**Ba thuật toán ko sử dụng do sai số quá lớn trong việc nhận diện hình ảnh**

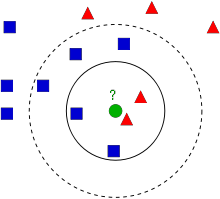
**Understanding k-Nearest Neighbour**

**Goal**

In this chapter, we will understand the concepts of k-Nearest Neighbour (kNN) algorithm.

**Theory**

kNN is one of the simplest of classification algorithms available for supervised learning. The idea is to search for closest match of the test data in feature space. We will look into it with below image.



In the image, there are two families, *Blue Squares and Red Triangles*. We call each family as **Class**. Their houses are shown in their town map which we call *feature space*. *(You can consider a feature space as a space where all datas are projected. For example, consider a 2D coordinate space. Each data has two features, x and y coordinates. You can represent this data in your 2D coordinate space, right? Now imagine if there are three features, you need 3D space. Now consider N features, where you need N-dimensional space, right? This N-dimensional space is its feature space. In our image, you can consider it as a 2D case with two features)*.

Now a new member comes into the town and creates a new home, which is shown as green circle. He should be added to one of these Blue/Red families. We call that process, **Classification**. What we do? Since we are dealing with kNN, let us apply this algorithm.

One method is to check who is his nearest neighbour. From the image, it is clear it is the Red Triangle family. So he is also added into Red Triangle. This method is called simply **Nearest Neighbour**, because classification depends only on the nearest neighbour.

But there is a problem with that. Red Triangle may be the nearest. But what if there are lot of Blue Squares near to him? Then Blue Squares have more strength in that locality than Red Triangle. So just checking nearest one is not sufficient. Instead we check some *k* nearest families. Then whoever is majority in them, the new guy belongs to that family. In our image, let’s take *k=3*, ie 3 nearest families. He has two Red and one Blue (there are two Blues equidistant, but since k=3, we take only one of them), so again he should be added to Red family. But what if we take *k=7*? Then he has 5 Blue families and 2 Red families. Great!! Now he should be added to Blue family. So it all changes with value of k. More funny thing is, what if *k = 4*? He has 2 Red and 2 Blue neighbours. It is a tie !!! So better take k as an odd number. So this method is called **k-Nearest Neighbour** since classification depends on k nearest neighbours.

Again, in kNN, it is true we are considering k neighbours, but we are giving equal importance to all, right? Is it justice? For example, take the case of *k=4*. We told it is a tie. But see, the 2 Red families are more closer to him than the other 2 Blue families. So he is more eligible to be added to Red. So how do we mathematically explain that? We give some weights to each family depending on their distance to the new-comer. For those who are near to him get higher weights while those are far away get lower weights. Then we add total weights of each family separately. Whoever gets highest total weights, new-comer goes to that family. This is called **modified kNN**.

So what are some important things you see here?

* You need to have information about all the houses in town, right? Because, we have to check the distance from new-comer to all the existing houses to find the nearest neighbour. If there are plenty of houses and families, it takes lots of memory, and more time for calculation also.
* There is almost zero time for any kind of training or preparation.

Now let’s see it in OpenCV.

**kNN in OpenCV**

We will do a simple example here, with two families (classes), just like above. Then in the next chapter, we will do much more better example.

So here, we label the Red family as **Class-0** (so denoted by 0) and Blue family as **Class-1** (denoted by 1). We create 25 families or 25 training data, and label them either Class-0 or Class-1. We do all these with the help of Random Number Generator in Numpy.

Then we plot it with the help of Matplotlib. Red families are shown as Red Triangles and Blue families are shown as Blue Squares.

import cv2

import numpy as np

import matplotlib.pyplot as plt

# Feature set containing (x,y) values of 25 known/training data

trainData = np.random.randint(0,100,(25,2)).astype(np.float32)

# Labels each one either Red or Blue with numbers 0 and 1

responses = np.random.randint(0,2,(25,1)).astype(np.float32)

# Take Red families and plot them

red = trainData[responses.ravel()==0]

plt.scatter(red[:,0],red[:,1],80,'r','^')

# Take Blue families and plot them

blue = trainData[responses.ravel()==1]

plt.scatter(blue[:,0],blue[:,1],80,'b','s')

plt.show()

You will get something similar to our first image. Since you are using random number generator, you will be getting different data each time you run the code.

Next initiate the kNN algorithm and pass the *trainData* and *responses* to train the kNN (It constructs a search tree).

Then we will bring one new-comer and classify him to a family with the help of kNN in OpenCV. Before going to kNN, we need to know something on our test data (data of new comers). Our data should be a floating point array with size number \; of \; testdata \times number \; of \; features. Then we find the nearest neighbours of new-comer. We can specify how many neighbours we want. It returns:

1. The label given to new-comer depending upon the kNN theory we saw earlier. If you want Nearest Neighbour algorithm, just specify *k=1* where k is the number of neighbours.
2. The labels of k-Nearest Neighbours.
3. Corresponding distances from new-comer to each nearest neighbour.

So let’s see how it works. New comer is marked in green color.

newcomer = np.random.randint(0,100,(1,2)).astype(np.float32)

plt.scatter(newcomer[:,0],newcomer[:,1],80,'g','o')

knn = cv2.KNearest()

knn.train(trainData,responses)

ret, results, neighbours ,dist = knn.find\_nearest(newcomer, 3)

print "result: ", results,"\n"

print "neighbours: ", neighbours,"\n"

print "distance: ", dist

plt.show()

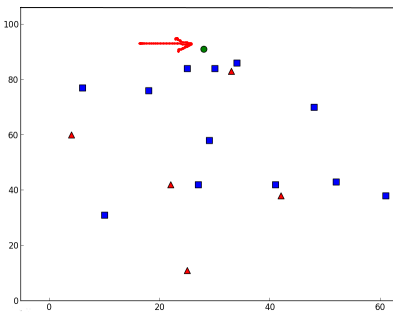
I got the result as follows:

result: [[ 1.]]

neighbours: [[ 1. 1. 1.]]

distance: [[ 53. 58. 61.]]

It says our new-comer got 3 neighbours, all from Blue family. Therefore, he is labelled as Blue family. It is obvious from plot below:



If you have large number of data, you can just pass it as array. Corresponding results are also obtained as arrays.

# 10 new comers

newcomers = np.random.randint(0,100,(10,2)).astype(np.float32)

ret, results,neighbours,dist = knn.find\_nearest(newcomer, 3)

# The results also will contain 10 labels.

# Understanding SVM

## Goal

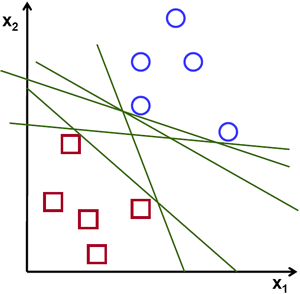
**In this chapter**

* We will see an intuitive understanding of SVM

## Theory

### Linearly Separable Data

Consider the image below which has two types of data, red and blue. In kNN, for a test data, we used to measure its distance to all the training samples and take the one with minimum distance. It takes plenty of time to measure all the distances and plenty of memory to store all the training-samples. But considering the data given in image, should we need that much?



Consider another idea. We find a line, f(x)=ax_1+bx_2+c which divides both the data to two regions. When we get a new test\_data X, just substitute it in f(x). If f(X) > 0, it belongs to blue group, else it belongs to red group. We can call this line as **Decision Boundary**. It is very simple and memory-efficient. Such data which can be divided into two with a straight line (or hyperplanes in higher dimensions) is called **Linear Separable**.

So in above image, you can see plenty of such lines are possible. Which one we will take? Very intuitively we can say that the line should be passing as far as possible from all the points. Why? Because there can be noise in the incoming data. This data should not affect the classification accuracy. So taking a farthest line will provide more immunity against noise. So what SVM does is to find a straight line (or hyperplane) with largest minimum distance to the training samples. See the bold line in below image passing through the center.



So to find this Decision Boundary, you need training data. Do you need all? NO. Just the ones which are close to the opposite group are sufficient. In our image, they are the one blue filled circle and two red filled squares. We can call them **Support Vectors** and the lines passing through them are called **Support Planes**. They are adequate for finding our decision boundary. We need not worry about all the data. It helps in data reduction.

What happened is, first two hyperplanes are found which best represents the data. For eg, blue data is represented by w^Tx+b_0 > 1while red data is represented by w^Tx+b_0 < -1 where w is **weight vector** ( w=[w_1, w_2,..., w_n]) and x is the feature vector (x = [x_1,x_2,..., x_n]). b_0 is the **bias**. Weight vector decides the orientation of decision boundary while bias point decides its location. Now decision boundary is defined to be midway between these hyperplanes, so expressed as w^Tx+b_0 = 0. The minimum distance from support vector to the decision boundary is given by, distance_{support \, vectors}=\frac{1}{||w||}. Margin is twice this distance, and we need to maximize this margin. i.e. we need to minimize a new function L(w, b_0) with some constraints which can expressed below:

\min_{w, b_0} L(w, b_0) = \frac{1}{2}||w||^2 \; \text{subject to} \; t_i(w^Tx+b_0) \geq 1 \; \forall i

where t_i is the label of each class, t_i \in [-1,1].

### Non-Linearly Separable Data

Consider some data which can’t be divided into two with a straight line. For example, consider an one-dimensional data where ‘X’ is at -3 & +3 and ‘O’ is at -1 & +1. Clearly it is not linearly separable. But there are methods to solve these kinds of problems. If we can map this data set with a function, f(x) = x^2, we get ‘X’ at 9 and ‘O’ at 1 which are linear separable.

Otherwise we can convert this one-dimensional to two-dimensional data. We can use f(x)=(x,x^2) function to map this data. Then ‘X’ becomes (-3,9) and (3,9) while ‘O’ becomes (-1,1) and (1,1). This is also linear separable. In short, chance is more for a non-linear separable data in lower-dimensional space to become linear separable in higher-dimensional space.

In general, it is possible to map points in a d-dimensional space to some D-dimensional space (D>d) to check the possibility of linear separability. There is an idea which helps to compute the dot product in the high-dimensional (kernel) space by performing computations in the low-dimensional input (feature) space. We can illustrate with following example.

Consider two points in two-dimensional space, p=(p_1,p_2) and q=(q_1,q_2). Let \phi be a mapping function which maps a two-dimensional point to three-dimensional space as follows:

\phi (p) = (p_{1}^2,p_{2}^2,\sqrt{2} p_1 p_2)
\phi (q) = (q_{1}^2,q_{2}^2,\sqrt{2} q_1 q_2)

Let us define a kernel function K(p,q) which does a dot product between two points, shown below:

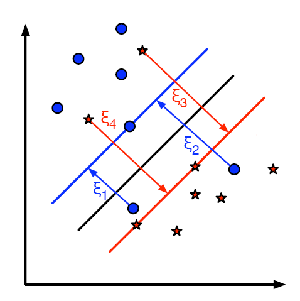
K(p,q)  = \phi(p).\phi(q) &= \phi(p)^T , \phi(q) \\
                          &= (p_{1}^2,p_{2}^2,\sqrt{2} p_1 p_2).(q_{1}^2,q_{2}^2,\sqrt{2} q_1 q_2) \\
                          &= p_{1}^2 q_{1}^2 + p_{2}^2 q_{2}^2 + 2 p_1 q_1 p_2 q_2 \\
                          &= (p_1 q_1 + p_2 q_2)^2 \\
          \phi(p).\phi(q) &= (p.q)^2

It means, a dot product in three-dimensional space can be achieved using squared dot product in two-dimensional space. This can be applied to higher dimensional space. So we can calculate higher dimensional features from lower dimensions itself. Once we map them, we get a higher dimensional space.

In addition to all these concepts, there comes the problem of misclassification. So just finding decision boundary with maximum margin is not sufficient. We need to consider the problem of misclassification errors also. Sometimes, it may be possible to find a decision boundary with less margin, but with reduced misclassification. Anyway we need to modify our model such that it should find decision boundary with maximum margin, but with less misclassification. The minimization criteria is modified as:

min \; ||w||^2 + C(distance \; of \; misclassified \; samples \; to \; their \; correct \; regions)

Below image shows this concept. For each sample of the training data a new parameter \xi_i is defined. It is the distance from its corresponding training sample to their correct decision region. For those who are not misclassified, they fall on their corresponding support planes, so their distance is zero.



So the new optimization problem is :

\min_{w, b_{0}} L(w,b_0) = ||w||^{2} + C \sum_{i} {\xi_{i}} \text{ subject to } y_{i}(w^{T} x_{i} + b_{0}) \geq 1 - \xi_{i} \text{ and } \xi_{i} \geq 0 \text{ } \forall i

How should the parameter C be chosen? It is obvious that the answer to this question depends on how the training data is distributed. Although there is no general answer, it is useful to take into account these rules:

* Large values of C give solutions with less misclassification errors but a smaller margin. Consider that in this case it is expensive to make misclassification errors. Since the aim of the optimization is to minimize the argument, few misclassifications errors are allowed.
* Small values of C give solutions with bigger margin and more classification errors. In this case the minimization does not consider that much the term of the sum so it focuses more on finding a hyperplane with big margin.

# Understanding K-Means Clustering

## Goal

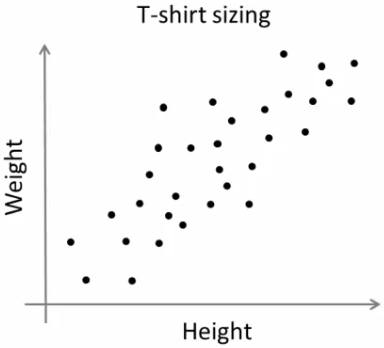
In this chapter, we will understand the concepts of K-Means Clustering, how it works etc.

## Theory

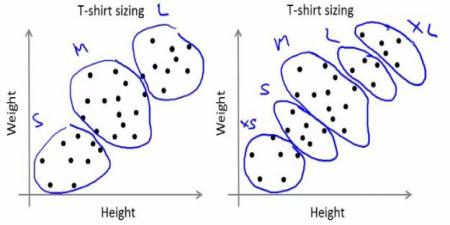
We will deal this with an example which is commonly used.

### T-shirt size problem

Consider a company, which is going to release a new model of T-shirt to market. Obviously they will have to manufacture models in different sizes to satisfy people of all sizes. So the company make a data of people’s height and weight, and plot them on to a graph, as below:



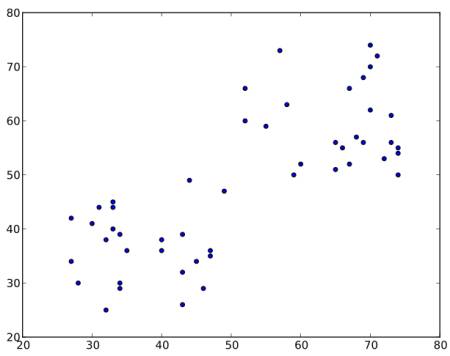
Company can’t create t-shirts with all the sizes. Instead, they divide people to Small, Medium and Large, and manufacture only these 3 models which will fit into all the people. This grouping of people into three groups can be done by k-means clustering, and algorithm provides us best 3 sizes, which will satisfy all the people. And if it doesn’t, company can divide people to more groups, may be five, and so on. Check image below :



### How does it work ?

This algorithm is an iterative process. We will explain it step-by-step with the help of images.

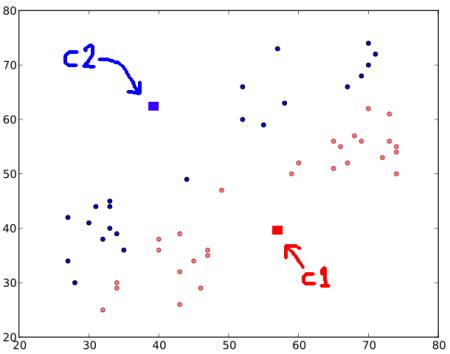
Consider a set of data as below ( You can consider it as t-shirt problem). We need to cluster this data into two groups.



**Step : 1** - Algorithm randomly chooses two centroids, C1 and C2(sometimes, any two data are taken as the centroids).

**Step : 2** - It calculates the distance from each point to both centroids. If a test data is more closer to C1, then that data is labelled with ‘0’. If it is closer to C2, then labelled as ‘1’ (If more centroids are there, labelled as ‘2’,‘3’ etc).

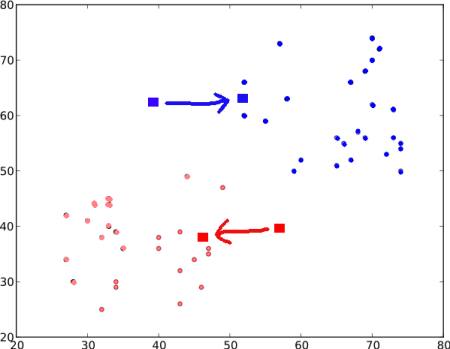
In our case, we will color all ‘0’ labelled with red, and ‘1’ labelled with blue. So we get following image after above operations.



**Step : 3** - Next we calculate the average of all blue points and red points separately and that will be our new centroids. That is C1 and C2 shift to newly calculated centroids. (Remember, the images shown are not true values and not to true scale, it is just for demonstration only).

And again, perform step 2 with new centroids and label data to ‘0’ and ‘1’.

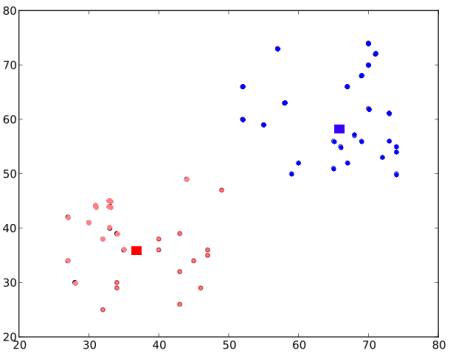
So we get result as below :



Now **Step - 2** and **Step - 3** are iterated until both centroids are converged to fixed points. (Or it may be stopped depending on the criteria we provide, like maximum number of iterations, or a specific accuracy is reached etc.) **These points are such that sum of distances between test data and their corresponding centroids are minimum**. Or simply, sum of distances between C1 \leftrightarrow Red\_Points and C2 \leftrightarrow Blue\_Points is minimum.

minimize \;\bigg[J = \sum_{All\: Red_Points}distance(C1,Red\_Point) + \sum_{All\: Blue\_Points}distance(C2,Blue\_Point)\bigg]

Final result almost looks like below :



So this is just an intuitive understanding of K-Means Clustering. For more details and mathematical explanation, please read any standard machine learning textbooks or check links in additional resources. It is just a top layer of K-Means clustering. There are a lot of modifications to this algorithm like, how to choose the initial centroids, how to speed up the iteration process etc.