Chapter 8 – Video Surveillance, Background Modelling, and Morphological Operations

* What is native background subtraction?
* What is frame differencing?
* How do we build a background model?
* How do we identify a new object in a static video?
* What is morphological image processing and how is it related to background modelling?
* How do we achieve different effects using morphological operators?

Background Subtraction

Background subtraction is very useful in video surveillance. Basically, the background subtraction technique performs really well in cases where we have to detect moving objects in a static scene. How is this useful for video surveillance? The process of video surveillance involves dealing with constant data flow. The data stream keeps coming in and we need to analyse it to recognize any suspicious activity.

Let’s consider the example of a hotel lobby. All the walls and furniture have a fixed location. If we build a background mode, we can use it to identify suspicious activity in the lobby. We are taking advantage of the fact that the background scene remains static. This helps us avoid any unnecessary computational overhead. As the name indicates, this algorithm works by detecting and assigning each pixel of an image to two classes, either the background (assumed static and stable) or the foreground, and subtracting it from the current frame to obtain the foreground image part, which includes moving objects such as persons, cars, and so on. With the static assumption, the foreground objects will naturally correspond to objects or people moving in front of the background.

In order to detect moving objects, we need to build a model of the background. This is not the same as direct frame differencing because we are actually modelling the background and using this model to detect moving objects. When we say that we are modelling the background, we are basically building a mathematical formula that can be used to represent the background. This is much better than the simple frame-differencing technique. This technique tries to detect static parts of the scene and then include small updates in the build statistic formula of the background model. This background model is then used to detect background pixels. So, it’s an adaptive technique that can adjust according to the scene.

HOWEVER, while the **naïve approach** does a reasonably good job of computing the shape of a given object under some constraints, the method is far from ideal. One of the main requirements of this approach is that the color and intensity of the object should be sufficiently different from that of the background. Some of the factors that affect this kind of algorithm are image noise, lighting conditions, and autofocus in cameras.

Once a new object enters our scene and stays there, it will be difficult to detect new objects that are in front of it. This is because we are not updating our background model and now the new object is a part of the background. In this approach, we assume that the background is static. If some parts of our background starts moving, those parts will start getting detected as new objects. So, even movements that are minor, say a waving flag, will cause problems in our detection algorithm. This approach is also sensitive to changes in illumination and it cannot handle any camera movement. Needless to say, it’s a delicate approach. We need something that can handle all these things in the real world.

**Frame Differencing**

We know that we cannot keep a static background image pattern that can be used to detect objects. One of the ways to fix this would be by using frame differencing. It is one of the simplest techniques we can use to see what parts of the video are moving. When we consider a live video stream, the differences between successive frames gives a lot of information. The concept is fairly straightforward – we just take the difference between successive frames and display the differences between them.

As you can see from the previous images, only the moving parts of the video get highlighted. This gives us a good starting point to see what areas are moving in the video. Let’s look at the function to compute the frame differences:

|  |
| --- |
| Mat frameDiff(Mat prevFrame, Mat curFrame, Mat nextFrame) {  Mat diffFrames 1, diffFrames2, output;  // Compute abs diff between current frame and the next frame  absdiff(nextFrame, curFrame, diffFrames1);  // Compute abs diff between current frame and the previous frame  absdiff(curFrame, prevFrame, diffFrames2);  // Bitwise “AND” operation between the previous two diff images  bitwise\_and(diffFrames1, diffFrames2, output);  return output; } |

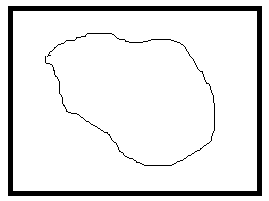
Frame differencing is fairly straightforward – you compute the absolute differences between the current frame and the previous frame, and between the current frame and the next frame. We then take these frame differences and apply a bitwise AND operator. This will highlight the moving parts in the image. If you just compute the difference between the current frame and the previous frame, it tends to be noisy. Hence, we need to use the bitwise AND operator between successive frame differences to get some stability when we see the moving objects.

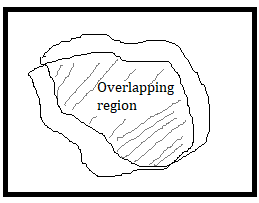
Let’s look at the function that can extract and return a frame from the webcam:

|  |
| --- |
| Mat getFrame (VideoCapture cap, float scalingFactor) {  Mat frame, output;  // Capture the current frame  cap >> frame;  // Resize the frame  resize(frame, frame, Size(), scalingFactor, scalingFactor, INTER\_AREA);  // Convert to grayscale  cvtColor(frame, output, COLOR\_BGR2GRAY);  return output; } |

As we can see, it’s pretty straightforward. We just need to resize the frame and convert it to grayscale. Now we have the helper functions ready, let’s look at the main function and see how it all comes together:

|  |
| --- |
| int main(int argc, char\* argv[])  {  Mat frame, prevFrame, curFrame, nextFrame;  char ch;  // Create the capture object  // 0 -> input arg that specifies it should take the input from the webcam  VideoCapture cap(0):  // If you cannot open the webcam, stop the execution  if(!cap.isOpened())  return -1;  // Create GUI windows  namedWindow(“Frame”);  // Scaling factor to resize the input frames from the webcam  float scalingFactor = 0.75;  prevFrame = getFrame(cap, scalingFactor);  curFrame = getFrame(cap, scalingFactor);  nextFrame = getFrame(cap, scalingFactor);  // Iterate until the user Escapes  while (true)  {  // Show the object movement  imshow(“Object Movement”, frameDiff(prevFrame, curFrame, nextFrame));  // Update the variables and grab the next frame  prevFrame = curFrame;  curFrame = nextFrame;  nextFrame = getFrame(cap, scalingFactor);  // Gey keyboard input and check if it’s ‘Esc’  Ch = waitKey(30);  if (ch == 27) {  break;  }  }  // Release video capture object  cap.release();   // Close all windows  destroyAllWindows();  return 1;  } |

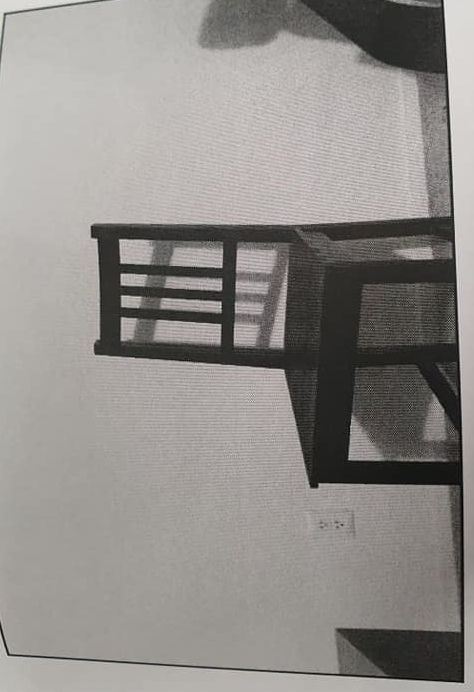
As we can see, frame differencing addresses a couple of important problems we faced earlier. It can quickly adapt to lighting changes or camera movement. If an object comes in to the frame and stays there, it will not be detected in future frames. One of the main concerns of this approach is about detecting uniformly coloured objects. If can only detect the edges of a uniformly coloured object. The reason is that a large portion of this object will result in very low pixel differences:  


Let’s say that this object moved slightly. If we compare this with the previous frame, it will look like this:  


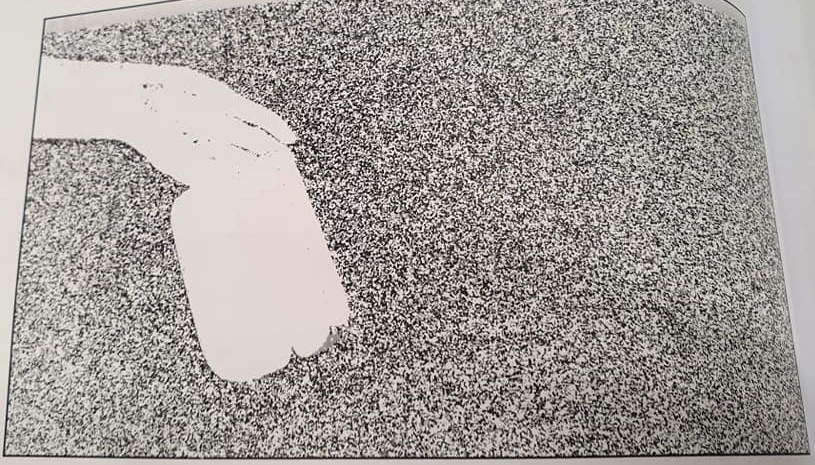
Hence, we have very few pixels that are labelled on that object. Another concern is that it is difficult to detect whether an object is moving towards the camera or away from it.

**The Mixture of Gaussians approach**

Before we talk about Mixture of Gaussians (MOG), let’s see what a mixture model is. A mixture model is just a statistical model that can be used to represent the presence of subpopulations within our data. We don’t really care about what category each data point belongs to. All we need is to identify that the data has multiple groups inside it. If we represent each subpopulation using the Gaussian function, then it’s called Mixture of Gaussians.  
Consider the following photograph:



Now, as we gather more frames in this scene, every part of the image will gradually become a part of the background model. This is what we discussed earlier in the Frame Differencing section as well. If a scene is static, the model adapts itself to make sure the background model is updated. The foreground mask, which is supposed to represent the foreground object, looks like a black image at this point because every pixel is part of the background model.

Let’s wait for some time and then introduce a new object into the scene. Let’s look at what the new foreground mask looks like using the MOG2 approach:  


As you can see, the new objects are being identified correctly. Let’s look at the interesting part of the code.

|  |
| --- |
| int main(int argc, char\* argv[])  {  // Variable declaration and initialization  …  // Iterate until the user presses “Esc”  while (true)  {  // Capture the current frame  cap >> frame;  // Resize the frame  resize(frame, frame, Size(), scalingFactor, scalingFactor, INTER\_AREA);  // Update the MOG2 background model based on the current frame  pMOG2->apply(frame, fgMaskMOG2);  // Show the MOG2 foreground mask  imshow(“FG Mask MOG 2”, fgMaskMOG2);    // Get the keyboard input and check if it’s “Esc”  ch = waitKey (30);  if (ch == 27) {  break;  }  }  // Release video capture object  cap.release();   // Close all windows  destroyAllWindows();  return 1;  } |

Let’s quickly go through the code and see what’s happening there. We use the Mixture of Gaussians model to create a background subtractor object. This object represents the model that will be updated as and when we encounter new frames from the webcam. We initialized two background subtraction models – BackgroundSubtractorMOG and BackgroundSubtractorMOG2. They represent two different algorithms that are used for background subtraction.

We start an infinite while loop and continuously read the input frames from the webcam. With each frame, we update the background model, as indicated by the following lines:

pMOG2->apply(frame, fgMaskMOG2);

The background model gets updated in these steps. Now, if a new object enters the scene and stays there, it will become part of the background model. This helps us overcome one of the biggest shortcomings of the naïve background subtraction model.

FURTHER READING:  
1. An Improved Adaptive Background Mixture Model for Real-time Tracking with Shadow Detection (P. KadewTraKuPong and R. Bowden)  
2. Improved Adaptive Gaussian Mixture Model for Background Subtraction  
(Z. Zivkovic)

**MORPHOLOGICAL IMAGE PROCESSING**

1. Erosion   
2. Dilation  
3. Opening  
4. Closing  
5. Morphological Gradient – Difference between the dilation and erosion of an image  
6. Top Hat transform – Difference between input image and morphological opening  
7. Black Hat transform – Difference between closing of an image and the input image itself

Chapter 9 - Learning Object Tracking

* How to track objects of a specific color
* How to build an interactive object tracker
* What a corner detector is
* How to detect good features to track
* How to build an optical flow-based feature tracker

Tracking objects of a specific color

In order to build a good object tracker, we need to understand what characteristics can be used to make our tracking robust and accurate. So, let’s take a baby step in that direction and see whether we can use colorspace information to come up with a good visual tracker.

There are many different colorspaces, and picking a good one will depend on the different applications that a user is using. While RGB is the native representation on a computer screen, it’s not necessarily ideal for humans. When it comes to humans, we give names to colors more naturally based on their hue, which is why hue saturation value (HSV) is probably one of the most informative colorspaces. It closely aligns with how we perceive colors. Hue refers to the color spectrum, saturation refers to the intensity of a particular color, and value refers to the brightness of that pixel. This is actually represented in a cylindrical format.

We take the pixels of an image to the HSV colorspace and then use this colorspace to measure distances in this colorspace and threshold in this space thresholding to track a given object.

Consider the following example, where a tracker recognizes a particular object in the video based on the color characteristics. In order to use this tracker, we need to know the color distribution of our target object. Here is the code to track a colored object, which selects only pixels that have a certain given hue.

|  |
| --- |
| int main(int argc, char\* argv[])  {  // Variable declaration and initialization  …  // Iterate until the user presses “Esc”  while (true)  {  // Initialize the output image before each iteration  outputImage = Scalar(0,0,0);  // Capture the current frame  cap >> frame;  // Check if ‘frame’ is empty  If (frame.empty())  break;  // Resize the frame  resize(frame, frame, Size(), scalingFactor, scalingFactor, INTER\_AREA);    // Convert to HSV colorspace  cvtColor(frame, hsvImage, COLOR\_BGR2HSV);  // Define the range of “blue” color in HSV colorspace  Scalar lowerLimit = Scalar(60,100,100);  Scalar upperLimit = Scalar(180,255,255);  // Threshold the HSV image to get only blue color  inRange(hsvImage, lowerLimit, upperLimit, mask);  // Compute bitwise-AND of input image and mask  bitwise\_and(frame, frame, outputImage, mask=mask);  // Run median filter on the output to smoothen it  medianBlur(outputImage, outputImage, 5);  // Display the input and output image  imshow(“Input”, frame);  Imshow(“Output”, outputImage);    // Get the keyboard input and check if it’s “Esc”  ch = waitKey (30);  if (ch == 27) {  break;  }  }  return 1;  } |

**Building an interactive object tracker**

A colorspace-based tracker gives us the freedom to track a colored object, but we are also constrained to a predefined color. What if we want to just pick an object at random? How do we build an object tracker that can learn the characteristics of the selected object and just track it automatically? This is where the continuously-adaptive meanshift (CAMShift) algorithm comes into the picture. It’s basically an improved version of the meanshift algorithm.

The concept of meanshift is actually nice and simple. Let’s say we select a region of interest and we want our object tracker to track that object. In this region, we select a bunch of points based on the color histogram and we compute the centroid of spatial points. If the centroid lies at the center of this region, we know that the object hasn’t moved. But if the centroid is not at the center of this region, then we know that the object is moving in some direction. The movement of the centroid controls the direction in which the object is moving. So, we move the bounding box of the object to a new location so that the new centroid becomes the center of this bounding box. Hence, this algorithm is called meanshift, because the mean (centroid) is shifting. This way, we keep ourselves updated with the current location of the object.

But the problem with the meanshift is that the size of the bounding box is not allowed to change. When you move the object away from the camera, the object will appear smaller to the human eye, but meanshift will not take that into account. The size of the bounding box will remain the same throughout the tracking session. Hence, we need to use CAMShift. The advantage of CAMShift is that it can adapt the size of the bounding box to the size of the object. Along with that, it can also keep track of the orientation of the object.

How to use the OpenCV CamShift algorithm - <https://www.youtube.com/watch?v=T3e5z6qoCpA>

Consider the video where the object of interest is highlighted. Now that we have selected the object, the algorithm computes the histogram backprojection and extracts all the information. What is histogram backprojection? It’s just a way of identifying how well the image fits into our histogram model. We compute the histogram model of a particular thing and then use this model to find that thing in an image. The tracking persists even though the object is moved or orientation is changed. The bounding ellipse has changed the aspect ratio to reflect the fact that the object looks skewed now (because of the perspective transformation).

Let’s look at the user interface functionality in the code:

|  |
| --- |
| Mat image;  Point originPoint;  Rect selectedRect;  bool selectRegion = false;  int trackingFlag = 0;  // Function to track the mouse events  void onMouse(int event, int x, int y, int, void\*)  {  if (selectRegion)  {  selectedRect.x = MIN(x, originPoint.x);  selectedRect.y = MIN(y, originPoint.y);  selectedRect.width = std::abs(x – originPoint.x);  selectedRect.height = std::abs(y – originPoint.y);  selectedRect &= Rect(0, 0, image.cols, image.rows);  }  switch(event)  {  case EVENT\_LBUTTONDOWN:  originPoint = Point(x,y,);  selectedRect = Rect(x,y,0,0);  selectRegion = true;  break;  case EVENT\_LBUTTONUP:  selectRegion = false;  if (selectedRect.width > 0 && selectedRect.height > 0 )  {  trackingFlag = -1;  }  break;  }  } |

The function basically captures the coordinates of the rectangle that was selected in the window. The user just needs to click and drag with the mouse. There are a set of built-in functions in OpenCV that help us to detect these different mouse events.

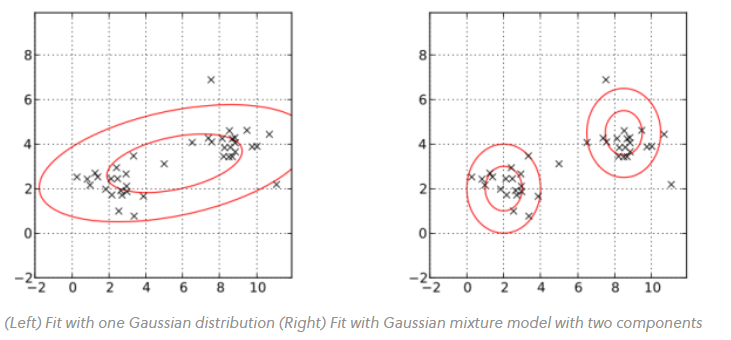
Here is the code for performing object tracking based on CAMShift:

|  |
| --- |
| int main(int argc, char\* argv[])  {  // Variable declaration and initialization  …  // Iterate until the user presses “Esc”  while (true)  {  // Capture the current frame  cap >> frame;  // Check if ‘frame’ is empty  if (frame.empty())  break;  // Resize the frame  resize(frame, frame, Size(), scalingFactor, scalingFactor, INTER\_AREA);  // Clone the input frame  frame.copyTo(image);  // Convert to HSV colorspace  cvtColor(image, hsvImage, COLOR\_BGR2HSV);  if (trackingFlag)  {  // Check for all the values in ‘hsvimage’ that are within the specified range  // and put the result in ‘mask’  inRange(hsvImage, Scalar(0, minSaturation, minValue), Scalar(180, 256, maxValue), mask);  // Mix the specified channels  int channels[] = {0,0};  hueImage.create(hsvImage.size(), hsvImage.depth());  mixChannels(&hsvImage, 1, &hueImage, 1, channels, 1);  if (trackingFlag < 0)  {  // Create images based on selected regions of interest  Mat roi(hueImage, selectedRect), maskroi (mask, selectedRect);  // Compute the histogram and normalize it  calcHist(&roi, 1, 0, maskroi, hist, 1, &histSize, &histRanges);  normalize(hist, hist, 0, 255, NORM\_MINMAX);  trackingRect = selectedRect;  trackingFlag = 1;  }    // Compute histogram backprojection  calcBackProject(&hueImage, 1, 0, hist, backproj, &histRanges);  backproj &= mask;  RotatedRect rotatedTrackingRect = CamShift(backproj, trackingRect, TermCriteria(TermCriteria::EPS | TermCriteria::Cout, 10, 1));    // Check if area of trackingRect is too small  if(trackingRect.area() <= 1)  {  // Use an offset value to make sure the trackingRect has a minimum size  int cols = backproj.cols, rows = backproj.rows;  int offset = MIN(rows, cols) + 1;  trackingRect = Rect(trackingRect.x – offset, trackingRect.y – offset, trackingRect.x + offset, trackingRect.y + offset) & Rect(0,0,cols,rows);  }  // Draw the ellipse on top of the image  ellipse(image, rotatedTrackingRect, Scalar(0,255,0), 3, LINE\_AA);  // Apply the ‘negative’ effect on the selected ROI  if (selectedRegion && selectedRect.width > 0 && selectedRect.height > 0)  {  Mat roi(image, selectedRect);  bitwise\_not(roi, roi);  }  // Display the output image  imshow(windowName, image);    // Get the keyboard input and check if it’s “Esc”  ch = waitKey (30);  if (ch == 27) {  break;  }  }  return 1;  } |

Gaussian Mixture Models

**Gaussian mixture models** are a probabilistic model for representing [normally distributed](https://brilliant.org/wiki/multivariate-normal-distribution/) subpopulations within an overall population. [Mixture models](https://brilliant.org/wiki/mixture-model/) in general don't require knowing which subpopulation a data point belongs to, allowing the model to learn the subpopulations automatically. Since subpopulation assignment is not known, this constitutes a form of [unsupervised learning](https://brilliant.org/wiki/unsupervised-learning/).

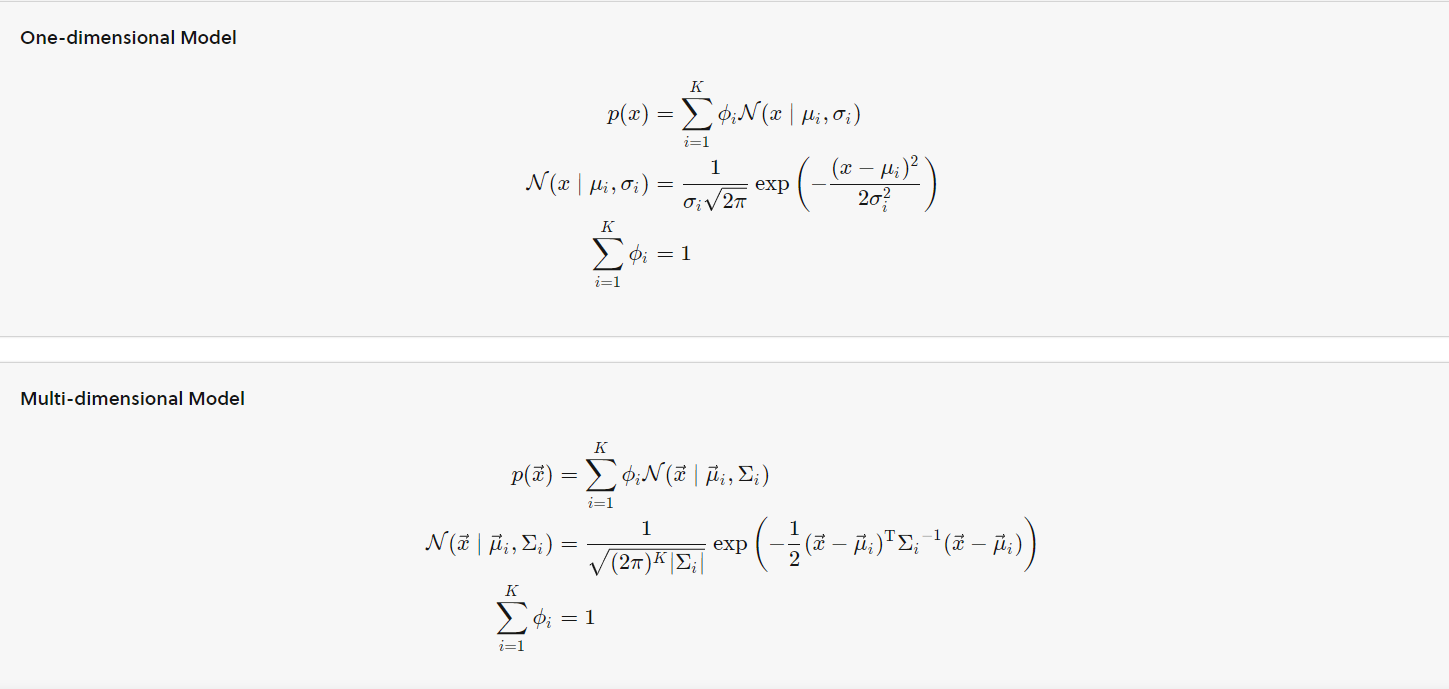
One hint that data might follow a mixture model is that the data looks [multimodal](https://brilliant.org/wiki/data-mode/), i.e. there is more than one "peak" in the distribution of data. Trying to fit a multimodal distribution with a [unimodal](https://brilliant.org/wiki/data-mode/) (one "peak") model will generally give a poor fit, as shown in the example below. Since many simple distributions are unimodal, an obvious way to model a multimodal distribution would be to assume that it is generated by multiple unimodal distributions. For several [theoretical reasons](https://brilliant.org/wiki/central-limit-theorem/), the most commonly used distribution in modeling real-world unimodal data is the Gaussian distribution. Thus, modeling multimodal data as a mixture of many unimodal Gaussian distributions makes intuitive sense. Furthermore, GMMs maintain many of the theoretical and computational benefits of Gaussian models, making them practical for efficiently modeling very large datasets.



A Gaussian mixture model is parameterized by two types of values, the mixture **component weights** and the component **means** and **variances/covariances**.

For a Gaussian mixture model with *K* components, the *k*th component has a mean of *μk*​ and variance of *σk*​ for the [univariate case](https://brilliant.org/wiki/normal-distribution/) and a mean of *μ*​*k*​ and covariance matrix of Σ*k*​ for the [multivariate case](https://brilliant.org/wiki/multivariate-normal-distribution/).

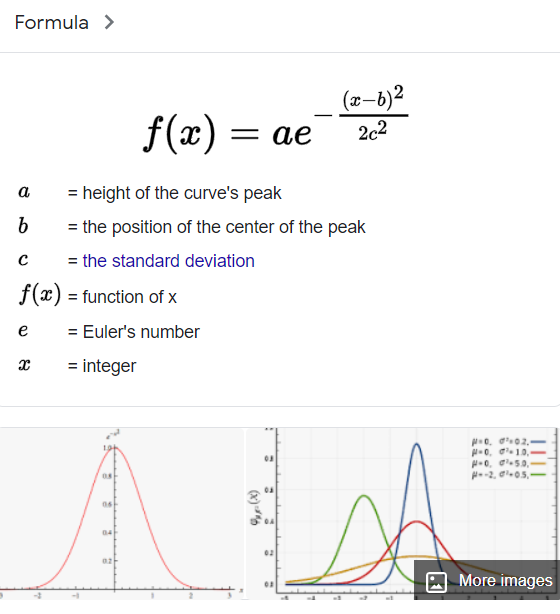
The mixture component weights are defined as *ϕk*​ for component *Ck*​, with the constraint that ∑*i*=1*K*​*ϕi*​=1 so that the total probability distribution normalizes to 1. If the component weights aren't learned, they can be viewed as an [a-priori](https://brilliant.org/wiki/a-priori/) distribution over components such that *p*(*x* generated by component *Ck*​)=*ϕk*​. If they are instead learned, they are the [a-posteriori](https://brilliant.org/wiki/a-posterior/) estimates of the component probabilities given the data.



PROBLEM STATEMENT: For a set of given customers, if we know how much money they withdraw or deposit into it this month, let’s see if we can predict if they will continue using the app next month or not.

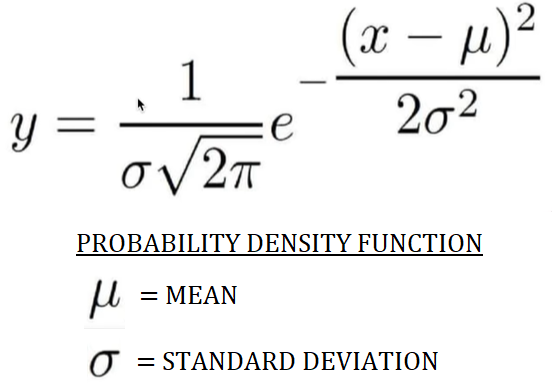
An universally used generative unsupervised clustering is Gaussian Mixture Model (GMM) which is also known as “EM Clustering”. The idea of GMM is very simple – for a given dataset, each point is generated by linearly combining multiple multivariate Gaussians.

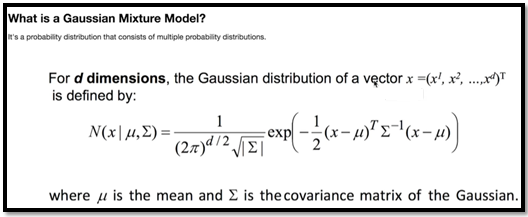
GAUSSIAN   
- Being or having the shape of a normal curve or a normal distribution.  
- A distribution that lists the outcomes of an experiment and the probability associated with each outcome

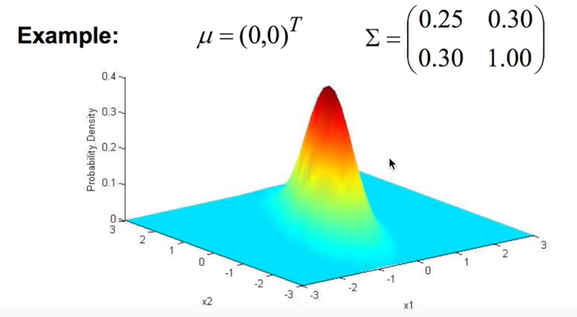


PROBABILITY DENSITY FUNCTION

The formula for a Gaussian distribution. It is a function of a continuous random variable, whose integral across an interval gives the probability that the value of the variable lies within the same interval.



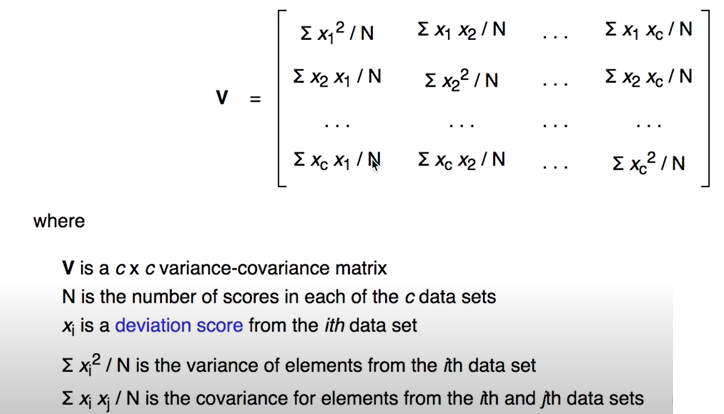
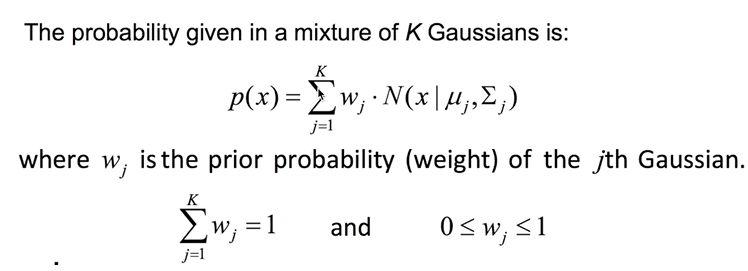


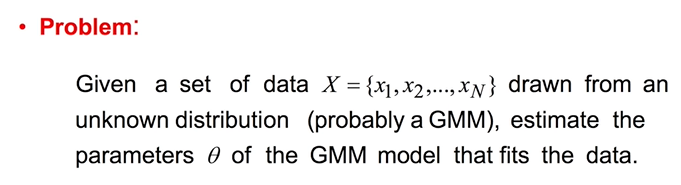


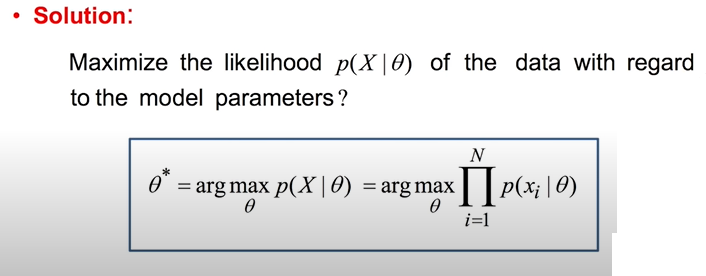
Covariance is a measure of how changes in one variable are associated with changes in a second variable. Specifically, covariance measures the degree to which two variables are linearly associated. However, it is also often used informally as a general measure of how monotonically related two variables are.

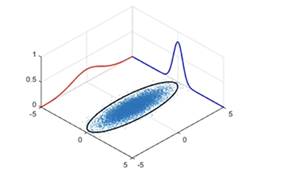
Variance-Covariance matrix

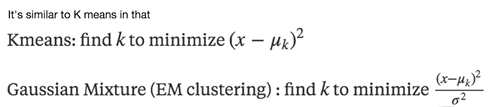
Variance and covariance are often displayed together in a variance-covariance matrix. The variances appear along the diagonal and covariances appear in the off-diagonal elements.









The difference is the denominator, which means GM takes variance into consideration when it calculates the measurement.

K-means only calculate conventional Euclidean distance.

In other words, K-means calculate distance, while GM calculates “weighted” distance.

How is it optimized?

One of the most popular approaches to maximize the likelihood is to use the Expectation-Maximization (EM) algorithm.

Basic idea of the EM algorithm:

* Introduce a hidden variable such that its knowledge would simplify the maximization of the likelihood
* At each iteration:  
  E-Step: Estimate the distribution of the hidden variable given the data and the current value of the parameters

M-Step: Maximize the joint distribution of the data and the hidden variable.