithms are used to generate initial plane hypotheses, such as region-based segmentation, superpixel clustering, or random sample consensus (RANSAC).
- Plane fitting: RANSAC or similar algorithms are employed to fit planes to the generated hypotheses by iteratively selecting subsets of points and estimating the parameters of the plane model.
- Model selection: Techniques like model selection criteria or verification methods are utilized to select the best-fit plane models among multiple hypotheses.

2. Plane Segmentation:

- Image-based plane segmentation: This technique involves segmenting planar regions directly from 2D images using color, texture, or gradient-based features.
- Point cloud-based plane segmentation: In 3D point cloud data, plane segmentation can be achieved using methods like region growing, clustering, or graph-cut algorithms.

3. Plane Estimation from Multiple Views:

- Multi-view geometry: Utilizing multiple views of a scene, geometric relationships between points and planes can be exploited to estimate planes more accurately.
- Epipolar geometry: Epipolar constraints and fundamental matrix estimation are used to establish correspondences between points in different views and recover plane parameters.

4. Plane Refinement and Reconstruction:

- Plane refinement: Once planes are estimated, refinement techniques can be applied to improve the accuracy of the estimated plane parameters by minimizing the fitting error or optimizing energy functions.
- 3D plane reconstruction: By integrating depth information from stereo or depth sensors, 3D reconstruction of planes can be performed to obtain their geometric properties and spatial layout.

6. Applications and Challenges:

- Applications: Estimating planes in computer vision has broad applications, including autonomous driving, augmented reality, robot navigation, and indoor mapping.
- Challenges: Challenges in plane estimation include handling occlusions, textureless regions, noise, and accurate estimation of plane parameters in complex scenes with multiple intersecting planes.

Estimating planes in computer vision is a fundamental task with significant implications for scene understanding and 3D reconstruction. By leveraging techniques like plane detection, segmentation, multi-view geometry, and refinement, accurate estimation of planes can be achieved. Further advancements in this area will continue to improve the capabilities of computer vision systems in various real-world applications.

Figure 4-4. Example of computing the projection matrix for a new view using a planar object as marker. Matching image features to an aligned marker gives a homography that can be used to compute the pose of the camera. Template image with a gray square (top left); an image taken from an unknown viewpoint with the same square transformed with the estimated homography (top right); a cube transformed using the estimated camera matrix (bottom).

Now we have a homography that maps points on the marker (in this case the book) in one image to their corresponding locations in the other image. Let's define our 3D coordinate system so that the marker lies in the X-Y plane (Z = 0) with the origin somewhere on the marker.
To check our results, we will need some simple 3D object placed on the marker.

Here we will use a cube and generate the cube points using the function:

```
def cube_points(c,wid):
""" Creates a list of points for plotting
a cube with plot. (the first 5 points are
the bottom square, some sides repeated). """
p = []
# bottom
p.append([c[0]-wid,c[1]-wid,c[2]-wid])
p.append([c[0]-wid,c[1]+wid,c[2]-wid])
p.append([c[0]+wid,c[1]+wid,c[2]-wid])
p.append([c[0]+wid,c[1]-wid,c[2]-wid])
p.append([c[0]-wid,c[1]-wid,c[2]-wid]) # same as first to close plot
# top
p.append([c[0]-wid,c[1]-wid,c[2]+wid])
p.append([c[0]-wid,c[1]+wid,c[2]+wid])
p.append([c[0]+wid,c[1]+wid,c[2]+wid])
p.append([c[0]+wid,c[1]-wid,c[2]+wid])
p.append([c[0]-wid,c[1]-wid,c[2]+wid]) # same as first to close plot
# vertical sides
p.append([c[0]-wid,c[1]-wid,c[2]+wid])
p.append([c[0]-wid,c[1]+wid,c[2]+wid])
p.append([c[0]-wid,c[1]+wid,c[2]-wid])
p.append([c[0]+wid,c[1]+wid,c[2]-wid])
p.append([c[0]+wid,c[1]+wid,c[2]+wid])
p.append([c[0]+wid,c[1]-wid,c[2]+wid])
p.append([c[0]+wid,c[1]-wid,c[2]-wid])
return array(p).T
```

Some points are reoccurring so that plot() will generate a nice-looking cube.

$$P_1 = K \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \end{bmatrix},$$

To compose P1 with H as a camera matrix for the second image (P2 = HP1), and ensure that points on the marker plane Z = 0 are transformed correctly, follow these steps:

1. Calculate the first two columns and the fourth column of P2 as they are correct:

P2 = np.dot(H, P1)

2. Recover the third column of P2:

K_inv = np.linalg.inv(K) # Inverse of the calibration matrix

P2[:, 2] = np.cross(P2[:, 0], P2[:, 1])

P2[:, 2] = np.dot(K_inv, P2[:, 2])

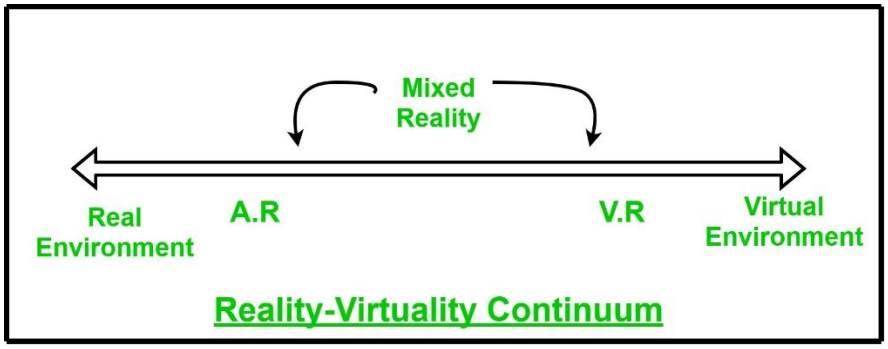
Explanation:

- K represents the calibration matrix, which contains the intrinsic parameters of the camera.
- K_inv is the inverse of the calibration matrix.
- The third column of P2 can be recovered by taking the cross product of the first two columns of P2 and then multiplying it with the inverse calibration matrix to ensure the correct scaling.

3. Now, P2 is the composed camera matrix for the second image, with correctly transformed points on the marker plane.

Note: Ensure that the matrices P1, H, and K are appropriately defined before performing the above operations.

7. AUGMENTED REALITY:



Augmented Reality (AR) combines virtual and real-world elements to enhance user perception and interaction. In computer vision, AR integrates computer-generated content into the user's view of the real environment. It finds applications in entertainment, education, gaming, advertising, and industry.

1. **Marker-based AR:** Detect and track visual markers like QR codes. Estimate marker pose for accurate alignment of virtual content. Render and align 3D models/images with the marker's pose.
2. **Markerless AR:** Detect and track distinctive features (corners, edges) in the real environment. Use SLAM to create a map and track camera position. Handle occlusion using depth sensors or estimation techniques.
3. **Projection-based AR:** Project virtual content onto real-world surfaces. Detect and map surfaces accurately. Calibration ensures correct alignment and surface following.
4. **Applications of AR:** Enhance gaming, entertainment, education, and training experiences. Aid industrial operations, maintenance, and product visualization. Create interactive marketing campaigns. Assist in surgical planning, medical imaging, and patient education.

Example:

PyGame and PyOpenGL

AR in computer vision seamlessly integrates virtual content into the real world, transforming user experiences. Marker-based, markerless, and projection-based approaches continue to advance, revolutionizing interaction with the environment.

UNIT-III

1.Epipolar geometry:

- The study of the relationship between two views of the same scene, as seen from two different cameras. It provides a mathematical framework for understanding how the cameras are related to each other and how they can be used to reconstruct the 3D structure of the scene. These geometric relationships are described by is called epipolar geometry.
- The main concept in epipolar geometry is the epipolar plane, which is the plane containing the two camera centers and any point in the scene. The epipolar plane intersects each image plane in a line, called an epipolar line. The epipolar line corresponds to the projection of a 3D point in the scene onto the other camera's image plane.
- The epipolar lines have a special property known as the epipolar constraint, which states that the corresponding points in two views must lie on the same epipolar line. This constraint is used to reduce the search space for finding corresponding points between two images, making stereo vision more efficient.
- Without any prior knowledge of the cameras, there is an inherent ambiguity in that a 3D point, **X**, transformed with an arbitrary (4 × 4) homography H as H**X** will have the same image point in a camera PH power -1 as the original point in the camera P.

- Expressed with the camera equation

$$\lambda \mathbf{x} = P\mathbf{X} = PH^{-1}H\mathbf{X} = \hat{P}\hat{\mathbf{X}}.$$

A good choice is to set the origin and coordinate axis to align with the first camera so that

$$P_1 = K_1[I \mid 0] \text{ and } P_2 = K_2[R \mid \mathbf{t}].$$

- K1 and K2 are the calibration matrices
- R is the rotation of the second camera
- **t** is the translation of the second camera.

Using these camera matrices, one can derive a condition for the projection of a point **X** to image points **x**1 and **x**2 (with P1 and P2, respectively).

- The following equation must be satisfied:

Equation 5-1.

$$\mathbf{x}_2^T F \mathbf{x}_1 = 0,$$

where

$$F = K_2^{-T} S_{\mathbf{t}} R K_1^{-1}$$

And the matrix S**t** is the skew symmetric matrix

Equation 5-2.

$$S_{\mathbf{t}} = \begin{bmatrix} 0 & -t_3 & t_2 \\ t_3 & 0 & -t_1 \\ -t_2 & t_1 & 0 \end{bmatrix}.$$

Equation (Equation 5-1) is called the epipolar constraint. The matrix F in the epipolar constraint is called the fundamental matrix

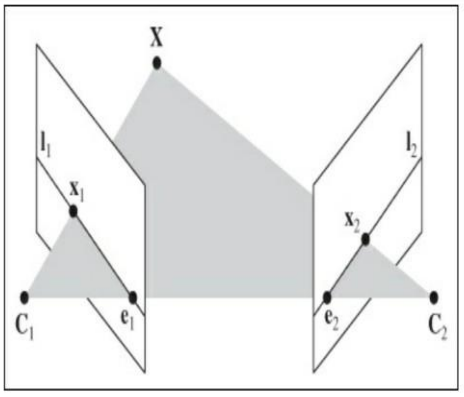


Figure 5-1. An illustration of epipolar geometry. A 3D point X is projected to x_1 and x_2, in the two views, respectively. The baseline between the two camera centers, C_1 and C_2, intersect the image planes in the epipoles, e_1 and e_2. The lines l_1 and l_2 are called epipolar lines.

2.Computing with Cameras and 3D Structure:

Here we briefly explain the tools we need for computing with cameras and 3D structure

- **Triangulation:**
  Given known camera matrices, **a set of point correspondences** can be triangulated to recover the 3D positions of these points. The basic algorithm is fairly simple.

  For two views with camera matrices P1 and P2, each with a projection x1 and x2 of the same 3D point X the camera equation gives the relation

$$\begin{bmatrix} P_1 & -\mathbf{x}_1 & 0 \\ P_2 & 0 & -\mathbf{x}_2 \end{bmatrix} \begin{bmatrix} \mathbf{X} \\ \lambda_1 \\ \lambda_2 \end{bmatrix} = 0$$

Add the following function that computes the least squares triangulation of a point pair to sfm.py:

```
def triangulate_point(x1,x2,P1,P2):
  """ Point pair triangulation from
    least squares solution. """

  M = zeros((6,6))
  M[:3,:4] = P1
  M[3:,:4] = P2
  M[:3,4] = -x1
  M[3:,5] = -x2

  U,S,V = linalg.svd(M)
  X = V[-1,:4]

  return X / X[3]
```

Computing the Camera Matrix from 3D Points

- With known 3D points and their image projections, the camera matrix, P, can be computed using a direct linear transform approach.
- This is essentially the inverse problem to triangulation and is sometimes called camera resectioning.

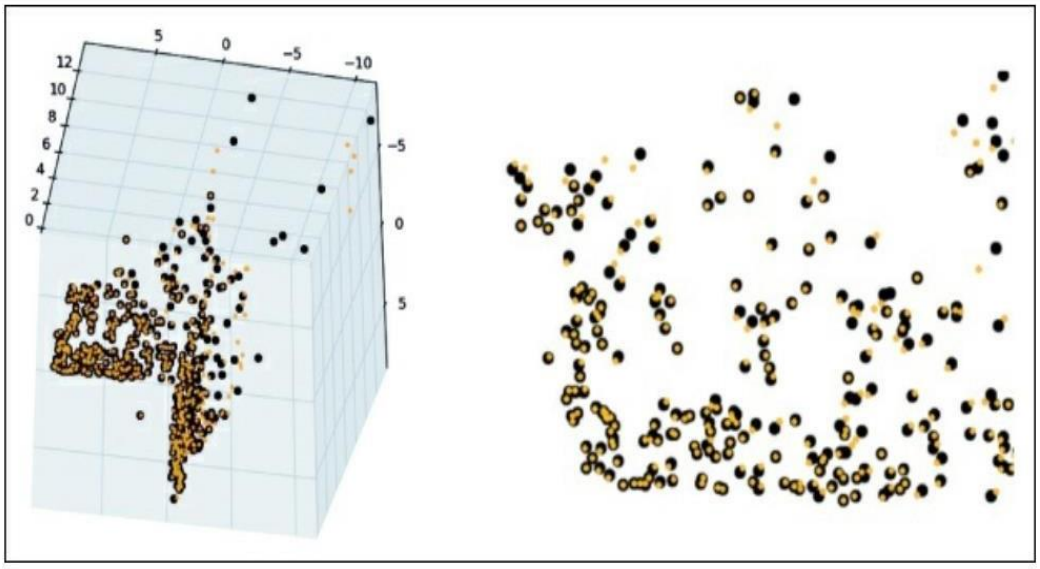
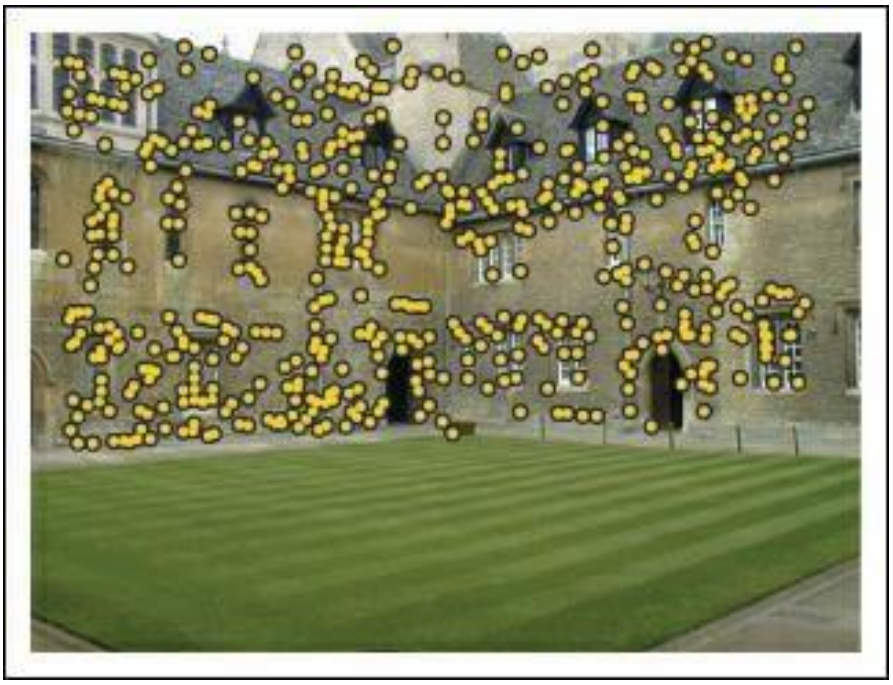


Figure 5-5. Triangulated points using camera matrices and point correspondences. The estimated points are shown with dark circles and the true points with light dots. View from above and to the side (left). Closeup of the points from one of the building walls (right).

From the camera equation (Equation 4-1), each visible 3D point $\mathbf{X}_i$ (in homogeneous coordinates) is projected to an image point $\mathbf{x}_i = [x_i, y_i, 1]$ as $\lambda_i \mathbf{x}_i = \mathrm{P}\,\mathbf{X}_i$ and the corresponding points satisfy the relation

$$\begin{bmatrix} \mathbf{X}_1^T & 0 & 0 & -x_1 & 0 & 0 & \dots \\ 0 & \mathbf{X}_1^T & 0 & -y_1 & 0 & 0 & \dots \\ 0 & 0 & \mathbf{X}_1^T & -1 & 0 & 0 & \dots \\ \mathbf{X}_2^T & 0 & 0 & 0 & -x_2 & 0 & \dots \\ 0 & \mathbf{X}_2^T & 0 & 0 & -y_2 & 0 & \dots \\ 0 & 0 & \mathbf{X}_2^T & 0 & -1 & 0 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \end{bmatrix} \begin{bmatrix} \mathbf{p}_1^T \\ \mathbf{p}_2^T \\ \mathbf{p}_3^T \\ \lambda_1 \\ \lambda_2 \\ \vdots \end{bmatrix} = 0,$$

where $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{p}_3$ are the three rows of P. This can be written more compactly as $\mathrm{M}\mathbf{v} = 0$.



Projected points in view 1 computed using an estimated camera matrix

The 3D points are projected using the estimated camera and plotted. The result looks like Figure with the true points shown as circles and the estimated camera projection as dots.

3. Multiple View Reconstruction:

Computing a 3D reconstruction like this is usually referred to as structure from motion (SfM) since the motion of a camera (or cameras) gives you 3D structure.

Assuming the camera has been calibrated, the steps are as follows:

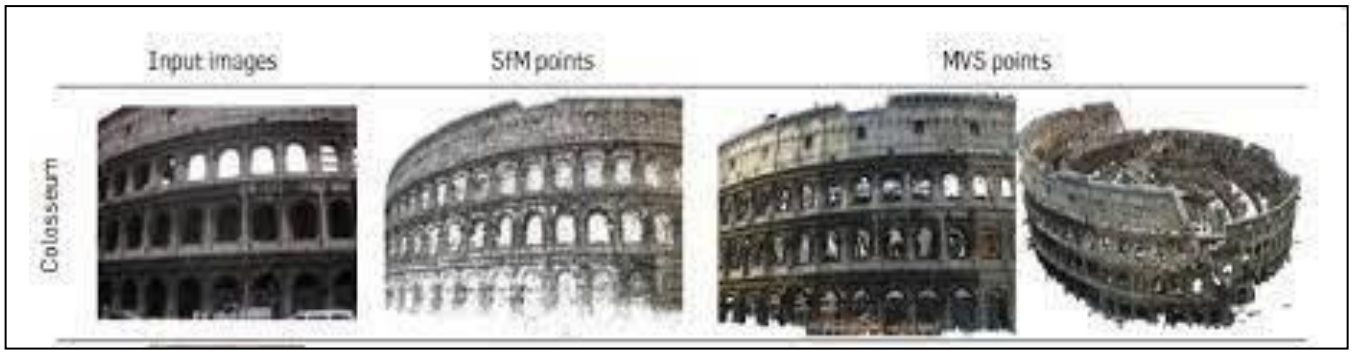
1. Detect feature points and match them between the two images.
2. Compute the fundamental matrix from the matches.
3. Compute the camera matrices from the fundamental matrix.
4. Triangulate the 3D points

3D Reconstruction Example:

```
import homography
import sfm
import sift

# calibration
K = array([[2394,0,932],[0,2398,628],[0,0,1]])


# load images and compute features
im1 = array(Image.open('alcatraz1.jpg'))
sift.process_image('alcatraz1.jpg','im1.sift')
l1,d1 = sift.read_features_from_file('im1.sift')

im2 = array(Image.open('alcatraz2.jpg'))
sift.process_image('alcatraz2.jpg','im2.sift')
l2,d2 = sift.read_features_from_file('im2.sift')


# match features
matches = sift.match_twosided(d1,d2)
ndx = matches.nonzero()[0]

# make homogeneous and normalize with inv(K)
x1 = homography.make_homog(l1[ndx,:2].T)
ndx2 = [int(matches[i]) for i in ndx]
x2 = homography.make_homog(l2[ndx2,:2].T)

x1n = dot(inv(K),x1)
x2n = dot(inv(K),x2)


# estimate E with RANSAC
model = sfm.RansacModel()
E,inliers = sfm.F_from_ransac(x1n,x2n,model)

# compute camera matrices (P2 will be list of four solutions)
P1 = array([[1,0,0,0],[0,1,0,0],[0,0,1,0]])
P2 = sfm.compute_P_from_essential(E)
```

The calibration is known, so here we just hardcode the K matrix at the beginning. As in earlier examples, we pick out the points that belong to matches. After that, we normalize them with K–1 and run the RANSAC estimation with the normalized eight-point algorithm. Since the points are normalized, this gives us an essential matrix. We make sure to keep the index of the inliers, as we will need them. From the essential matrix we compute the four possible solutions of the second camera matrix.

Eg:



4.Stereo Images:

A special case of multi-view imaging is stereo vision (or stereo imaging), **where two cameras are observing the same scene with only a horizontal (sideways) displacement between the cameras.** When the cameras are configured so that the two images have the same image plane with the image rows vertically aligned, the image pair is said to be rectified. This is common in robotics, and such a setup is often called a stereo rig.

Once a corresponding point is found, its depth (Z coordinate) can be computed directly from the horizontal displacement as it is inversely proportional to the displacement

$$Z = \frac{f\,b}{x_l - x_r},$$

- where f is the rectified image focal length.
- b is the distance between the camera centers.
- xl and xr the x-coordinate of the corresponding point in the left and right image.
- The distance separating the camera centers is called the baseline.

Stereo reconstruction (sometimes called dense depth reconstruction) is the problem of recovering a depth map (or, inversely, a disparity map) where the depth (or disparity) for each pixel in the image is estimated. This is a classic problem in computer vision and there are many algorithms for solving it. The Middlebury Stereo Vision Page contains a constantly updated evaluation of the best algorithms with code and descriptions of many implementations. In the next section, we will implement a stereo reconstruction algorithm based on normalized cross-correlation.

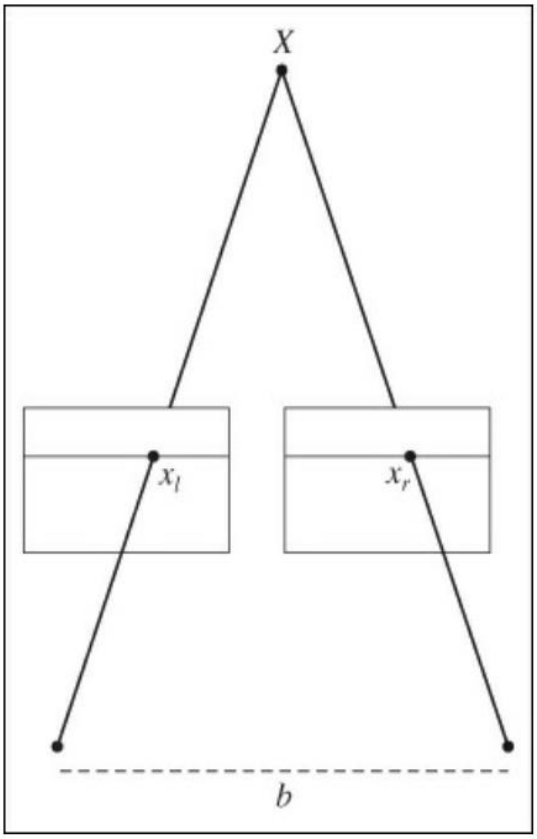


Figure 5-9. An illustration of a rectified stereo image setup where corresponding points are on the same row in both images.

Clustering Images:

- Clustering can be used for recognition, for dividing data sets of images, and for organization and navigation. We also look at using clustering for visualizingsimilarity between images.

5.K-Means Clustering:

K-means is a very simple clustering algorithm that tries to partition the input data in k clusters.

K-means works by iteratively refining an initial estimate of class centroids as follows:

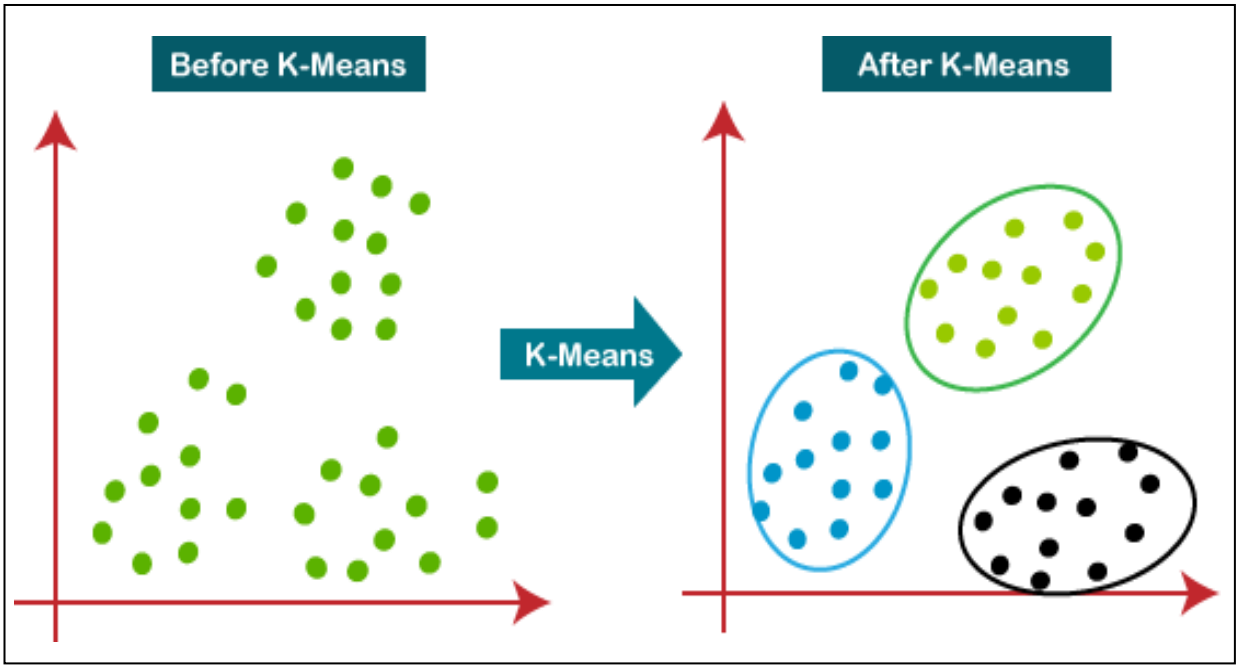
1. Initialize centroids **μi, i = 1 . . . k,** randomly or with some guess.
2. Assign each data point to the class **ci** of its nearest centroid
3. Update the centroids as the average of all data points assigned to that class.
4. Repeat 2 and 3 until convergence.

K-means tries to minimize the total within-class variance

$$V = \sum_{i=1}^{k} \sum_{\mathbf{x}_j \in c_i} (\mathbf{x}_j - \boldsymbol{\mu}_i)^2$$

where **x**j are the data vectors.

The algorithm above is a heuristic refinement algorithm that works fine for most cases, but it does not guarantee that the best solution is found. To avoid the effects of choosing a bad centroid initialization, the algorithm is often run several times with different initialization centroids. Then the solution with lowest variance V is selected. The main drawback of this algorithm is that the number of clusters needs to be decided beforehand, and an inappropriate choice will give poor clustering results. The benefits are that it is simple to implement, it is parallelizable, and it works well for a large range of problems without any need for tuning.



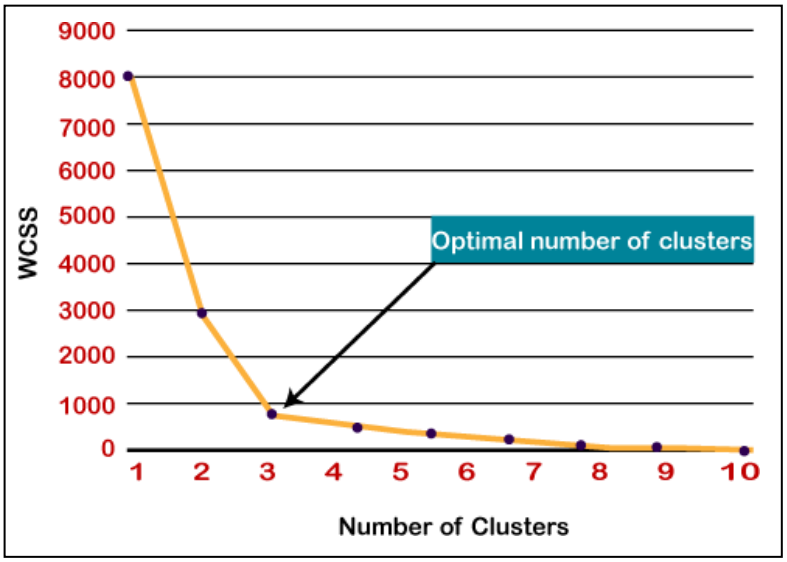
How to choose the value of "K number of clusters" in K-means Clustering?

Elbow Method:

The Elbow method is one of the most popular ways to find the optimal number of clusters. This method uses the concept of WCSS value. **WCSS** stands for **Within Cluster Sum of Squares**, which defines the total variations within a cluster. The formula to calculate the value of WCSS (for 3 clusters) is given below:

$$\mathbf{WCSS} = \sum_{\text{Pi in Cluster1}} \mathbf{distance}\ (P_i\ C_1)^2 + \sum_{\text{Pi in Cluster2}} \mathbf{distance}\ (P_i\ C_2)^2 + \sum_{\text{Pi in CLuster3}} \mathbf{distance}\ (P_i\ C_3)^2$$

In the above formula of WCSS, $\sum_{\text{Pi in Cluster1}}$ distance $(P_i\ C_1)^2$: It is the sum of the square of the distances between each data point and its centroid within a cluster1 and the same for the other two terms.



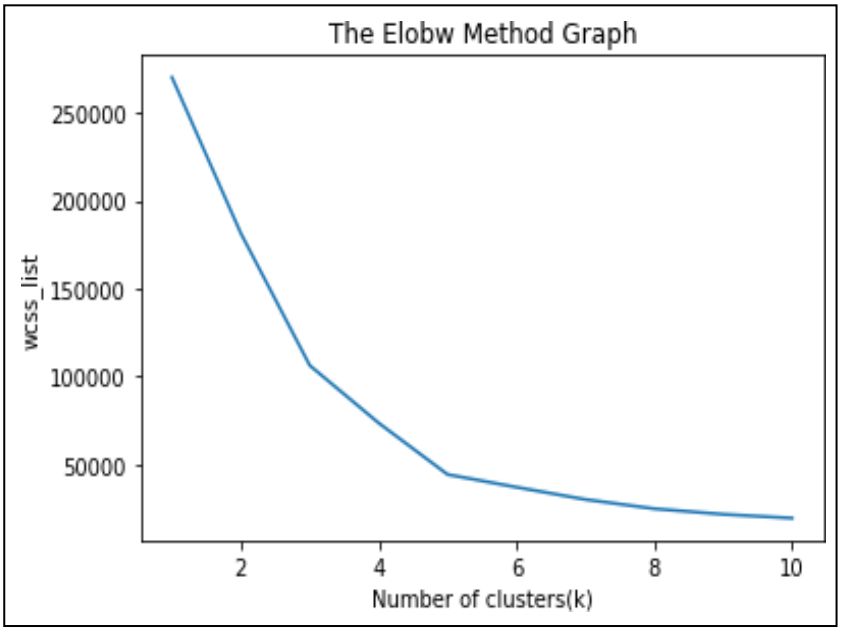
Python Implementation of K-means Clustering Algorithm

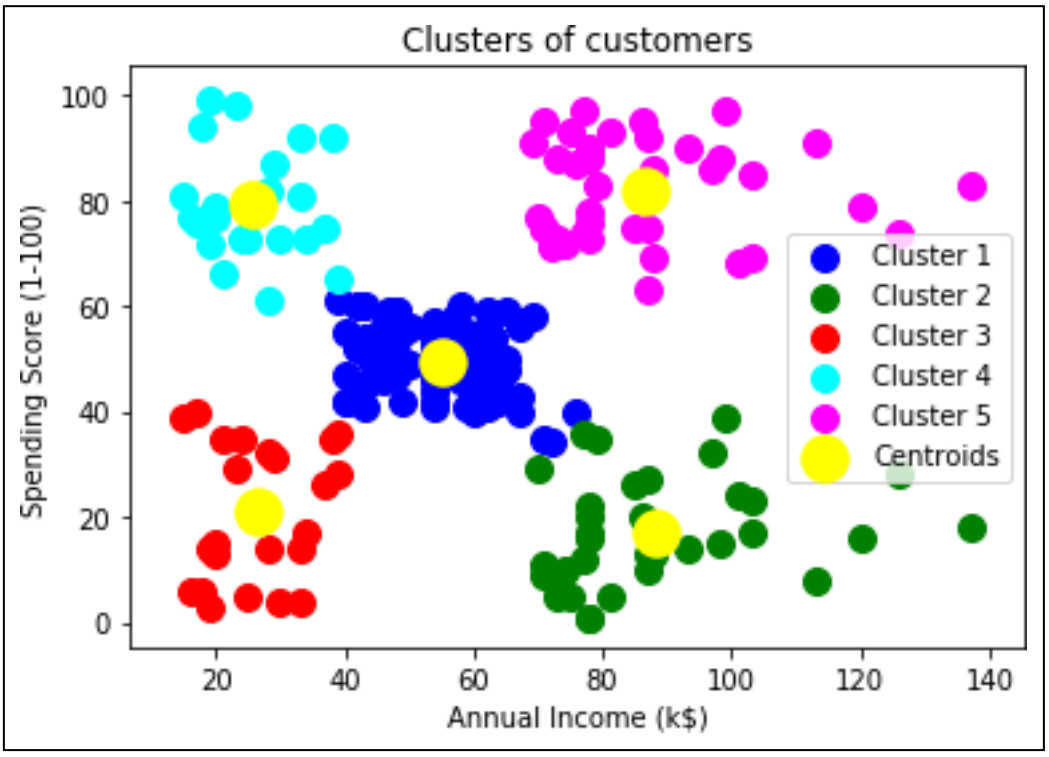
The steps to be followed for the implementation are given below:

- Data Pre-processing
- Finding the optimal number of clusters using the elbow method
- Training the K-means algorithm on the training dataset
- Visualizing the clusters

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.cluster import KMeans

# Load the dataset
dataset = pd.read_csv('Mall_Customers_data.csv')
x = dataset.iloc[:, [3, 4]].values

# Find the optimal number of clusters using the elbow method
wcss_list = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters=i, init='k-means++', random_state=42)
    kmeans.fit(x)
    wcss_list.append(kmeans.inertia_)
```

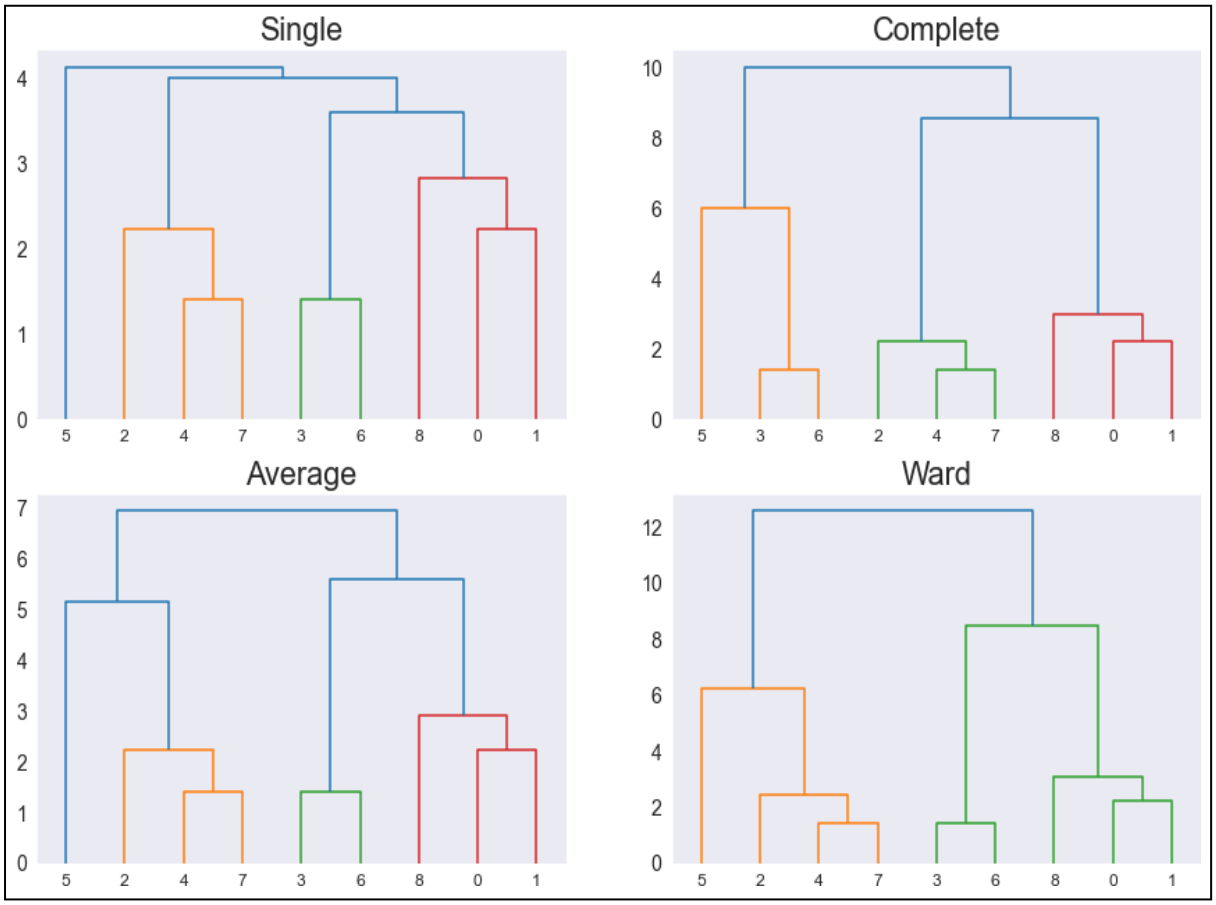
```
# Plot the elbow method graph
plt.plot(range(1, 11), wcss_list)
plt.title('The Elbow Method Graph')
plt.xlabel('Number of clusters (k)')
plt.ylabel('WCSS')
plt.show()
```

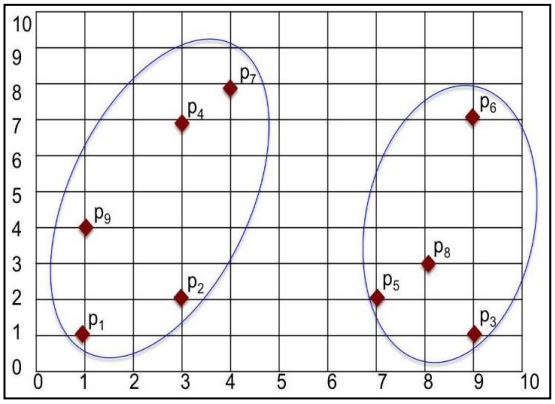




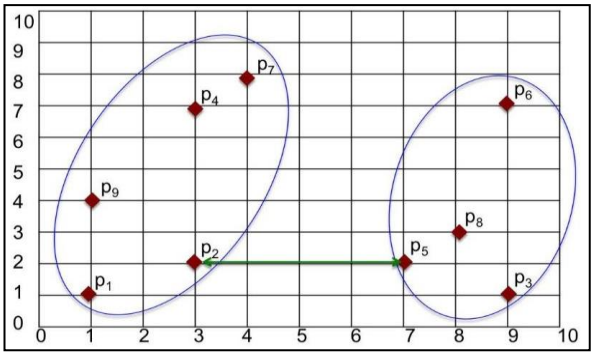
6. *Hierarchical Clustering:*

- Hierarchical clustering (or agglomerative clustering) is another simple but powerful clustering algorithm.
- The idea is to build a similarity tree based on pairwise distances.
- The algorithm starts with grouping the two closest objects and creates an "average" node in a tree with the two objects as children.
- Then the next closest pair is found among the remaining objects but then also including any average nodes, and so on.
- At each node, the distance between the two children is also stored.
- Clusters can then be extracted by traversing this tree and stopping at nodes with distance smaller than some threshold that then determines the cluster size.

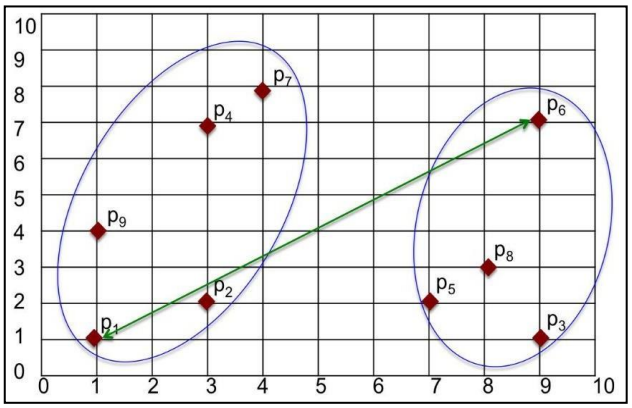




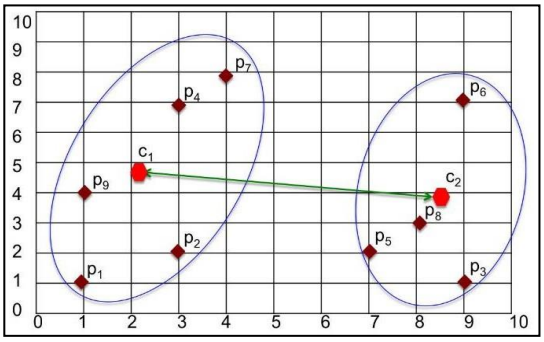
Min (Single) Linkage:



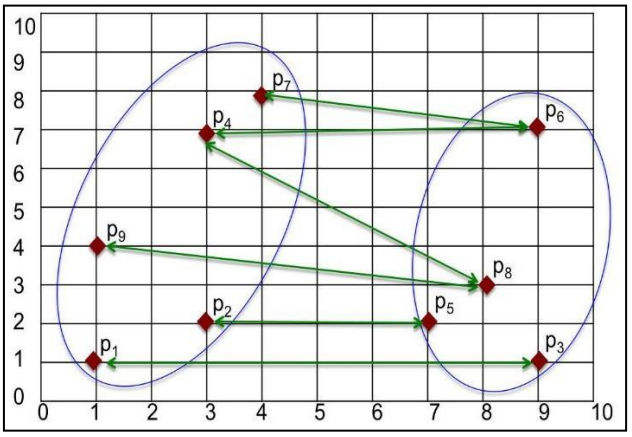
Max (Complete) Linkage:



Centroid Linkage:



Average Linkage:



Steps for implementation of AHC using Python:

The steps for implementation will be the same as the k-means clustering, except for some changes such as the method to find the number of clusters. Below are the steps:

1. Data Pre-processing
2. Finding the optimal number of clusters using the Dendrogram
3. Training the hierarchical clustering model
4. Visualizing the clusters

Implementation in python:

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style('dark')
```

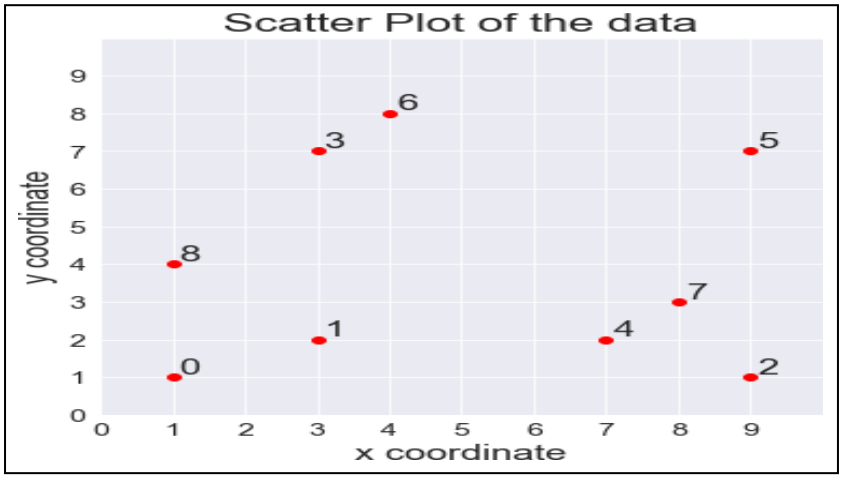
- Next, we'll define a small sample data set:

```
X1 = np.array([[1,1], [3,2], [9,1], [3,7], [7,2],[9,7], [4,8], [8,3],[1,4]])
```

- Let's graph this data set as a scatter plot:

```
plt.figure(figsize=(6, 6))
plt.scatter(X1[:,0], X1[:,1], c='r')
#Create numbered labels for each point
for i in range(X1.shape[0]):
```

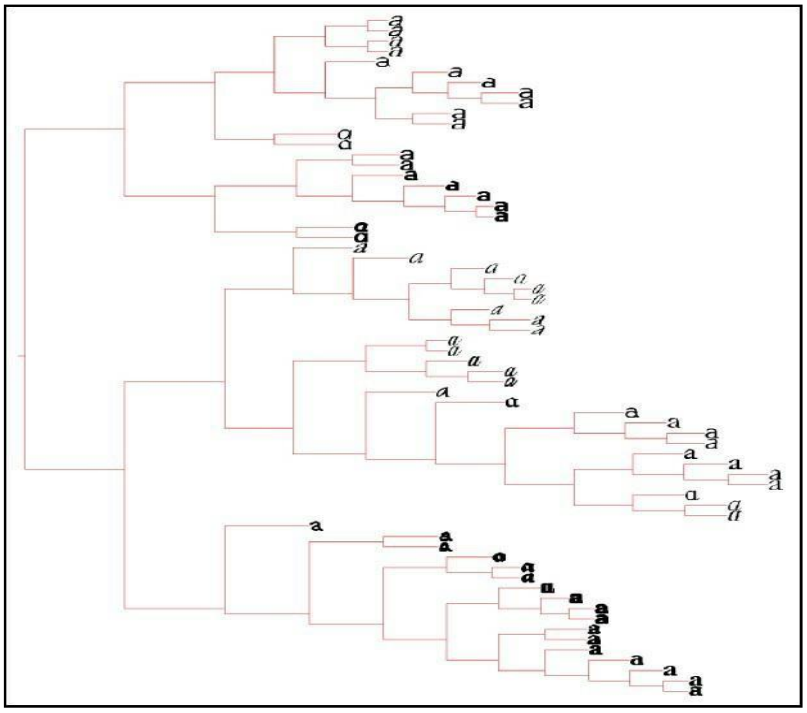
```
        plt.annotate(str(i),xy=(X1[i,0],X1[i,1]),xytext=(3,3),
        textcoords='offset points')
plt.xlabel('x coordinate')
plt.ylabel('y coordinate')
plt.title('Scatter Plot of the data')
plt.xlim([0,10]), plt.ylim([0,10])
plt.xticks(range(10)), plt.yticks(range(10))
plt.grid()
plt.show()
```



7.Spectral Clustering:

Spectral clustering methods are an interesting type of clustering algorithm that have a different approach compared to k-means and hierarchical clustering.

A similarity matrix (or affinity matrix, or sometimes distance matrix) for n elements (for example images) is an n × n matrix with pair-wise similarity scores. Spectral clustering gets its name from the use of the spectrum of a matrix constructed from a similarity matrix. The eigenvectors of this matrix are used for dimensionality reduction and then clustering.



An example of hierarchical clustering of 66 selected font images using 40 principal components as feature vector.

Algorithm:

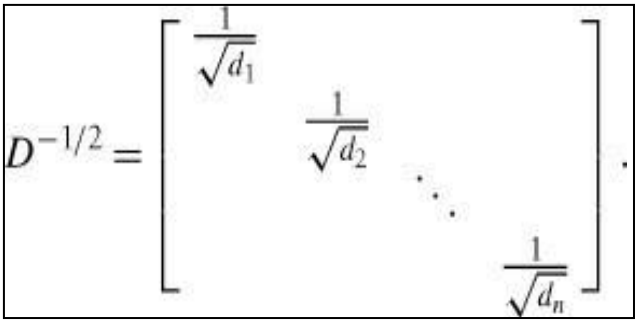
The algorithm can be broken down into 4 basic steps.

1. Construct a similarity graph.
2. Determine the Adjacency matrix W, Degree matrix D and the Laplacian matrix L.
3. Compute the eigenvectors of the matrix L.
4. Using the second smallest eigenvector as input, train a k-means model and use it to classify the data.

Here's how it works. Given a n × n similarity matrix S with similarity scores sij, we can create a matrix, called the Laplacian matrix,

$$L = I - D^{-1/2}SD^{-1/2},$$

where I is the identity matrix and D is the diagonal matrix containing the row sums of S, D = diag(di), di = Σjsij. The matrix $D^{-1/2}$ used in the construction of the Laplacian matrix is then In



Spectral Clustering is a type of clustering algorithm in machine learning that uses eigenvectors of a similarity matrix to divide a set of data points into clusters. The basic idea behind spectral clustering is to use the eigenvectors of the Laplacian matrix of a graph to represent the data points and find clusters by applying k-means or another clustering algorithm to the eigenvectors.

Advantages of Spectral Clustering:

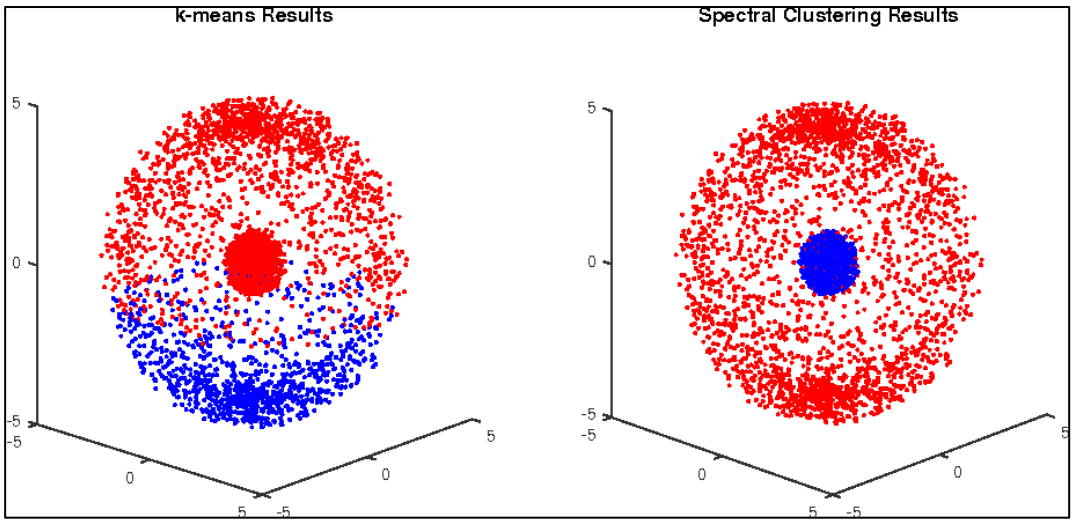
- Scalability: Spectral clustering can handle large datasets and high-dimensional data, as it reduces the dimensionality of the data before clustering.
- Flexibility: Spectral clustering can be applied to non-linearly separable data, as it does not rely on traditional distance-based clustering methods.
- Robustness: Spectral clustering can be more robust to noise and outliers in the data, as it considers the global structure of the data, rather than just local distances between data points.

Disadvantages of Spectral Clustering:

- Complexity: Spectral clustering can be computationally expensive, especially for large datasets, as it requires the calculation of eigenvectors and eigenvalues.
- Model selection: Choosing the right number of clusters and the right similarity matrix can be challenging and may require expert knowledge or trial and error.

Python implementation:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import SpectralClustering
from sklearn.datasets import make_blobs

# Generate sample data
X, y = make_blobs(n_samples=200, centers=3, random_state=42)

# Perform spectral clustering
spectral = SpectralClustering(n_clusters=3, eigen_solver='arpack',
affinity='nearest_neighbors')
labels = spectral.fit_predict(X)

# Plot the clustering result
plt.scatter(X[:, 0], X[:, 1], c=labels)
plt.title('Spectral Clustering')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```



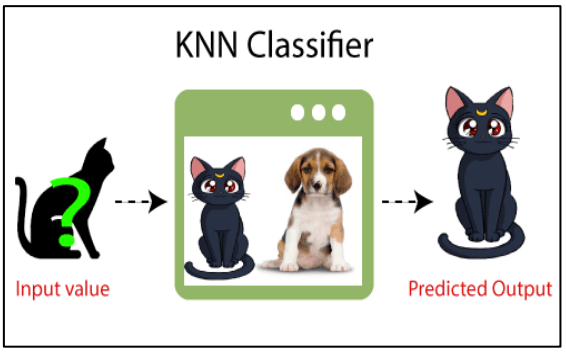
UNIT – IV

Classifying Image Content

This chapter introduces algorithms for classifying images and image content. We look at some simple but effective methods as well as state-of-the-art classifiers and apply them to two-class and multi-class problems. We show examples with applications in gesture recognition and object recognition.

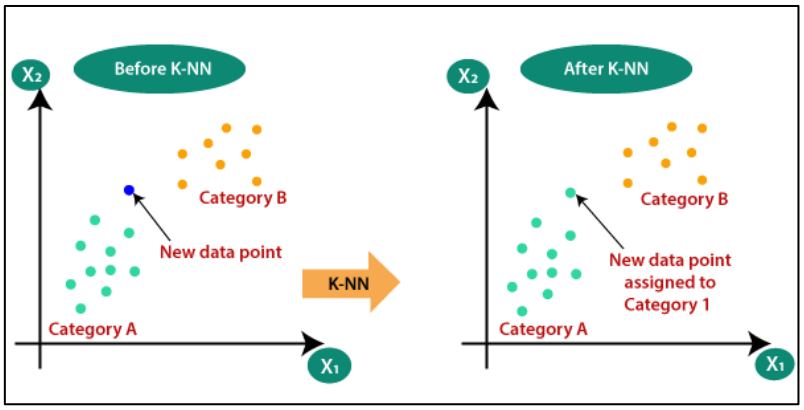
1. K-NEAREST NEIGHBORS:

- K-Nearest Neighbour is one of the simplest Machine Learning algorithms based on Supervised Learning technique.
- K-NN algorithm assumes the similarity between the new case/data and available cases and put the new case into the category that is most similar to the available categories.
- K-NN algorithm stores all the available data and classifies a new data point based on the similarity. This means when new data appears then it can be easily classified into a well suite category by using K- NN algorithm.
- K-NN algorithm can be used for Regression as well as for Classification but mostly it is used for the Classification problems.
- K-NN is a **non-parametric algorithm**, which means it does not make any assumption on underlying data.
- It is also called a **lazy learner algorithm** because it does not learn from the training set immediately instead it stores the dataset and at the time of classification, it performs an action on the dataset.
- KNN algorithm at the training phase just stores the dataset and when it gets new data, then it classifies that data into a category that is much similar to the new data.
- **Example:** Suppose, we have an image of a creature that looks similar to cat and dog, but we want to know either it is a cat or dog. So for this identification, we can use the KNN algorithm, as it works on a similarity measure. Our KNN model will find the similar features of the new data set to the cats and dogs images and based on the most similar features it will put it in either cat or dog category.



Why do we need a K-NN Algorithm?

Suppose there are two categories, i.e., Category A and Category B, and we have a new data point x1, so this data point will lie in which of these categories. To solve this type of problem, we need a K-NN algorithm. With the help of K-NN, we can easily identify the category or class of a particular dataset. Consider the below diagram:

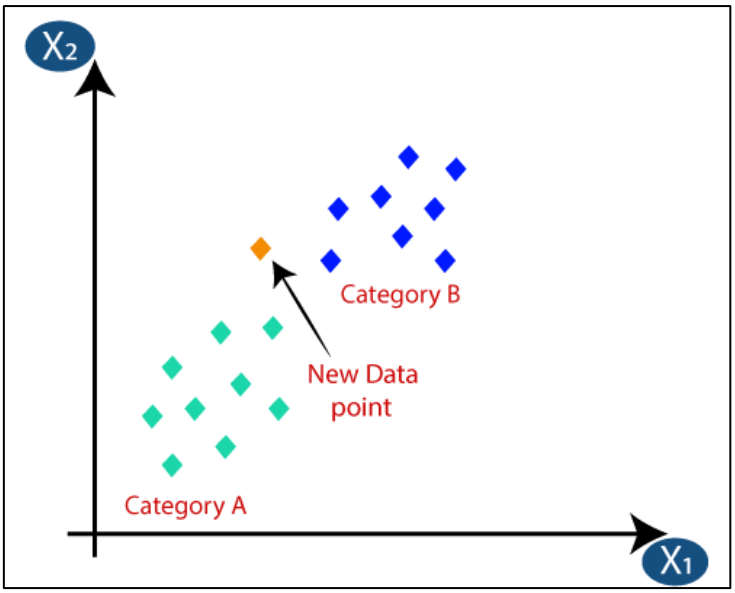


How does K-NN work?

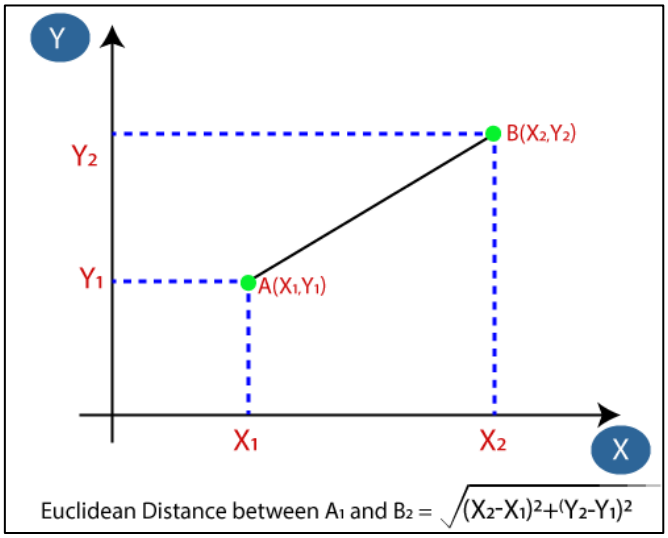
The K-NN working can be explained on the basis of the below algorithm:

- **Step-1:** Select the number K of the neighbors
- **Step-2:** Calculate the Euclidean distance of **K number of neighbors**
- **Step-3:** Take the K nearest neighbors as per the calculated Euclidean distance.
- **Step-4:** Among these k neighbors, count the number of the data points in each category.
- **Step-5:** Assign the new data points to that category for which the number of the neighbor is maximum.
- **Step-6:** Our model is ready.

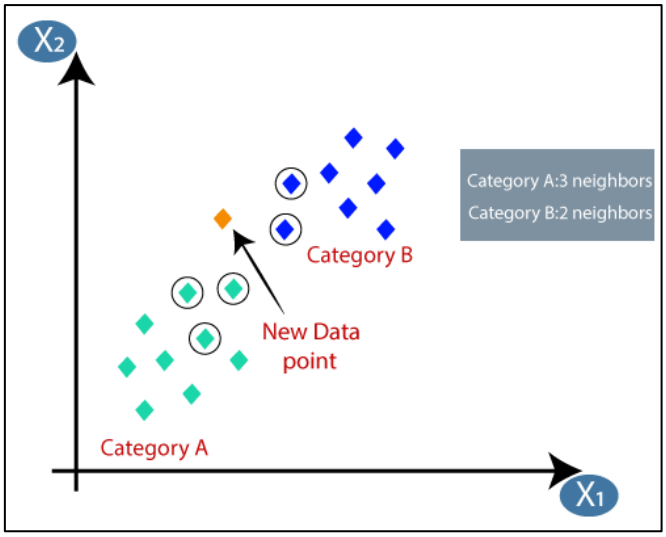
Suppose we have a new data point and we need to put it in the required category. Consider the below image:



- Firstly, we will choose the number of neighbors, so we will choose the k=5.
- Next, we will calculate the **Euclidean distance** between the data points. The Euclidean distance is the distance between two points, which we have already studied in geometry. It can be calculated as:



- By calculating the Euclidean distance we got the nearest neighbors, as three nearest neighbors in category A and two nearest neighbors in category B. Consider the below image:



- As we can see the 3 nearest neighbors are from category A, hence this new data point must belong to category A.

How to select the value of K in the K-NN Algorithm?

Below are some points to remember while selecting the value of K in the K-NN algorithm:

- There is no particular way to determine the best value for "K", so we need to try some values to find the best out of them. The most preferred value for K is 5.
- A very low value for K such as K=1 or K=2, can be noisy and lead to the effects of outliers in the model.
- Large values for K are good, but it may find some difficulties.

Advantages of KNN Algorithm:

- It is simple to implement.
- It is robust to the noisy training data
- It can be more effective if the training data is large.

Disadvantages of KNN Algorithm:

- Always needs to determine the value of K which may be complex some time.
- The computation cost is high because of calculating the distance between the data points for all the training samples.

Python implementation of the KNN algorithm

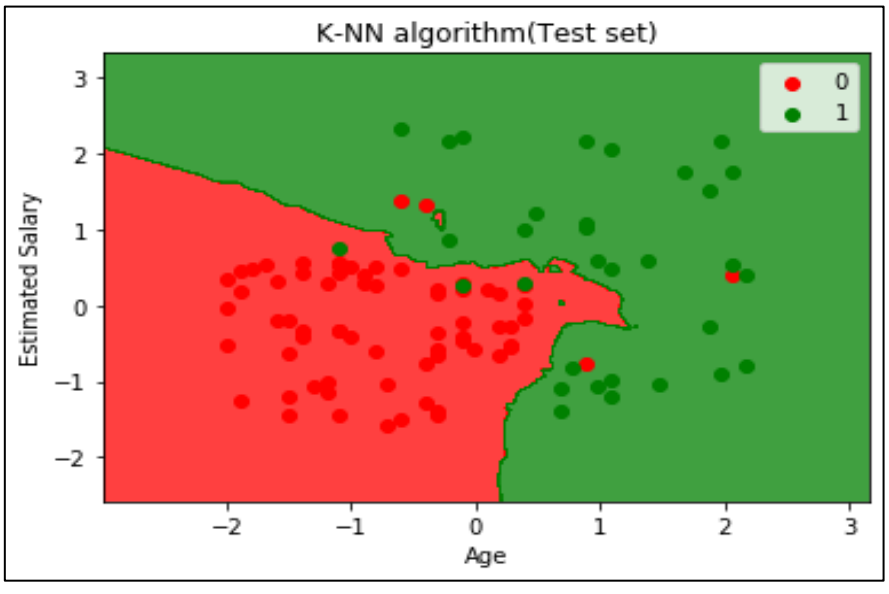
```
import numpy as np

from sklearn.neighbors import KNeighborsClassifier


X_train = np.array([[1, 2], [2, 3], [3, 1], [4, 2]])
y_train = np.array(['cat', 'cat', 'dog', 'dog'])
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)
X_test = np.array([[3, 2]])
predictions = knn.predict(X_test)
print(predictions)
```

['dog']

The K-nearest neighbors classifier predicts the label for the given test data X_test based on its nearest neighbors in the training data X_train. In this case, the test data [3, 2] is classified as 'dog'.

Steps to implement the K-NN algorithm:

- Data Pre-processing step
- Fitting the K-NN algorithm to the Training set
- Predicting the test result
- Test accuracy of the result(Creation of Confusion matrix)
- Visualizing the test set result.



2. BAYES CLASSIFIER:

- Naïve Bayes algorithm is a supervised learning algorithm, which is based on **Bayes theorem** and used for solving classification problems.
- It is mainly used in *text classification* that includes a high-dimensional training dataset.
- Naïve Bayes Classifier is one of the simple and most effective Classification algorithms which helps in building the fast machine learning models that can make quick predictions.
- **It is a probabilistic classifier, which means it predicts on the basis of the probability of an object**.
- Some popular examples of Naïve Bayes Algorithm are **spam filtration, Sentimental analysis, and classifying articles**.

Why is it called Naïve Bayes?

The Naïve Bayes algorithm is comprised of two words Naïve and Bayes, which can be described as:

- **Naïve**: It is called Naïve because it assumes that the occurrence of a certain feature is independent of the occurrence of other features. Such as if the fruit is identified on the bases of color, shape, and taste, then red, spherical, and sweet fruit is recognized as an apple. Hence each feature individually contributes to identify that it is an apple without depending on each other.
- **Bayes**: It is called Bayes because it depends on the principle of Bayes' Theorem.

Bayes' Theorem:

- Bayes' theorem is also known as **Bayes' Rule** or **Bayes' law**, which is used to determine the probability of a hypothesis with prior knowledge. It depends on the conditional probability.
- The formula for Bayes' theorem is given as:

$$P(A|B)= \frac{P(B|A)P(A)}{P(B)}$$

Where,

P(A|B) is Posterior probability: Probability of hypothesis A on the observed event B.
P(B|A) is Likelihood probability: Probability of the evidence given that the probability of a hypothesis is true.
P(A) is Prior Probability: Probability of hypothesis before observing the evidence.
P(B) is Marginal Probability: Probability of Evidence.

Working of Naïve Bayes' Classifier:

Working of Naïve Bayes' Classifier can be understood with the help of the below example:

Suppose we have a dataset of **weather conditions** and corresponding target variable "**Play**". So using this dataset we need to decide that whether we should play or not on a particular day according to the weather conditions. So to solve this problem, we need to follow the below steps:

1. Convert the given dataset into frequency tables.
2. Generate Likelihood table by finding the probabilities of given features.
3. Now, use Bayes theorem to calculate the posterior probability.

Problem: If the weather is sunny, then the Player should play or not?

Solution: To solve this, first consider the below dataset:

	Outlook	Play
0	Rainy	Yes
1	Sunny	Yes
2	Overcast	Yes
3	Overcast	Yes
4	Sunny	No
5	Rainy	Yes
6	Sunny	Yes
7	Overcast	Yes
8	Rainy	No
9	Sunny	No
10	Sunny	Yes
11	Rainy	No
12	Overcast	Yes
13	Overcast	Yes

Frequency table for the Weather Conditions:

Weather	Yes	No
Overcast	5	0
Rainy	2	2
Sunny	3	2
Total	10	5

Likelihood table weather condition:

Weather	No	Yes	
Overcast	0	5	5/14= 0.35
Rainy	2	2	4/14=0.29
Sunny	2	3	5/14=0.35
All	4/14=0.29	10/14=0.71	

Applying Bayes'theorem:

P(Yes|Sunny)= P(Sunny|Yes)*P(Yes)/P(Sunny)

P(Sunny|Yes)= 3/10= 0.3

P(Sunny)= 0.35

P(Yes)=0.71

So P(Yes|Sunny) = 0.3*0.71/0.35= **0.60**

P(No|Sunny)= P(Sunny|No)*P(No)/P(Sunny)

P(Sunny|NO)= 2/4=0.5

P(No)= 0.29

P(Sunny)= 0.35

So P(No|Sunny)= 0.5*0.29/0.35 = **0.41**

So as we can see from the above calculation that **P(Yes|Sunny)>P(No|Sunny)**

Hence on a Sunny day, Player can play the game.

Advantages of Naïve Bayes Classifier:

- Naïve Bayes is one of the fast and easy ML algorithms to predict a class of datasets.
- It can be used for Binary as well as Multi-class Classifications.
- It performs well in Multi-class predictions as compared to the other Algorithms.
- It is the most popular choice for **text classification problems**.

Disadvantages of Naïve Bayes Classifier:

- Naive Bayes assumes that all features are independent or unrelated, so it cannot learn the relationship between features.

Applications of Naïve Bayes Classifier:

- It is used for **Credit Scoring**.
- It is used in **medical data classification**.
- It can be used in **real-time predictions** because Naïve Bayes Classifier is an eager learner.
- It is used in Text classification such as **Spam filtering** and **Sentiment analysis**.

Types of Naïve Bayes Model:

There are three types of Naive Bayes Model, which are given below:

- Gaussian Naïve Bayes: Assumes features follow a normal distribution. Suitable for continuous values. Commonly used for classification tasks when predictors are continuous variables.
- Multinomial Naïve Bayes: Used when data is multinomial distributed. Popular for document classification tasks. Predictors are word frequencies in documents.
- Bernoulli Naïve Bayes: Similar to Multinomial Naïve Bayes, but predictors are independent Boolean variables. Often used for document classification based on the presence or absence of specific words.

Remember, Gaussian Naïve Bayes is for continuous variables, Multinomial Naïve Bayes is for word frequencies, and Bernoulli Naïve Bayes is for presence/absence of specific features.

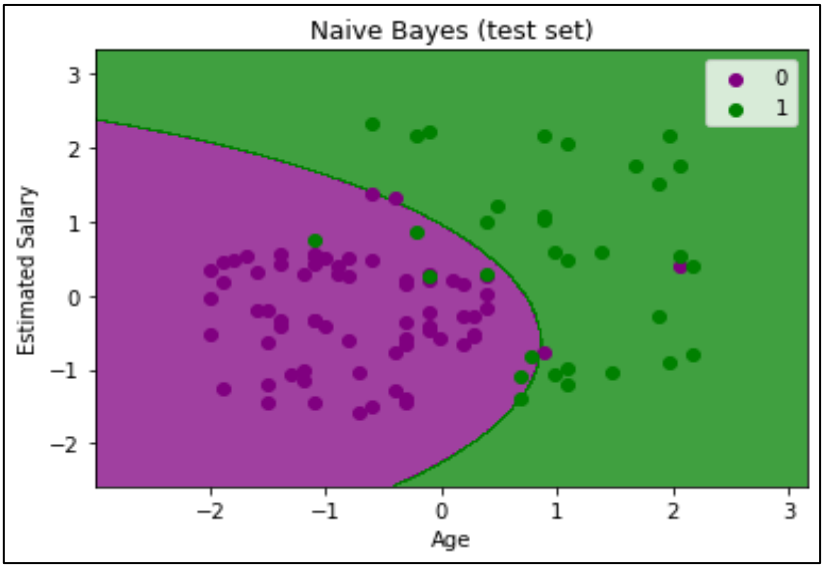
Python Implementation of the Naïve Bayes algorithm:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

iris = load_iris()
X_train,X_test,y_train,y_test = train_test_split(iris.data, iris.target, test_size=0.2,
random_state=42)
gnb = GaussianNB()
gnb.fit(X_train, y_train)
y_pred = gnb.predict(X_test)
print("Accuracy:", accuracy)
```

Accuracy: 0.9666666666666667

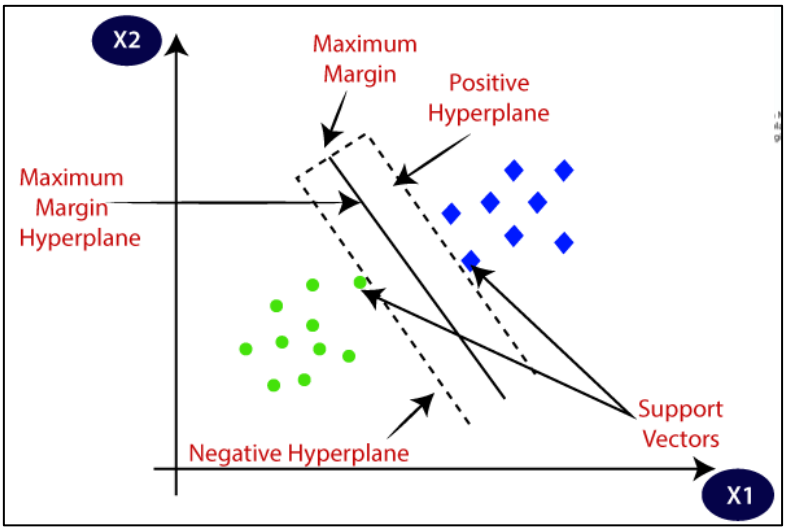
Steps to implement:

- Data Pre-processing step
- Fitting Naive Bayes to the Training set
- Predicting the test result
- Test accuracy of the result(Creation of Confusion matrix)
- Visualizing the test set result.



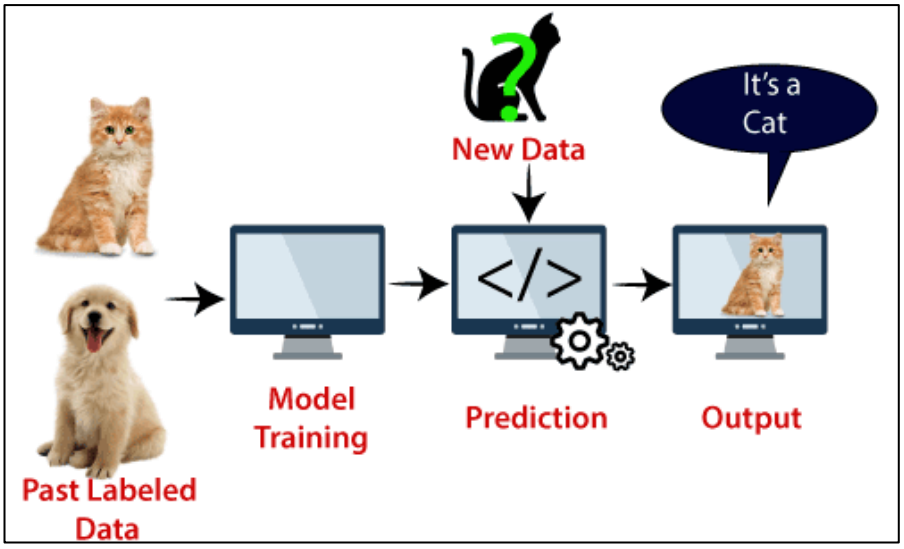
3. SUPPORT VECTOR MACHINES:

- SVM is a popular Supervised Learning algorithm used mainly for Classification problems.
- The goal of SVM is to create a decision boundary (hyperplane) that separates different classes in a multi-dimensional space.
- SVM selects important points called support vectors to define the decision boundary.
- SVM handles complex datasets effectively and finds the optimal hyperplane that maximizes the margin between classes.
- The optimal hyperplane helps classify new data points based on their position in relation to the decision boundary.



Example:

- We want to build a model that can accurately determine whether it is a cat or a dog. We can achieve this using SVM.
- We start by training our SVM model with a large dataset of cat and dog images. This allows the model to learn and understand the distinct features of cats and dogs.
- Once the training is complete, we test the model with the image of the strange creature.
- The SVM algorithm creates a decision boundary (hyperplane) that separates the cat and dog data points. It selects extreme cases (support vectors) that help define this boundary.
- Based on these support vectors, the model classifies the strange creature as a cat or a dog



SVM algorithm can be used for **Face detection, image classification, text categorization,** etc.

Types of SVM

SVM can be of two types:

- **Linear SVM:** Linear SVM is used for linearly separable data, which means if a dataset can be classified into two classes by using a single straight line, then such data is termed as linearly separable data, and classifier is used called as Linear SVM classifier.
- **Non-linear SVM:** Non-Linear SVM is used for non-linearly separated data, which means if a dataset cannot be classified by using a straight line, then such data is termed as non-linear data and classifier used is called as Non-linear SVM classifier.

Hyperplane and Support Vectors in the SVM algorithm:

Hyperplane:

- The hyperplane is the best decision boundary created by SVM to separate classes in a multi-dimensional space.
- In simple terms, the hyperplane is like a line or plane that separates the data points of different classes.
- The number of dimensions in the hyperplane depends on the number of features in the dataset. For example, if there are 2 features, the hyperplane is a straight line. If there are 3 features, it becomes a 2-dimensional plane.

- The key idea is to find a hyperplane with the maximum margin, which is the maximum distance between the data points. This maximizes the separation between classes and improves the accuracy of classification.

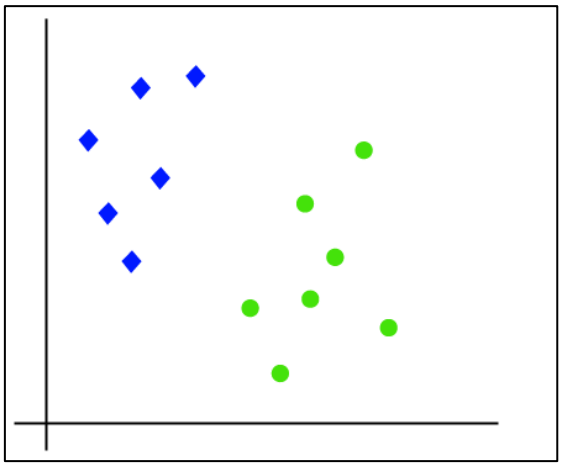
Support Vectors:

The data points or vectors that are the closest to the hyperplane and which affect the position of the hyperplane are termed as Support Vector. Since these vectors support the hyperplane, hence called a Support vector.

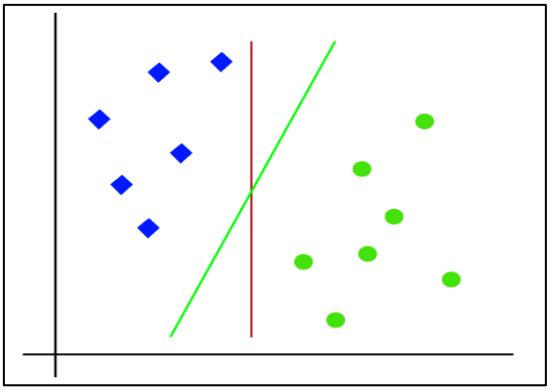
How does SVM works?

Linear SVM:

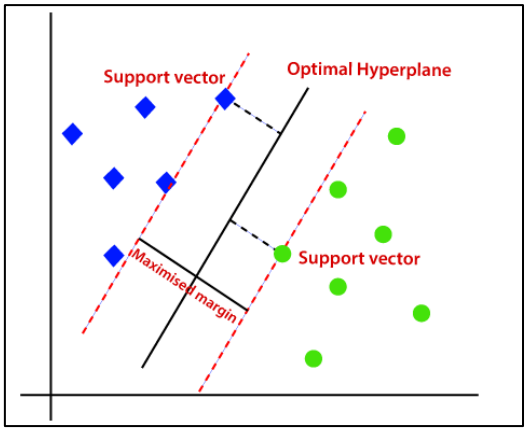
The working of the SVM algorithm can be understood by using an example. Suppose we have a dataset that has two tags (green and blue), and the dataset has two features x1 and x2. We want a classifier that can classify the pair(x1, x2) of coordinates in either green or blue. Consider the below image:



So as it is 2-d space so by just using a straight line, we can easily separate these two classes. But there can be multiple lines that can separate these classes. Consider the below image:

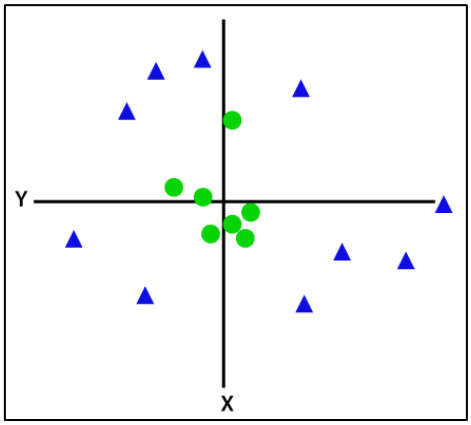


Hence, the SVM algorithm helps to find the best line or decision boundary; this best boundary or region is called as a **hyperplane**. SVM algorithm finds the closest point of the lines from both the classes. These points are called support vectors. The distance between the vectors and the hyperplane is called as **margin**. And the goal of SVM is to maximize this margin. The **hyperplane** with maximum margin is called the **optimal hyperplane**.



Non-Linear SVM:

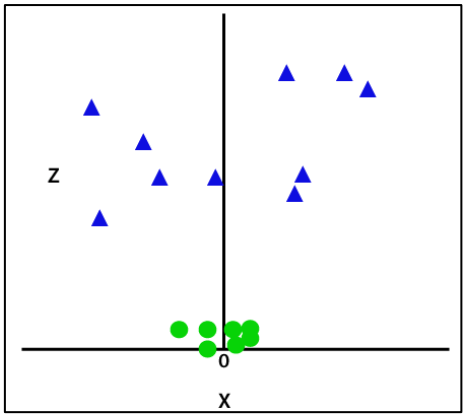
If data is linearly arranged, then we can separate it by using a straight line, but for non-linear data, we cannot draw a single straight line. Consider the below image:



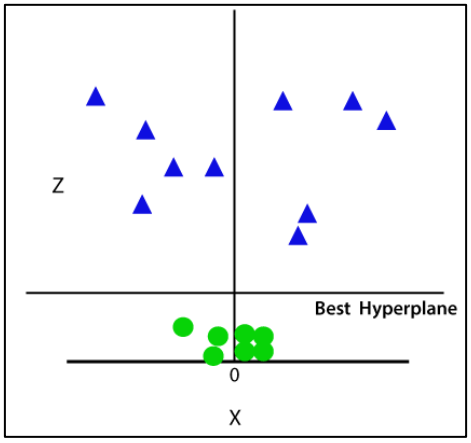
So to separate these data points, we need to add one more dimension. For linear data, we have used two dimensions x and y, so for non-linear data, we will add a third dimension z. It can be calculated as:

$$z=x^2+y^2$$

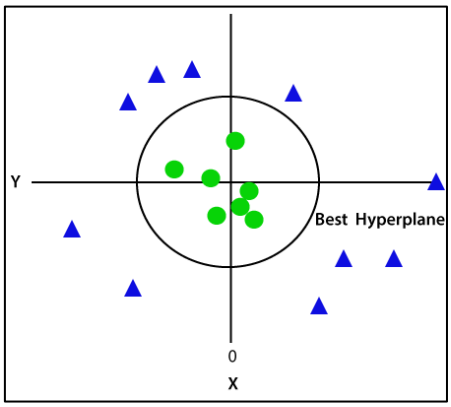
By adding the third dimension, the sample space will become as below image:



So now, SVM will divide the datasets into classes in the following way. Consider the below image:



Since we are in 3-d Space, hence it is looking like a plane parallel to the x-axis. If we convert it in 2d space with z=1, then it will become as:

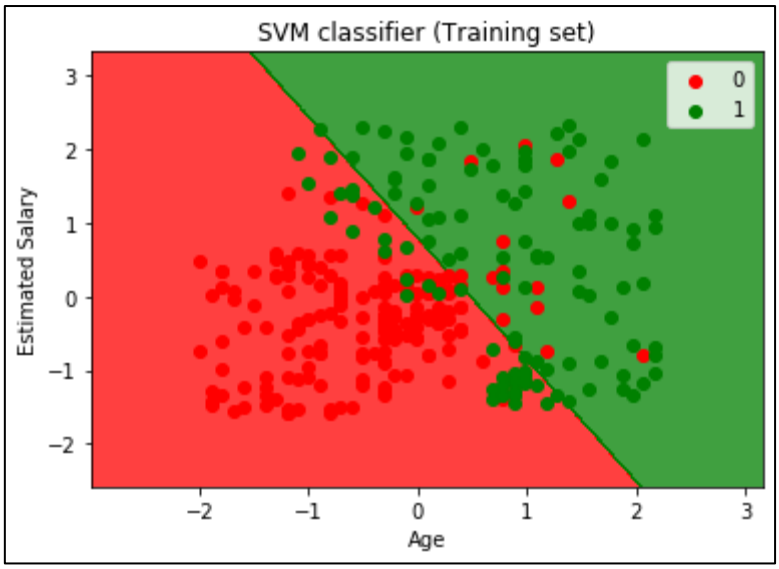


Hence, we get a circumference of radius 1 in case of non-linear data.

Python Implementation of Support Vector Machine

```
from sklearn import svm
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score


iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target,
test_size=0.2, random_state=42)
clf = svm.SVC(kernel='linear')
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

Accuracy: 0.9666666666666667



4. OPTICAL CHARACTER RECOGNITION:

- Optical character recognition (OCR) is the process of interpreting images of hand- or machine-written text.
- A common example is text extraction from scanned documents such as zip-codes on letters or book pages such as the library volumes in Google Books
- Here we will look at a simple OCR problem of recognizing numbers in images of printed Sudokus.
- Sudokus are a form of logic puzzles where the goal is to fill a 9 × 9 grid with the numbers 1 to 9
- so that each column, each row, and each 3 × 3 sub-grid contains all nine digits.

Training a Classifier:

- For this classification problem we have ten classes, the numbers 1 . . . 9, and the empty cells.
  Let's give the empty cells the label 0 so that our class labels are 0 . . . 9.
- To train this ten-class classifier, we will use a dataset of images of cropped Sudoku cells.
- In the file sudoku_images.zip are two folders, "ocr_data" and "sudokus".
- For now, take a look at the folder "ocr_data".
- It contains two subfolders with images, one for training and one for testing.
  The images are named with the first character equal to the class (0 . . . 9).
- Figure 8-6 shows some samples from the training set.
- The images are grayscale and roughly 80 × 80 pixels

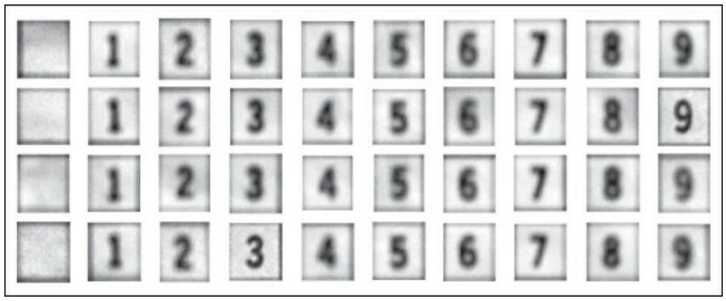


Figure 8-6. Sample training images for the 10 classes of the Sudoku OCR classifier.

Example :

```
def compute_feature(im):
""" Returns a feature vector for an
ocr image patch. """
# resize and remove border
        norm_im = imresize(im,(30,30))
        norm_im = norm_im[3:-3,3:-3]
        return norm_im.flatten()
```

Rectifying Images

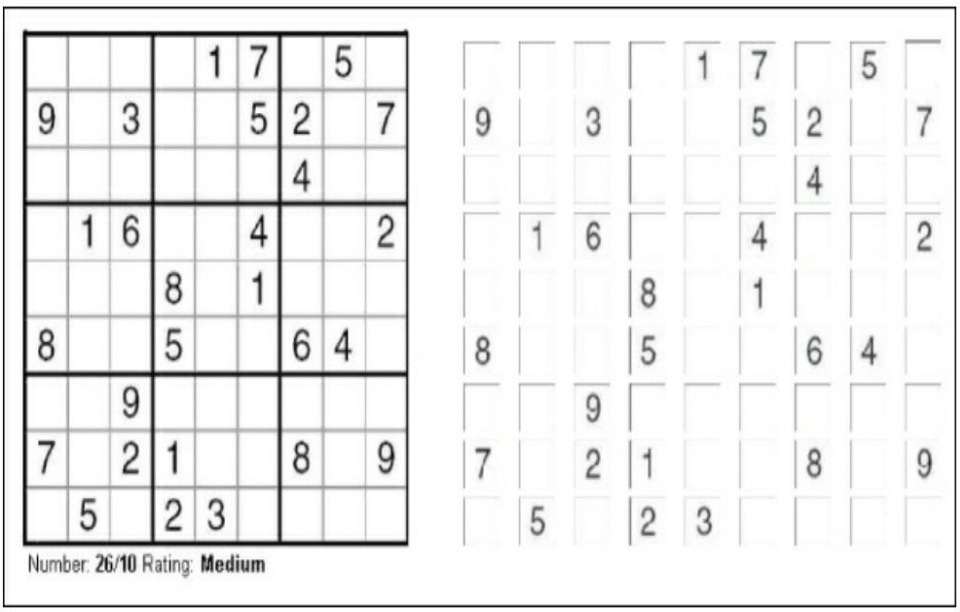


Figure 8-7. An example of detecting and cropping the fields of a Sudoku grid: image of a Sudoku grid (left); the 9 × 9 cropped images of the individual cells to be sent to the OCR classifier (right).

Searching Image

This chapter shows how to use text mining techniques to search for images based on their visual content. The basic ideas of using visual words are presented and the details of a complete setup are explained and tested on an example image data set.

5. Content-Based Image Retrieval:

Content-based image retrieval (CBIR) aims to find visually similar images from a database. It involves searching for images with similar colors, textures, objects, or scenes. However, comparing a query image to every image in a large database is inefficient.
To overcome this, researchers have adapted text mining techniques for CBIR. These techniques allow for efficient searching through millions of images for similar content.

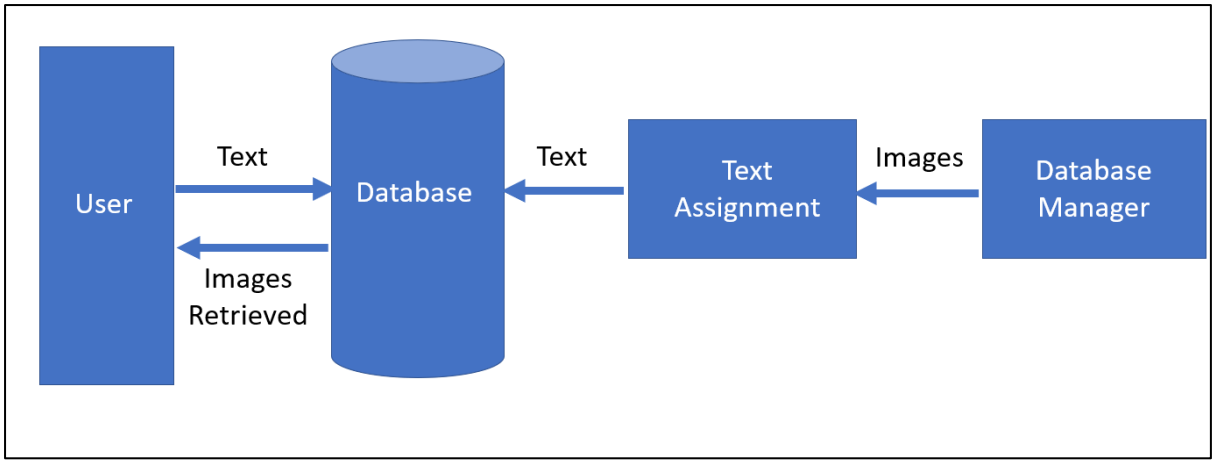
Inspiration from Text Mining — The Vector Space Model

The vector space model:

- Represents text documents as vectors by counting word frequencies.
- Ignores common words (stop words) like "the," "and," and "is."
- Normalizes vectors to handle differences in document length.
- Weights vector components to give importance to certain words.
- Words that appear frequently in the document have higher weights.
- Words that are common across all documents have lower weights.
- tf-idf (term frequency–inverse document frequency) is a common weighting scheme.
- tf-idf calculates word frequencies in a document and inversely weighs them based on their frequency in the entire corpus.
- In simple terms, the vector space model represents text documents as vectors of word frequencies. Common words are ignored, vectors are normalized, and words are weighted based on importance. tf-idf is a popular weighting scheme in this model.

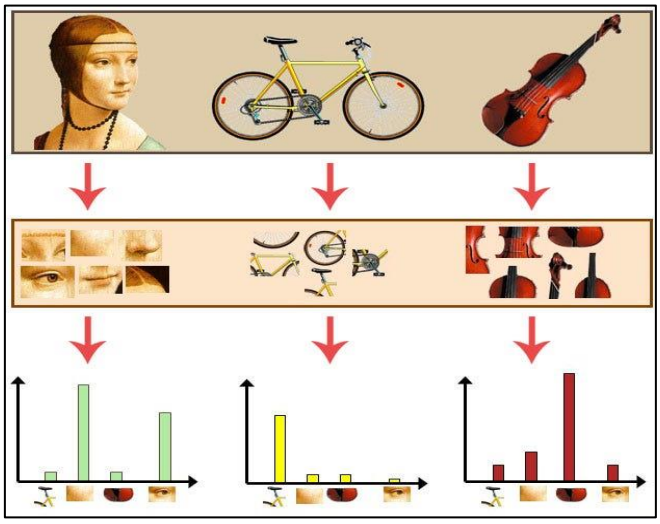
$$\mathrm{tf}_{w,d} = \frac{n_w}{\sum_j n_j},$$

where nw is the number of occurrences of w in d. To normalize, this is divided by the total number of occurrences of all words in the document.



6. VISUAL WORDS:

- To apply text mining techniques to images, we first need to create the visual equivalent of words.
- This is usually done using local descriptors like the SIFT—Scale-Invariant Feature Transform
- The visual words are constructed using some clustering algorithm applied to the feature descriptors extracted from a (large) training set of images.
- The most common choice is k-means, algorithm
- Visual words are nothing but a collection of vectors in the given feature descriptor space;
- In the case of k-means, they are the cluster centroids.
- Representing an image with a histogram of visual words is then called a bag-of visual-words model.



7. INDEXING IMAGES:

- Before we can start searching, we need to set up a database with the images and their visual word representations

Setting Up the Database:

- To start indexing images, we first need to set up a database.
- Indexing images in this context means
- extracting descriptors from the images,
- converting them to visual words using a vocabulary, and storing the visual words and word histograms with information about which image they belong to
- Here we will use SQLite as database.
- SQLite is a database that stores everything in a single file and is very easy to set up and use.

Table 7-1. A simple database schema for storing images and visual words.

imlist	imwords	imhistograms
rowid	imid	imid
filename	wordid	histogram
	vocname	vocname

8. SEARCHING THE DATABASE FOR IMAGES:

- With a set of images indexed, we can search the database for similar images.
- Here we have used a bag-of-word representation for the whole image,

  but the procedure explained here is generic and can be used to find

  - **similar objects,**
  - **similar faces,**
  - **similar colors, etc.**

- It all depends on the images and descriptors used.
- To handle searches, we introduce a Searcher class to imagesearch.py:

```
class Searcher(object):
    def __init__(self,db,voc):
     """ Initialize with the name of the database. """“
        self.con = sqlite.connect(db)
        self.voc = voc
    def __del__(self):
        self.con.close()
```

9. RANKING RESULTS USING GEOMETRY

- One way to have the feature points improve results is to re-rank the top results using some criteria that takes the features geometric relationships
- The most common approach is to fit homographies between the feature locations in the query image and the top result images
- Here is what a complete example of loading all the model files and re-ranking the top results using homographies looks like:

```
import pickle
import sift
import imagesearch
import homography
# load image list and vocabulary
with open('ukbench_imlist.pkl','rb') as f:
    imlist = pickle.load(f)
    featlist = pickle.load(f)
    nbr_images = len(imlist)
with open('vocabulary.pkl', 'rb') as f:
   voc = pickle.load(f)
   src = imagesearch.Searcher('test.db',voc)
```

Homographies are computed from the matches and the number of inliers counted. If the homography fitting fails, we set the inlier list to an empty list. Finally, we sort the dictionary rank that contains image index and inlier count according to decreasing number of inliers. The result lists are printed to the console and the top images visualized.

The output looks like this:
top matches (regular): [39, 22, 74, 82, 50, 37, 38, 17, 29, 68, 52, 91, 15, 90, 31, ...]
top matches (homography): [39, 38, 37, 45, 67, 68, 74, 82, 15, 17, 50, 52, 85, 22, 87, ...]

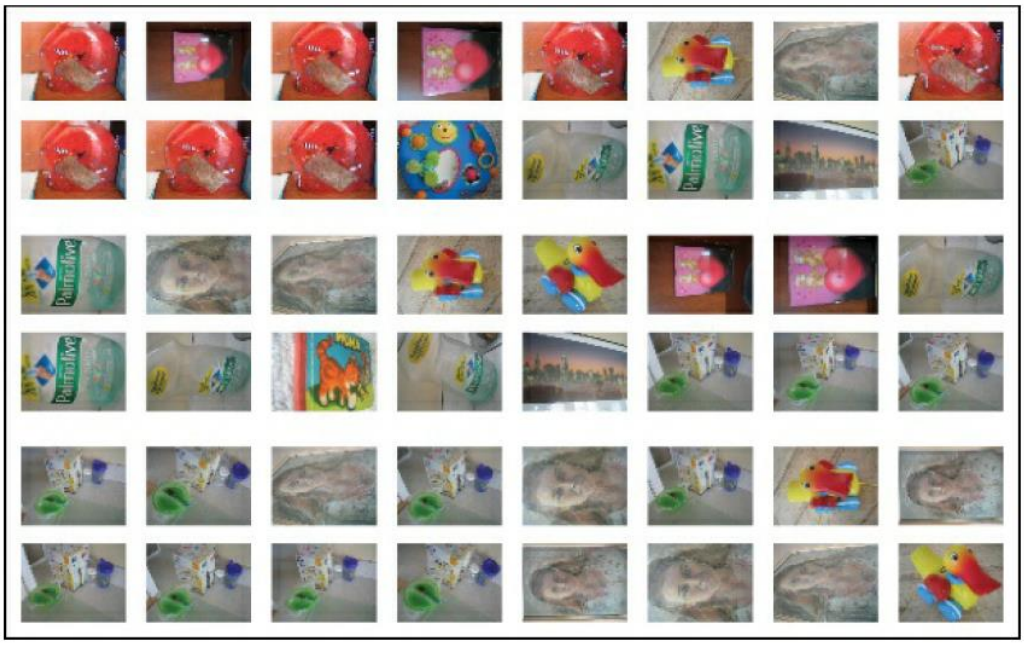


Figure 7-3. Some example search results with re-ranking based on geometric consistency using homographies. For each example, the top row is the regular result and the bottom row the re-ranked result.

10. BUILDING DEMOS AND WEB APPLICATIONS

we'll take a look at a simple way of building demos and web applications with Python. By making demos as web pages, you automatically get cross platform support and an easy way to show and share your project with minimal requirements. In the sections below we will go through an example of a simple image search engine.

Creating Web Applications with CherryPy:

To build these demos, we will use the CherryPy package, available at http://www.cherrypy.org/. CherryPy is a pure Python lightweight web server that uses an object-oriented model. let's build an image search web demo on top of the image searcher.

Image Search Demo:

- First, we need to initialize with a few html tags and load the data using Pickle.
- We need the vocabulary for the Searcher object that interfaces with the database.
- Create a file searchdemo.py and add the following class with two methods:
    I. The first part specifies which IP address and port to use.
    II. The second part enables a local folder for reading (in this case "tmp/"). This should be set to the folder containing your images.

```
import cherrypy
import imagesearch

class SearchDemo(object):
   def __init__(self, imlist_file, vocabulary_file):
      self.imlist = open(imlist_file).readlines()
      self.voc = imagesearch.load_vocabulary(vocabulary_file)
      self.maxres = 15

   @cherrypy.expose
   def index(self, query=None):
      src = imagesearch.Searcher('web.db', self.voc)
      html = "<br />".join([
        "<a href='?query={imname}'><img src='{imname}' width='100'
        /></a>".format(imname=src.get_filename(ndx))
          for dist, ndx in src.query(query)[:self.maxres]
      ])
        return f"<html><head><title>Image search example</title>
        </head><body>{html}</body></html>"

if __name__ == '__main__':
   cherrypy.quickstart(SearchDemo('webimlist.txt', 'vocabulary.pkl'))
```

Start your web server with
$ python searchdemo.py

from the command line. Opening your browser and pointing it at the right URL (in this case http://127.0.0.1:8080/) should show the initial screen with a random

selection of images. This should look like the top image in Figure 7-4. Clicking an image starts a query and shows the top results. Clicking an image in the results starts a new query with that image, and so on. There is a link to get back to the starting state of a random selection (by passing an empty query).

Figure 7-4. Some example search results on the ukbench data set: the starting page, which shows a random selection of the images (top); sample queries (bottom). The query image is shown on the top-left corner followed by the top image results shown row-wise.

This example shows a full integration from web page to database queries and presentation of results. Naturally, we kept the styling and options to a minimum and there are many possibilities for improvement. For example, you may add a stylesheet to make it prettier or upload files to use as queries.

UNIT-V

Image Segmentation

Image segmentation is the process of partitioning an image into meaningful regions. Regions can be foreground versus background or individual objects in the image. The regions are constructed using some feature such as color, edges, or neighbor similarity. In this chapter we will look at some different techniques for segmentation.

1. GRAPH CUTS:

A graph is a collection of dots connected by lines. The lines can be arrows or non-arrows and can have numbers on them. Graph cuts are used to divide this group into two parts and are commonly applied in computer vision for tasks like creating 3D images, merging pictures, and segmenting images into different sections.

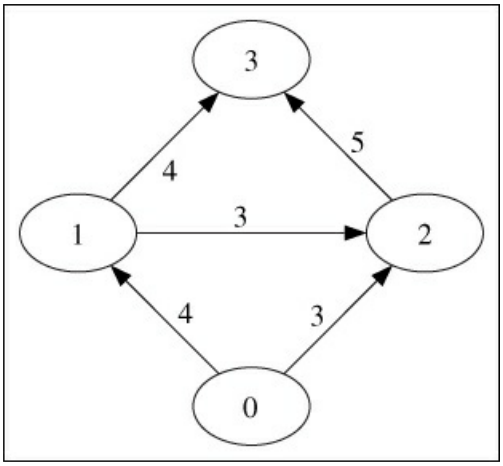
Graph cut segmentation begins by creating a graph using the pixels of an image as dots. The neighboring pixels are connected with lines. Two special dots, called the source and sink, are also added. The objective is to split the graph into two parts in a way that minimizes the cut.

The cost of a graph cut, denoted as C (which is a set of edges), is calculated by adding up the weights of the edges in the cut.

$$E_{cut} = \sum_{(i,j)\in C} w_{ij},$$

where w_{ij} is the weight of the edge (i, j) from node i to node j in the graph and the sum is taken over all edges in the cut C.

The idea behind graph cut segmentation is that pixels that look similar and are close together should be in the same group. By making the cut as small as possible, we find the best way to split the pixels into meaningful sections. Finding the minimum cut (min cut) in a graph is the same as finding the maximum flow (max flow) between a source and a sink. There are efficient algorithms available to solve these max flow/min cut problems.



A simple directed graph created using python-graph.

In our graph cut examples, we will use a Python package called python-graph. This package provides helpful graph algorithms. We will specifically use the maximum_flow() function, which calculates the max flow/min cut using the Edmonds-Karp algorithm.

One advantage of using a package written entirely in Python is that it's easy to install and compatible with different systems. However, it may not be the fastest implementation available. For our purposes, the performance is sufficient, but for larger images, a faster implementation would be necessary.

Here's a simple example of using python-graph to compute the max flow/min cut of a small graph. (Code example can be given separately)

1.Import the necessary modules:

```
from pygraph.classes.digraph import digraph
from pygraph.algorithms.minmax import maximum_flow
```

2.Create a directed graph and add nodes:

```
gr = digraph()
gr.add_nodes([0, 1, 2, 3])
```

3.Add edges with their weights:

```
gr.add_edge((0, 1), wt=4)
gr.add_edge((1, 2), wt=3)
gr.add_edge((2, 3), wt=5)
```

```
gr.add_edge((0, 2), wt=3)
gr.add_edge((1, 3), wt=4)
```

4.Calculate the maximum flow and minimum cut:

```
flows, cuts = maximum_flow(gr, 0, 3)
```

5.Print the results:

```
print('Flow is:', flows)
print('Cut is:', cuts)
```

The flow dictionary shows the flow through each edge, and the cut dictionary represents the label for each node, with 0 indicating the part of the graph containing the source and 1 indicating the nodes connected to the sink.

For the example graph, the output should be:

```
Flow is: {(0, 1): 4, (1, 2): 0, (1, 3): 4, (2, 3): 3, (0, 2): 3}
Cut is: {0: 0, 1: 1, 2: 1, 3: 1}
```

The cut dictionary confirms that the minimum cut separates the nodes into two sets: source (0) and sink (1)

Graphs from Images

- A graph is a way to represent data with nodes (points) and edges (connections) between them. In this case, we are using the pixels of an image as nodes in the graph.
- We consider a 4-neighborhood, which means that each pixel is connected to its immediate neighbors: the ones directly above, below, left, and right of it.
- We also have two image regions, foreground and background. Each pixel belongs to either the foreground or the background region.
- So, we create a graph where each pixel is a node, and we connect each pixel to its 4-neighborhood pixels. We label each node with its corresponding pixel value and whether it belongs to the foreground or background.

- This graph can then be used for different purposes, such as image segmentation or object recognition, by analyzing the connections and relationships between the nodes.

Here's how to build the graph:

- Every pixel node has an incoming edge from the source node.
- Every pixel node has an outgoing edge to the sink node.
- Every pixel node has one incoming and one outgoing edge to each of its neighbors.

Bayes Classifier on the color values of the pixels. Given that we have trained a Bayes classifier on foreground and background pixels (from the same image or from other images), we can compute the probabilities $p_F(I_i)$ and $p_B(I_i)$ for the foreground and background. Here I_i is the color vector of pixel i.

To assign weights to the edges between pixels and between pixels and the source/sink, you need a model that can segment and determine the maximum flow allowed for each edge. The weight of the edge between two pixels is called "wij". The weight from the source to a pixel is called "wsi" and from a pixel to the sink is called "wit".

We can now create a model for the edge weights as follows:

$$w_{si} = \frac{p_F(I_i)}{p_F(I_i) + p_B(I_i)}$$

$$w_{it} = \frac{p_B(I_i)}{p_F(I_i) + p_B(I_i)}$$

$$w_{ij} = \kappa\, e^{-|I_i - I_j|^2/\sigma}.$$

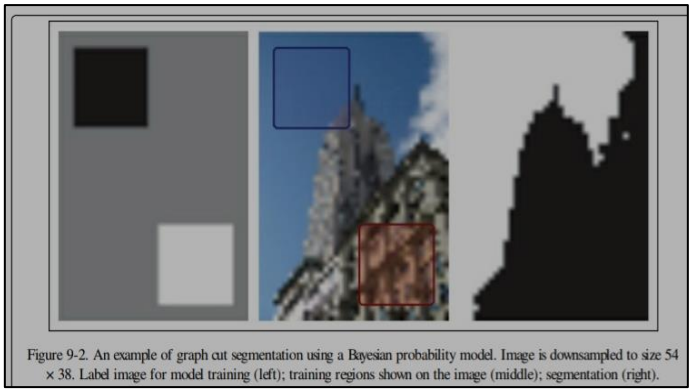


Figure 9-2. An example of graph cut segmentation using a Bayesian probability model. Image is downsampled to size 54 × 38. Label image for model training (left); training regions shown on the image (middle); segmentation (right).

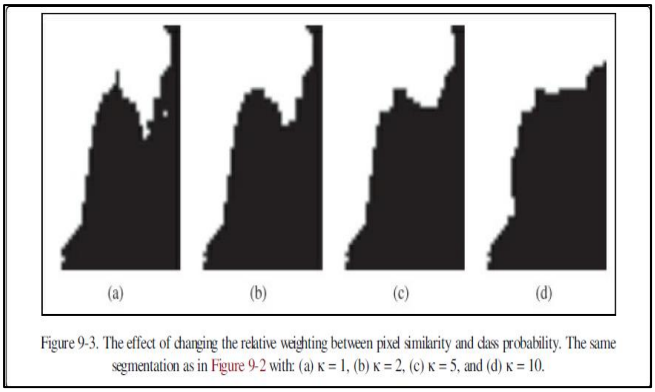


Figure 9-3. The effect of changing the relative weighting between pixel similarity and class probability. The same segmentation as in Figure 9-2 with: (a) κ = 1, (b) κ = 2, (c) κ = 5, and (d) κ = 10.

Segmentation with User Input:

Graph cut segmentation can be enhanced by incorporating user input in various ways. One approach involves users providing markers for the foreground and background by drawing directly on an image. Alternatively, they can select a region containing the foreground using a bounding box or a "lasso" tool.

These images come with ground truth labels for measuring segmentation performance. They also come with annotations simulating a user selecting a rectangular image region or drawing on the image with a “lasso” type tool to mark foreground and background.

We can use these user inputs to get training data and apply graph cuts to segment the image guided by the user input.

The user input is encoded in bitmap images with the following meaning:

Pixel value	Meaning
0, 64	background
128	unknown
255	foreground

2. SEGMENTATION USING CLUSTERING:

- The graph cut formulation is to solves the segmentation problem by finding a discrete solution using max flow/min cut over an image graph.
- Defining a cut cost that takes into account the size of the groups and "normalizes" the cost with the size of the partitions. The normalized cut formulation modifies the cut cost of equation

$$E_{ncut} = \frac{E_{cut}}{\sum_{i \in A} w_{ix}} + \frac{E_{cut}}{\sum_{j \in B} w_{jx}},$$

Where, A and B indicate the two sets of the cut and the sums add the weights from A and B, respectively, to all other nodes in the graph (which are pixels in the image in this case). This sum is called the association and for images where pixels have the same number of connections to other pixels, it is a rough measure of the size of the partitions.

The normalized cut segmentation is obtained as the minimum of the optimization problem

$$\min_{\mathbf{y}} \frac{\mathbf{y}^T(D - W)\mathbf{y}}{\mathbf{y}^T D\mathbf{y}}$$

Let D be the diagonal matrix of the row sums of S, D =diag(d_i).
where the vector **y** contains the discrete labels that satisfy the constraints $y_i \in \{1, -b\}$ for some constant b (meaning that **y** only takes two discrete values) and $\mathbf{y}_T$ D sum to zero.

Relaxing the problem results in solving for eigenvectors of a Laplacian matrix

$$L = D^{-1/2}WD^{-1/2},$$

just like the spectral clustering case. The only remaining difficulty is now to define the between-pixel edge weights wij. Normalized cuts have many similarities to spectral clustering and the underlying theory overlaps somewhat.

Example:(Spectral Clustering and K-mean Clustering)

Here we used the Spectral Clustering algorithm

```
import numpy as np
from sklearn.cluster import SpectralClustering
from sklearn.preprocessing import StandardScaler


def spectral_clustering(S, k, ndim):
    """Spectral clustering from a similarity matrix."""
    # Create Laplacian matrix
    D = np.diag(np.sum(S, axis=1))
    L = D - S
    # Compute eigenvectors of L
    _, V = np.linalg.eig(L)
    # Create feature vector from ndim first eigenvectors by stacking eigenvectors as columns
    features = V[:,: ndim]
    # Apply standardization to the features
    scaler = StandardScaler()
    features = scaler.fit_transform(features)
    # Perform spectral clustering
    spectral = SpectralClustering(n_clusters=k, affinity='nearest_neighbors')
    spectral.fit(features)
    code = spectral.labels_
    return code
```

Here we used the k-means clustering algorithm

```
import ncut
from scipy.misc import imresize

im = array(Image.open('C-uniform03.ppm'))
m,n = im.shape[:2]

# resize image to (wid,wid)
wid = 50
rim = imresize(im,(wid,wid),interp='bilinear')
rim = array(rim,'f')

# create normalized cut matrix
A = ncut.ncut_graph_matrix(rim,sigma_d=1,sigma_g=1e-2)

# cluster
code,V = ncut.cluster(A,k=3,ndim=3)

# reshape to original image size
codeim = imresize(code.reshape(wid,wid),(m,n),interp='nearest')

# plot result
figure()
imshow(codeim)
gray()
show()
```

In the example, we used one of the hand gesture images from the Static Hand Posture Database with k = 3. The resulting segmentation is shown in Figure 9-5 together with the first four eigenvectors.

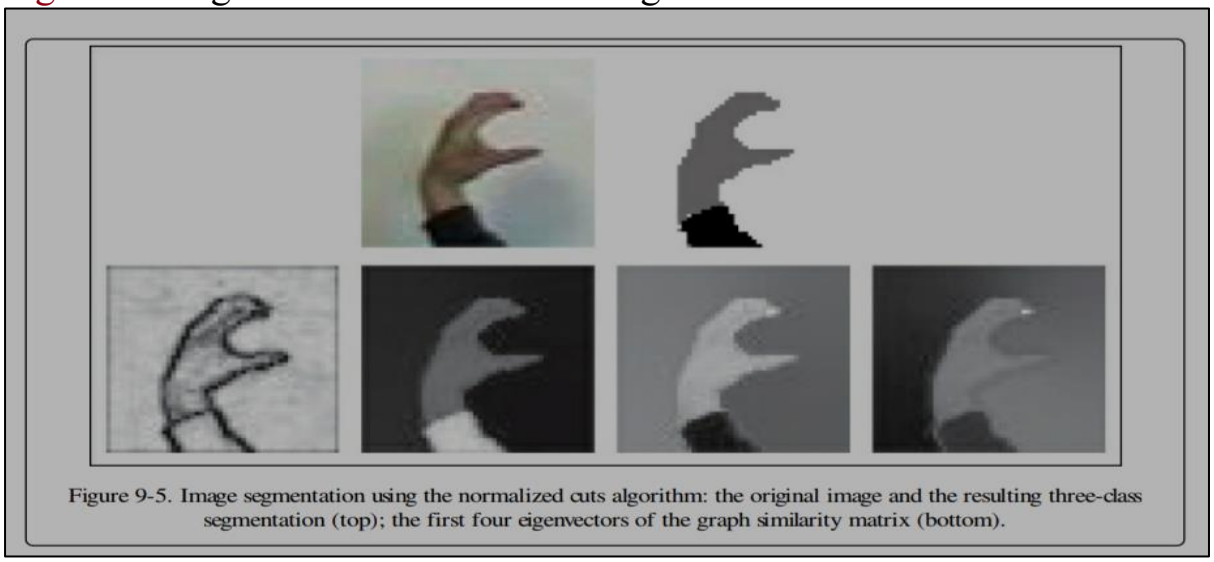


Figure 9-5. Image segmentation using the normalized cuts algorithm: the original image and the resulting three-class segmentation (top); the first four eigenvectors of the graph similarity matrix (bottom).

3.VARIATIONAL METHODS:

Variational methods are a popular class of optimization techniques used in image segmentation. Image segmentation is the process of dividing an image into multiple segments or regions, each with distinct visual properties. Image segmentation is a fundamental task in computer vision and is used in various applications such as object recognition, face detection, medical image analysis, and video surveillance.

Variational methods are used to find an optimal segmentation by minimizing an energy functional. The energy functional consists of two components: a data fidelity term, which measures the similarity between the image and the segmentation, and a regularization term, which promotes desired properties like smoothness or compactness. The goal is to find the segmentation variables that minimize this energy functional using optimization techniques such as gradient descent or conjugate gradient methods.

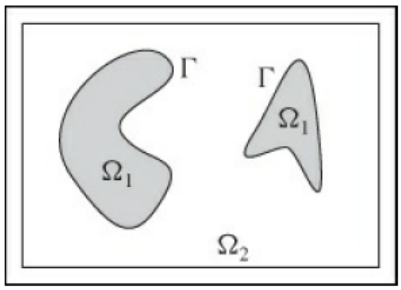


Figure 9-7. The piecewise constant Chan-Vese segmentation model.

we also saw examples like the ROF de-noising, k-means, and support vector machines. These are examples of optimization problems.

When the optimization is taken over functions, the problems are called variational problems, and algorithms for solving such problems are called variational methods. Let's look at a simple and effective variational model.

The Chan-Vese segmentation model assumes that an image can be divided into two regions, such as foreground and background. The model is based on a piecewise constant image assumption, meaning that each region has a relatively uniform intensity. The goal is to find the optimal segmentation by minimizing the Chan-Vese model energy.

The model energy is determined by a collection of curves (denoted as Γ) that separate the image into the two regions Ω1 and Ω2. The energy function is defined as the minimum of the Chan-Vese model and can be written as:

E (Γ, c1, c2) = Length(Γ) + λ * (Area(Ω1) * |Mean(Ω1) - c1|^2 + Area(Ω2) * |Mean(Ω2) - c2|^2

Here, Length(Γ) represents the length of the curves, Area(Ω1) and Area(Ω2) represent the areas of the regions, c1 and c2 represent the mean intensities of Ω1 and Ω2 respectively, and λ is a parameter that controls the trade-off between curve length and region fitting.

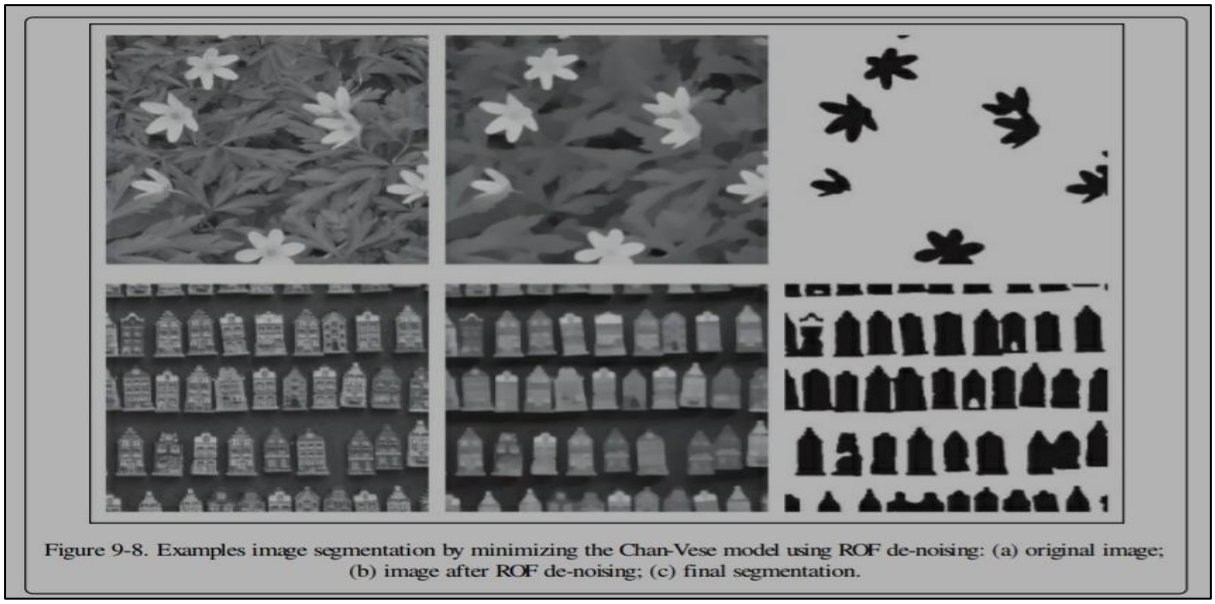


Figure 9-8. Examples image segmentation by minimizing the Chan-Vese model using ROF de-noising: (a) original image; (b) image after ROF de-noising; (c) final segmentation.

Eg:

Minimizing the Chan-Vese model now becomes a ROF de-noising followed by thresholding:

```
import rof

im = array(Image.open('ceramic-houses_t0.png').convert("L"))
U,T = rof.denoise(im,im,tolerance=0.001)
t = 0.4 #threshold

import scipy.misc
scipy.misc.imsave('result.pdf',U < t*U.max())
```

In this case, we turn down the tolerance threshold for stopping the ROF iterations to make sure we get enough iterations. Figure 9-8 shows the result on two rather difficult images.

Opencv

This chapter gives a brief overview of how to use the popular computer vision library OpenCV through the Python interface. OpenCV is a C++ library for real-time computer vision initially developed by Intel and now maintained by Willow Garage. OpenCV is open source and released under a BSD license, meaning it is free for both academic and commercial use. As of version 2.0, Python support has been greatly improved. We will go through some basic examples and look deeper into tracking and video.

4.Python Interface:

OpenCV is a C++ library with modules that cover many areas of computer vision. Besides C++ (and C), there is growing support for Python as a simpler scripting language through a Python interface on top of the C++ code base. The Python interface is still under development—not all parts of OpenCV are exposed and many functions are undocumented. This is likely to change, as there is an active community behind this interface.

The current OpenCV version (4.7.0) actually comes with two Python interfaces. The old cv module uses internal OpenCV datatypes and can be a little tricky to use from NumPy. The new cv2 module uses NumPy arrays and is much more intuitive to use. The module is available as

```
import cv2
```

and the old module can be accessed as

```
import cv2.cv
```

We will focus on the cv2 module. Look out for future name changes, as well as changes in function names and definitions in future versions. OpenCV and the Python interface is under rapid development.

5. OpenCV Basics:

OpenCV comes with functions for reading and writing images, as well as matrix operations and math libraries. For the details on OpenCV, there is an excellent (C++ only).

Let's look at some of the basic components and how to use them.

Reading and Writing Images

This short example will load an image, print the size, and convert and save the image in .png format:

```
import cv2
im = cv2.imread('empire.jpg')
h,w = im.shape[:2]
print h,w
cv2.imwrite('result.png',im)
```

The function imread() returns the image as a standard NumPy array and can handle a wide range of image formats. You can use this function as an alternative to the PIL image reading if you like. The function imwrite() automatically takes care of any conversion based on the file ending.

Color Spaces

In OpenCV images are not stored using the conventional RGB color channels; they are stored in BGR order (the reverse order). When reading an image, the default is BGR; however, there are several conversions available. Color space conversions are done using the function cvtColor().

For example, converting to grayscale is done like this:

```
im = cv2.imread('empire.jpg')
gray = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)
```

After the source image, there is an OpenCV color conversion code. Some of the most useful conversion codes are:

cv2.COLOR_BGR2GRAY
cv2.COLOR_BGR2RGB
cv2.COLOR_GRAY2BGR

In each of these, the number of color channels for resulting images will match the conversion code (single channel for gray and three channels for RGB and BGR). The last version converts grayscale images to BGR and is useful if you want to plot or overlay colored objects on the images.

Displaying Images and Results

To display images and results in OpenCV, you can use the imshow() function. Here's an example:

```
import cv2
image = cv2.imread('image.jpg')
cv2.imshow('Image', image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

In this example, imshow() is used to display the image with a window titled "Image". The waitKey(0) function will wait for a key press before closing the window. Finally, destroyAllWindows() is called to close all OpenCV windows.

You can also display intermediate results or processed images in the same way by calling imshow() before waitKey(). Just make sure to provide a different window title for each image to avoid overwriting the previous one.

Note: Draw a image for the above topic

6. Processing Video:

Video with pure Python is hard. There are speed, codecs, cameras, operating systems, and file formats to consider. There is currently no video library for Python. OpenCV with its Python interface is the only good option. In this section, we'll look at some basic examples using video.

Video Input:

Reading video from a camera is very well supported in OpenCV. A basic complete example that captures frames and shows them in an OpenCV window looks like this:

```
import cv2
cap = cv2.VideoCapture(0)
while True:
        ret,im = cap.read()
        cv2.imshow('video test',im)
        key = cv2.waitKey(10)
        if key == 27:
                break
        if key == ord(' '):
                cv2.imwrite('vid_result.jpg',im)
```

The capture object VideoCapture captures video from cameras or files. Here we pass an integer at initialization. This is the id of the video device; with a single camera connected this is 0. The method read() decodes and returns the next video frame. The first value is a success flag and the second the actual image array. The waitKey() function waits for a key to be pressed and quits the application if the 'Esc' key (Ascii number 27) is pressed, or saves the frame if the 'space' key is pressed.

Let's extend this example with some simple processing by taking the camera input and showing a blurred (color) version of the input in an OpenCV window. This is only a slight modification to the base example above:

```
import cv2
cap = cv2.VideoCapture(0)
while True:…
        ret,im = cap.read()
        blur = cv2.GaussianBlur(im,(0,0),5)
        cv2.imshow('camera blur',blur)
        if cv2.waitKey(10) == 27:
                break
```

Each frame is passed to the function GaussianBlur(), which applies a Gaussian filter to the image. In this case, we are passing a color image so each color channel is blurred separately. The function takes a tuple for filter size and the standard deviation for the Gaussian function (in this case 5). If the filter size is set to zero, it will automatically be determined from the standard deviation.

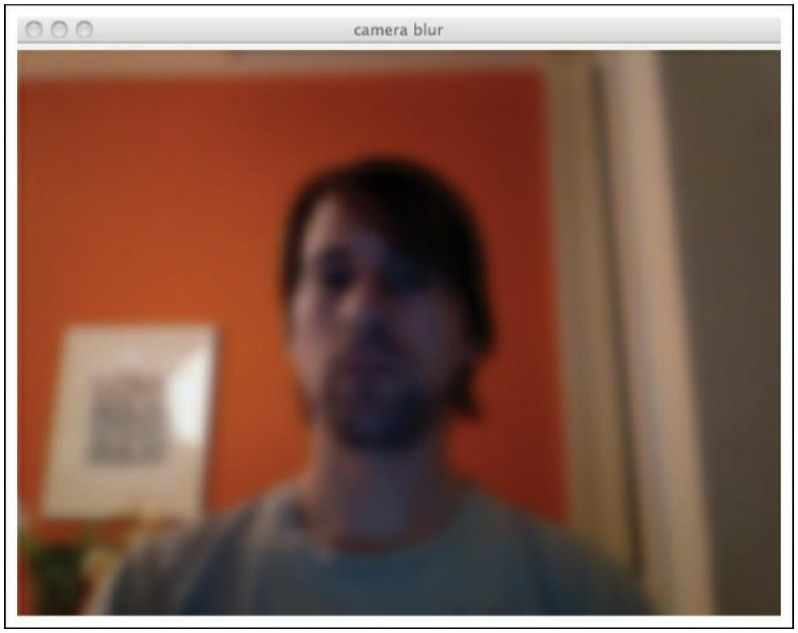


Figure 10-4. Screenshot of a blurred video of the author as he's writing this chapter.

Reading video from files works the same way but with the call to VideoCapture() taking the video filename as input:

```
capture = cv2.VideoCapture('filename')
```

Reading Video to NumPy Arrays

Using OpenCV, it is possible to read video frames from a file and convert them to NumPy arrays. Here is an example of capturing video from a camera and storing the frames in a NumPy array:

```
import cv2
cap = cv2.VideoCapture(0)
frames = []
while True:
        ret,im = cap.read()
        cv2.imshow('video',im)
        frames.append(im)
```

```
            if cv2.waitKey(10) == 27:
                    break
frames = array(frames)
print im.shape
print frames.shape
```

Each frame array is added to the end of a list until the capturing is stopped. The resulting array will have size (number of frames, height, width, 3). The printout confirms this:

```
(480, 640, 3)
(40, 480, 640, 3)
```

In this case, there were 40 frames recorded. Arrays with video data like this are useful for video processing, such as in computing frame differences and tracking.

7. Tracking:

Tracking is the process of following objects through a sequence of images or video. Optical flow, also known as optic flow, is the motion of objects within an image or between consecutive images. It is represented as a 2D vector field that describes the translation of pixels.

The optical flow technique relies on three key assumptions:

1. Brightness constancy: The pixel intensities of an object remain constant between consecutive images.
2. Temporal regularity: The time interval between frames is short enough to consider motion changes using differentials.
3. Spatial consistency: Adjacent pixels exhibit similar motion patterns.

Although these assumptions may not hold in all cases, they are effective for small motions and short time intervals.

The optical flow equation is derived by assuming that the intensity of a pixel at time t remains the same at time t + δt, after motion [δx, δy]:

$$\nabla \mathbf{I}^{\mathrm{T}}\mathbf{v} = -\mathbf{I}\mathbf{t},$$

Here, v = [u, v] represents the motion vector, ∇I^T is the gradient of the image intensity, and It is the time derivative. Solving this equation for individual points in the image is challenging due to the under-determined nature of the

problem. However, by enforcing spatial consistency, solutions can be obtained.

OpenCV provides various optical flow implementations, including CalcOpticalFlowBM(), CalcOpticalFlowHS(), calcOpticalFlowPyrLK(), and calcOpticalFlowFarneback(). These methods offer different approaches for estimating optical flow, with calcOpticalFlowFarneback() being one of the most effective for obtaining dense flow fields.

To use optical flow in OpenCV, you can apply the desired optical flow method to analyze video sequences and extract motion vectors. Each method has its own advantages and limitations.

Try running the following script:

```
import cv2
def draw_flow(im,flow,step=16):
        h,w = im.shape[:2]
        y,x = mgrid[step/2:h:step,step/2:w:step].reshape(2,-1)
        fx,fy = flow[y,x].T
        lines = vstack([x,y,x+fx,y+fy]).T.reshape(-1,2,2)
        lines = int32(lines)
        vis = cv2.cvtColor(im,cv2.COLOR_GRAY2BGR)
        for (x1,y1),(x2,y2) in lines:
        cv2.line(vis,(x1,y1),(x2,y2),(0,255,0),1)
        cv2.circle(vis,(x1,y1),1,(0,255,0), -1)
        return vis
cap = cv2.VideoCapture(0)
ret,im = cap.read()
prev_gray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)
while True:
        ret,im = cap.read()
        gray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)
        flow=cv2.calcOpticalFlowFarneback(prev_gray,gray,None,0.5,3,15,
        3,5,1.2,0)
        prev_gray = gray
        cv2.imshow('Optical flow',draw_flow(gray,flow))
if cv2.waitKey(10) == 27:
break
```

The Lucas-Kanade Algorithm

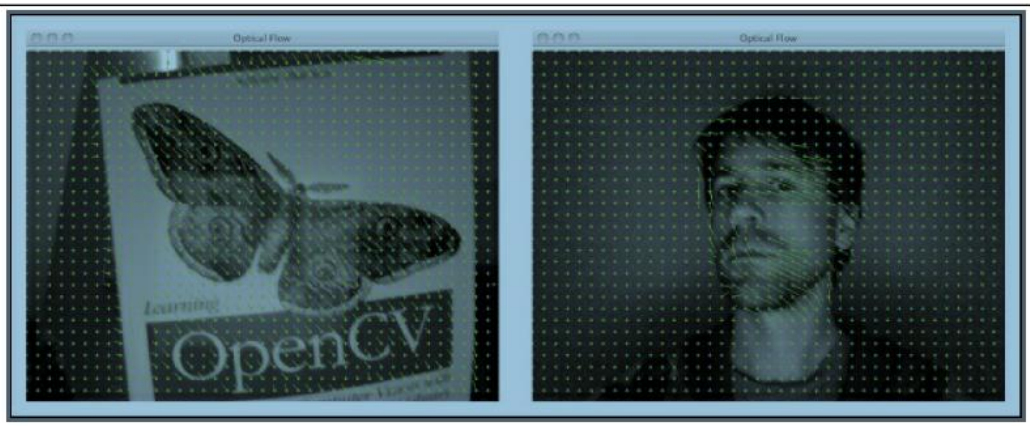


Figure 10-5. Optical flow vectors (sampled at every 16th pixel) shown on video of a translating book and a turning head.

The Lucas-Kanade algorithm is a widely used method for estimating optical flow. It is an iterative algorithm that assumes local motion consistency in a small neighborhood of pixels. The algorithm is named after the researchers Robert Lucas and Takeo Kanade who originally proposed it.

Here's a simplified explanation of the Lucas-Kanade algorithm:

1. Given two consecutive frames, convert them to grayscale.
2. Select a small window (e.g., a 3x3 or 5x5 neighborhood) around a pixel in the first frame.
3. Compute the spatial gradient (x and y derivatives) of the grayscale intensities in the first frame within the selected window.
4. Compute the temporal gradient (change in grayscale intensity) by subtracting the corresponding pixel intensity values between the first and second frames within the selected window.
5. Solve the optical flow equation for the motion vector (u, v) using the least squares method:

   A^T * A * [u, v] ^T = A^T * b

   where A is the matrix of spatial gradients and b is the vector of temporal gradients.
6. Repeat steps 2-5 for all pixels in the image or a subset of interest.
7. Optionally, apply a regularization term to account for noise and improve the stability of the estimates.
8. Visualize the estimated optical flow vectors on the image.

The Lucas-Kanade algorithm assumes that the motion within the local neighborhood is constant and that the pixel intensities do not change significantly between frames. These assumptions allow for estimating the motion using a linear approximation.

OpenCV provides the function cv2.calcOpticalFlowPyrLK() to implement the Lucas-Kanade algorithm efficiently, especially when dealing with larger motion and scale variations. This function performs pyramidal downsampling to estimate the motion at different scales and iteratively refines the results.

By applying the Lucas-Kanade algorithm, you can estimate optical flow and track objects or points of interest in a sequence of images or video.

Using tracker and Using generators:

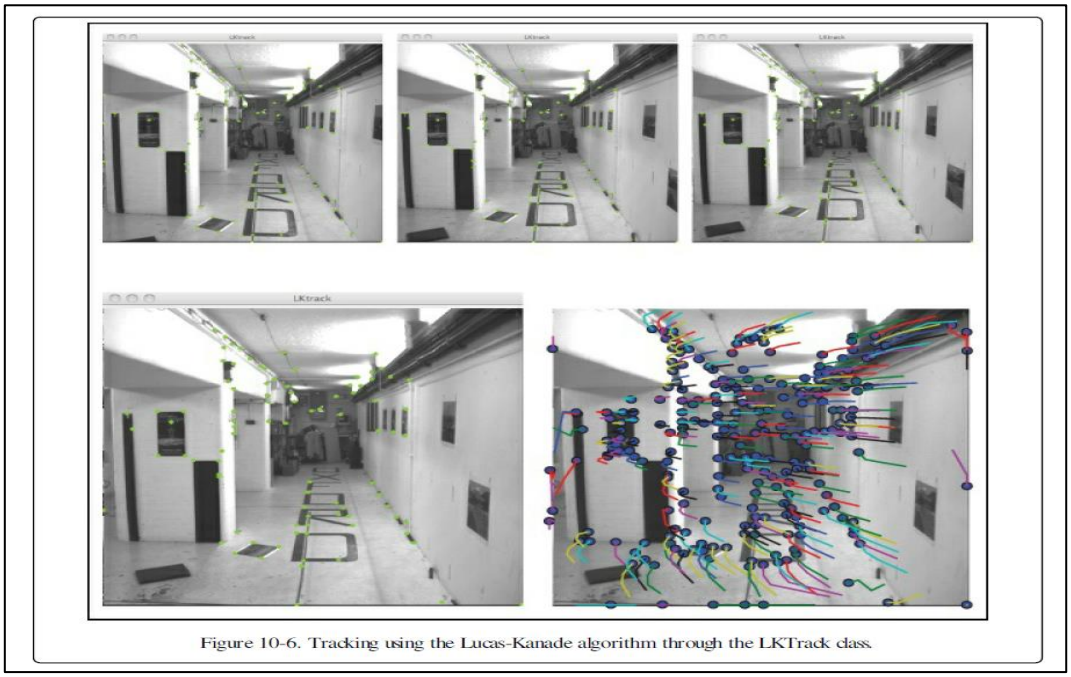


Figure 10-6. Tracking using the Lucas-Kanade algorithm through the LKTrack class.

This generator makes it really easy to use the tracker class and completely hides the OpenCV functions from the user.