

**Introduction to Machine Learning**  
**Prof. Sudeshna Sarkar**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Kharagpur**

**Lecture – 10**  
**k-Nearest Neighbour**

Good morning. Today we will start module 3. In this module, we will talk about instance based learning and then, we will talk about feature selection, will show instance base learning feature. Dimensionality is a problem and feature large having many features is a problem for many other learning algorithms and we will look for methods for feature selection.

In the first part of the lecture which we will have today, we will talk about instance base learning and specially the k nearest neighbor algorithm. So, what we have is that suppose in the machine learning in supervise learning, you have got training examples  $x, y$ .

(Refer Slide Time: 01:05)



A set of them  $x_i, y_i$  or you can say  $x_i, f(x_i)$ . So, these examples are given to you and given these examples, you want to come up with the function  $f$  and you want to find an estimate for the function  $f$ . In the previous module, we have seen how to learn linear

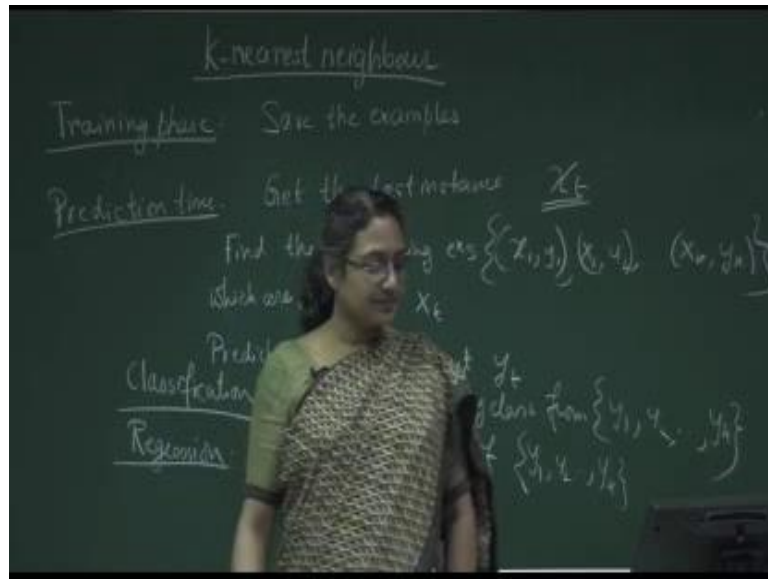
regression, a linear function  $f$  or a decision tree, a function which is a decision tree to estimate this  $f$ .

Today we will look at another setting called instance based learning, also called lazy learning. I mean the algorithm that we will discuss is a lazy algorithm; we will tell what it means. Instance based learning what we do is that when we get the training examples, we do not process them and learn a model instead. We just store the examples. When we need to classify an instance that time we do something. So, we do not immediately learn a model that is why the algorithm that we will discuss is also called a lazy algorithm that is the algorithm does not come up with the model a priori rather when it gets the test instance, it uses the stored instance in memory in order to find the possible  $y$ .

So, how does it do it? Suppose this is the instance phase and you have different points in the instance phase. For each point, you have the corresponding  $x$  value and the  $y$  value. When you are given a new instance, you find what the closest instance in terms of the  $x$  value is and suppose this is the closest you would find the  $y$  value of the instance and you guess the  $y$  value of this as the  $y$  value of this. This is the basic nearest neighbor algorithm. Now, in order to do this given a test instance, you have to find similar instance. So, you want to find most similar instance or we will see in some cases, we want to find some neighboring instance, the most the nearest instances.

Now, how to find the similarity or what is the distance function at the metric that we consider. For similarity we can use a standard metric like. Secondly, distance or other metrics like cosine and other depending on the type of data that you are looking for. So, the basic  $k$  nearest neighbor algorithm, the algorithm that I am going to describe is called the  $k$  nearest neighbor algorithm and let us outline that algorithm.

(Refer Slide Time: 04:38)



So, the  $k$  nearest neighbor works as follows. As we have seen earlier learning algorithm has two phases; training phase and testing phase or the use phase. So, in the training phase, normally the model is learned. As I have said for such lazy algorithms, we do not learn a model during training. So, in the training method phase, we just save the training examples.

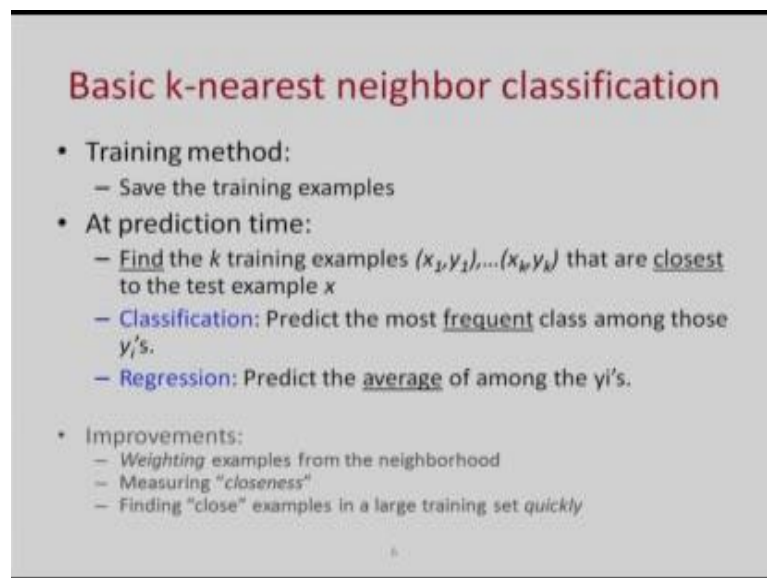
We just save the examples which is possible in a more advanced implementation to store the examples in some data structure, so that searching through this examples become faster, but we will talk about that later. So, basically we just store the instances and then a prediction time. So, this is training time. So, at prediction time, what we do is, we get a test instance and find  $k$  training examples. So, we get the test instance.

For the test instance, we are given only the  $x$  value. So, I given only  $x_t$  and we have to predict the corresponding  $y_t$ . So, what we do is that we find the training example  $x_1 y_1$  that is closest to  $x$ . So, among all the examples that we have stored in the training phase given  $x_t$ , we find that value of  $x_i$ . So, that it is closest to  $x_t$  and then, we predict  $y_t$  as the output. So, we predict, sorry not  $y_t y_1$  as the output  $y_t$ , this is the basic one nearest neighbor algorithm.

Now, in a more generalized form of the algorithm instead of finding the single nearest neighbor, instead of finding the single example which is closest to the test example, what we do is that we find  $k$  training examples  $x_1, y_1, x_2, y_2, \dots, x_k, y_k$  which are closest to  $x_t$ . So,  $k$  may be 3, 4, 5 etc. We find  $k$  nearest examples, nearest training examples and predict as the output  $y_t$ . What should we predict; it depends on whether we are doing classification problem or a regression problem. For a classification problem, we look at  $y_1, y_2, \dots, y_k$  and we can predict that class which is the majority class among  $y_1, y_2, \dots, y_k$  we can predict the most frequent of the majority class.

So, we predict the majority class from  $y_1, y_2, \dots, y_k$ , but what if it is a regression problem. We have got different numerical outputs  $y_1, y_2, \dots, y_k$  and we will predict, average predict the average of among  $y_1, y_2, \dots, y_k$  i find out the average and predict it as the estimate of  $y_t$ . This is the basic  $k$  nearest neighbor algorithm which is very simple. Now, let us just look at the one nearest neighbor.

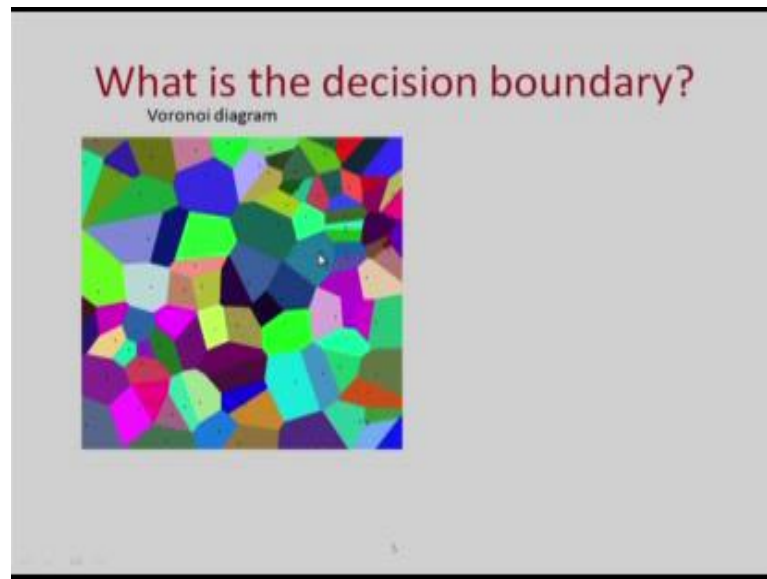
(Refer Slide Time: 10:19)



**Basic k-nearest neighbor classification**

- Training method:
  - Save the training examples
- At prediction time:
  - Find the  $k$  training examples  $(x_1, y_1), \dots, (x_k, y_k)$  that are closest to the test example  $x$
  - **Classification:** Predict the most frequent class among those  $y_i$ 's.
  - **Regression:** Predict the average of among the  $y_i$ 's.
- Improvements:
  - *Weighting* examples from the neighborhood
  - Measuring "closeness"
  - Finding "close" examples in a large training set *quickly*

(Refer Slide Time: 10:22)



For one nearest neighbor, let us see if we look at this slide. Here these points are the different instances. In the instance phase right now depending on where your test point will be if you test point will be here, the closest point to this is this point, right. So, in this phase we have different points. The classification in this phase for one nearest neighbor is coming from the may be most nearby point. Therefore, you can think of we can divide space into regions.

So, this blue region, this violet region, purple or whatever purple region corresponds to those places in phase for which this is the nearest neighbor. This region is the region which is the nearest neighbor. So, we have divided that the phase into different regions and this particular type division or this structure data structure called a Voronoi diagram. In this Voronoi diagram which can be constructed by you know you take any two neighboring points and to find a perpendicular bisector of the line jointing the two points that will give you separations or phase.

So, like this you can draw the Voronoi diagram and in this Voronoi diagram, in this region, this is the nearest point. For this region, all points in this region, this is the neighboring point. So, this captures the decision boundary of one nearest neighbor and as we have seen if you look at this slide, this is basic k nearest neighbor classification which

we have already discussed in the training time.

Training method is to save the training examples at prediction. You find the  $k$  training examples that are closest to the test example  $x$ . For classification you predict the most frequent class among those  $k$   $y_i$  values for regression, you predict the average among the  $y_i$ s. So, this is a very simple algorithm.

So, what we will discuss as certain points related to this algorithm and how to improve the algorithm. Some of the issue that will discuss is when we use  $k$  nearest neighbor where  $k$  is greater than 1, there is possibility of giving different ways to this  $k$  neighbors. So, we can weight different examples may be because of distance second issue is how to measure closeness. What type distance function one can use today, we will discuss that. We can use Euclidean distance function as one of the metric, but other distance functions are possible.

The third issue which is important which we will discuss later is how to find the closest points quickly at run time as we have seen such lazy algorithms during training time. We do not learn a model at prediction time. We get a point and find the nearest instances. If we do not use a good data structure or a good method to store the examples, we have to go through all the training examples, find the distance and find the smallest of them and if the training set is large, this may take considerable time. One issue of concern is to set up data structures, so that this can be done efficiently which we will not discuss today, but we will get some ideas in later classes.

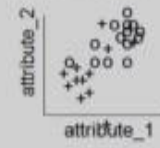
(Refer Slide Time: 14:32)

### k-Nearest Neighbor

$$Dist(c_1, c_2) = \sqrt{\sum_{i=1}^N (attr_i(c_1) - attr_i(c_2))^2}$$

$prediction_{test} = ?$

- Average of k points more reliable when:
  - noise in attributes
  - noise in class labels
  - classes partially overlap



So, first we look at a standard distance function called the Euclidean distance function. In Euclidean distance function, what we do?

(Refer Slide Time: 14:42)

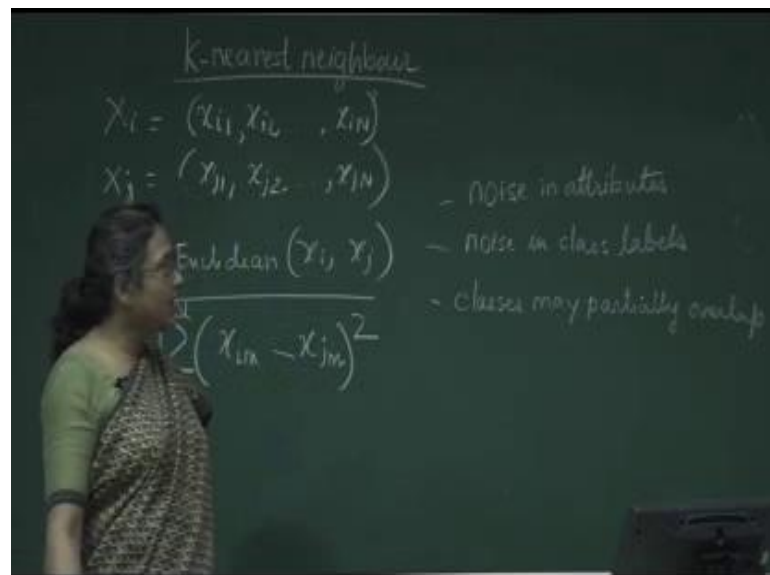
### k-nearest neighbour

$$x_i = (x_{i1}, x_{i2}, \dots, x_{in})$$
$$x_j = (x_{j1}, x_{j2}, \dots, x_{jn})$$

Euclidean  $(x_i, x_j)$

$$\sqrt{\sum (x_{in} - x_{jn})^2}$$

- noise in attributes
- noise in class labels
- classes may partially overlap



Suppose each instance  $x$  has  $n$  attributes  $x_{i1}, x_{i2}, \dots, x_{in}$  and suppose we have  $n$  attributes and between two instances  $x_i$  and  $x_j$ ,  $x_{j1}, x_{j2}, \dots, x_{jn}$ . So, given these two

instances, we want to find how close they are. I think all of you know about the Euclidean distance. In Euclidean distance, how do we measure the distance between these points. We say that distance Euclidean  $x_i, x_j$  is given by  $\sqrt{\sum_{k=1}^n (x_{ik} - x_{jk})^2}$ . So, this is the Euclidean distance. So, we can find the Euclidean distance from a test point to all the points that we have got from training and select that point which has the smallest Euclidean distance. When we take  $k$  as to be more than 1, we can take it as a regression problem.

We can take the average of the values, when do we go for averaging with large  $k$  or when do we go for  $k$  equal to 1. So, we will go for averaging under circumstances where there are noises in attributes. So, because of the noise, the instance that is closest to the test instance may not capture everything about the test instance in a better way than any other instance which is slightly further away, but still closer than there can be noise in class labels and thirdly classes may be overlapping.

So, under such circumstances, there is a case for using  $k$  larger than 1 and we will see that when we use a larger value  $k$ , we get a better smoother classifier. Now, the second thing we will worry about is when we look at the Euclidean distance, we are taking some of square root over some of squares of the distance for each attribute.

Let us not use  $k$  here because  $k$  is used for  $k$  nearest neighbor, but here we are talking about an index variable. So, let us use  $m$  now. All attributes are not equally important or the attributes may have different cases. So, a normal Euclidean distance treats all the attributes at the same scale, but there may be a case for giving different weights to the attribute. When you are giving equal weights to the attribute, you can give equal weights only under certain circumstances or certain assumptions.



(Refer Slide Time: 19:03)



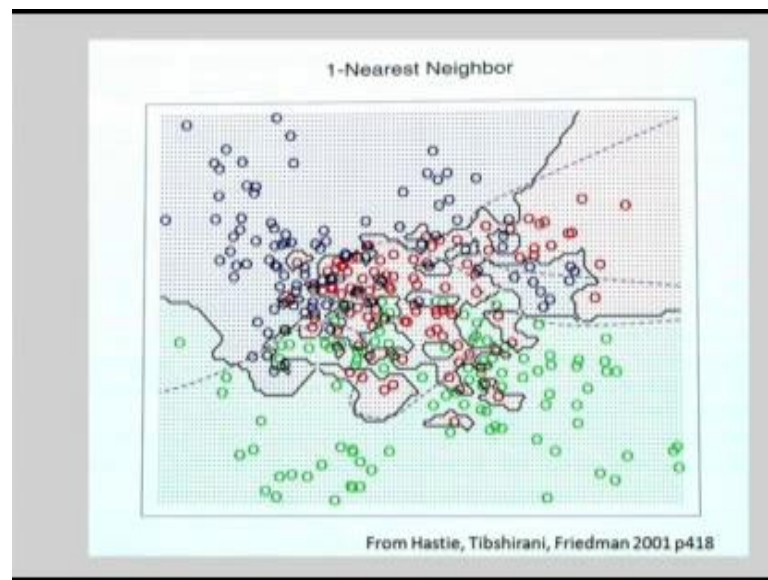
Should we give equal weights to all attributes? Now if you can do only if the scales of the attributes are similar. What do you mean by scale? Suppose height is an attribute and if you are measuring height in centimeter verse height in feet. The differences are not the same, right. So, you have the different attributes depending on what scale you use it. It will contribute more value or less value to the distance. So, if you are giving equal weights, you are assuming that the scale of the attributes and the differences and are similar. What do we mean by differences? If this  $x_{im}$  minus  $x_{jm}$ , how different they are for different pairs of training examples if that is similar, then only you can use equal weights. In fact, you make the assumption that you scale attributes, so that they have equal range one of the attribute values from 0 to 1000.

Another has values from 0 to 1, and then their range is not the same. So, the ranges should be similar and the variance should be similar under such condition only you can go for such a simple Euclidean distance function. Also you are assuming that you are taking  $x_{im}$  minus  $y_{im}$  whole square, assuming that classes are spherical.

So, the second assumption is that classes are spherical in shape. Under these assumptions, you can use basic Euclidean distance, but what the classes are not spherical. What if one attribute is more important than another attribute, what if some

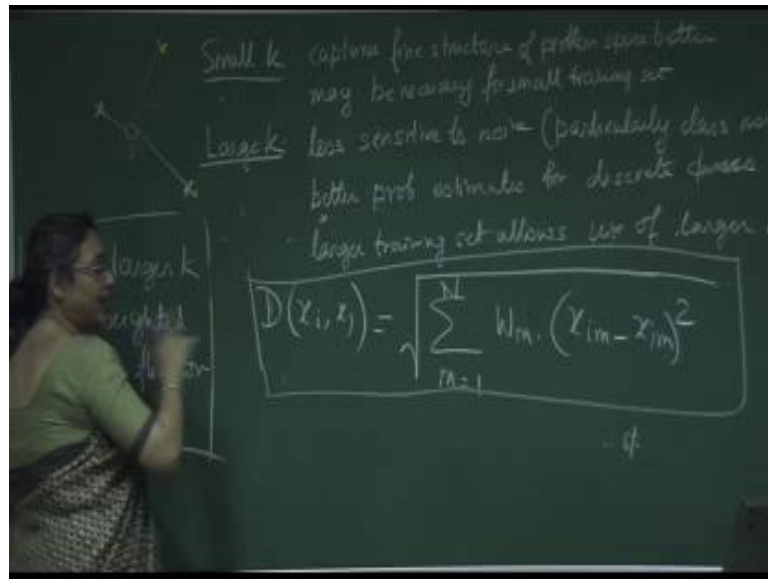
attributes have more noise than what you have in other attributes? Under those cases, this distance function will have some problems and the way you can overcome these problems, there are several things that one can do. One is you use larger  $k$  to moved out the difference and you use weighted Euclidean, you use weighted distance function. Now, what we will do? Now, when you say you use larger  $k$ , you have to have some idea how larger small  $k$  impacts. I think before we try to look at that, let us look at some example.

(Refer Slide Time: 23:11)



So, this picture taken from the book by Hastie Tibshirani Friedman shows an example where we have three classes; blue, red and green. They are denoted by the blue, red and green circles. So, if you use one nearest neighbor, you have decision boundaries between blue and green, green and red, red and blue etcetera. Based on that, these lines show you the decision boundary between the classes. Secondly, not at all smooth.

(Refer Slide Time: 24:02)



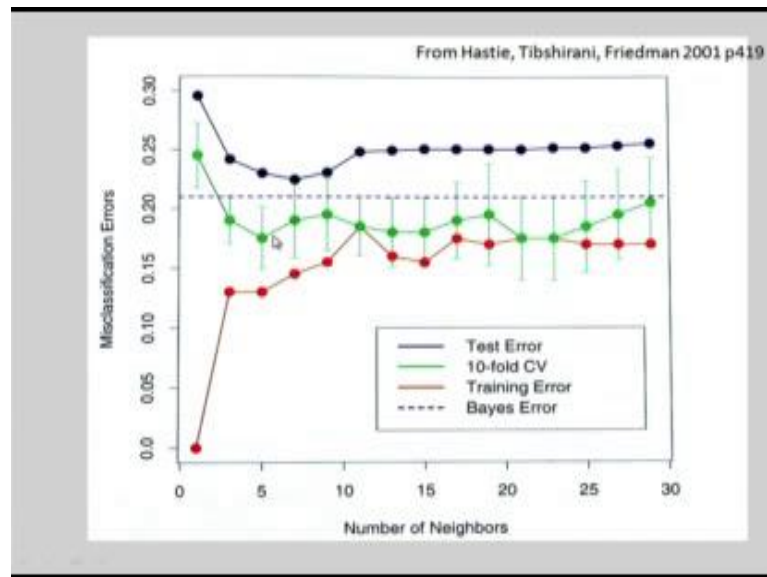
So, what we can see is that if we have small value of  $k$ , so small value of  $k$ , it captures fine structures of the problem phase better. You can see that these lines capture very fine structures of the problem phase. For example, here you see in this region, the class is blue where as in this region, the class is red, right. So, in this small region here, the classes blue where as these classes are red. So, the fine differences between the classes are captured when  $k$  is small. So, when small  $k$  captures, find structure of problems phase better if such fine structures exist and this may be necessary if the training set is small. On the other hand, you can use large value of  $k$  under the following circumstances.

Let us first look at an example. On the same data set using 15 nearest neighbor, what you notice here is that the classes are smoother. The neighboring classes in here, all of them are classified as red rather than blue for 50 nearest neighbor. So, use large  $k$  under these circumstances. When use large  $k$ , the classifier that you get is less sensitive to noise, particularly noise in the output class. So, particularly class noise, you get better probability estimates for discrete classes and you can use for discrete classes. If the classes are discrete, you get better probability estimates if use large values of  $k$ .

Thirdly, if you have larger size of training set, then you can use large values of  $k$ . So,

larger training set allows you to use larger  $k$ . If you have very small training set size, you cannot use large  $k$ . You have to use small  $k$ . So, this is the basic idea of getting different values of  $k$ . Now, let us look at the diagram on the slide.

(Refer Slide Time: 27:48)



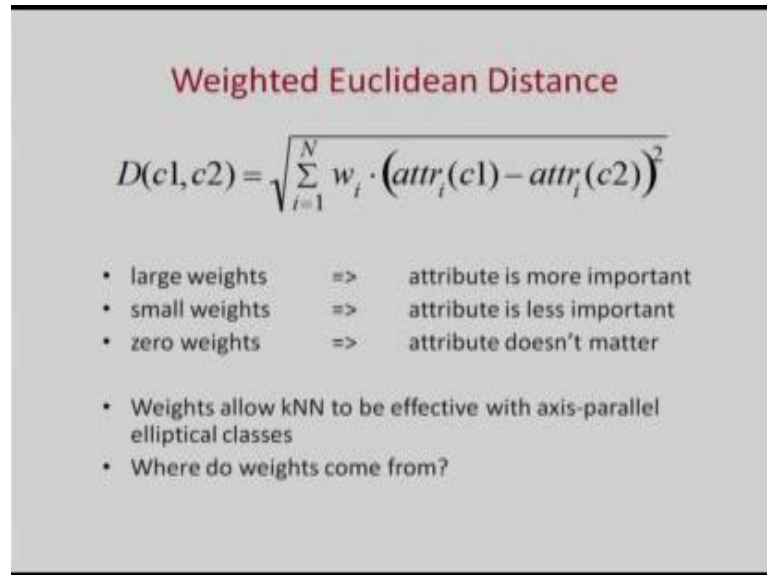
So, this picture shows you know this shows the number  $x$  axis is the number of neighbors. The blue line is test error, the green line is ten-fold cross validation error and the red line is training error and the dashed line is base error which is close to the true error. We see that in the training set as we are increasing the value of  $k$ , you know initially the error will increase and then, error will after that remain all most steady, but for the test error and for the cross validation error, we see that for  $k$  equal to 1, for one nearest neighbor, the error is high and then, the error falls off.

After sometime the error rises and then, keeps almost fixed. So, this is the region where the test error is smallest and this is region where the cross validation error is smallest. So, in this particular example, the error is smallest that around  $k$  equal to 5 or  $k$  equal to 7. So, refer small  $k$ , the error is high when large scale also, it increases and some middle value  $k$ , you have the best performance if you look at the test error.

Now, we come to the second question. The issue was using different value of  $k$ . The

second issue was to use weighted distance function. Now, we will see how we can use weighted distance function.

(Refer Slide Time: 29:35)



**Weighted Euclidean Distance**

$$D(c1, c2) = \sqrt{\sum_{i=1}^N w_i \cdot (attr_i(c1) - attr_i(c2))^2}$$

- large weights => attribute is more important
- small weights => attribute is less important
- zero weights => attribute doesn't matter
- Weights allow kNN to be effective with axis-parallel elliptical classes
- Where do weights come from?

In the case of weighted distance function, what we do is that we define the distance. You know let me write it in our notation distance between  $x_i$  and  $x_j$  depends on different attributes with different weights. We have  $n$  attributes and their weights  $w_1, w_2$  and  $w_n$ . So, we take sigma  $m$  equal to 1 to  $n$  and then, we use  $w_m$  into  $x_{i,m}$  minus  $x_{j,m}$  whole square. So, this is weighted Euclidean distance. We are different weights for different attributes. So, for those attributes which are more important for the particular classification, we can use larger weights and for less important attributes, we can use smaller weights.

These weights can also be decided based on you know if an attribute has larger range, we can use small weights. If it is small range, we can use larger weights or we can scale the attributes, so that they similar ranger we can sort of normalize the attributes. So, they have same mean and standard deviation. Based on that we can then fix the weights depending on the importance of the attributes and if an attribute does not matter, it is irrelevant the attribute has zero weight.

So, this brings us to an important question that if we have the instance phase defined in terms of a large number of attributes or features it possess, a problem in defining an appropriate similarity metric and it may also pose a problem for different other learning problems because some features may be more important than others and some features may be irrelevant and this specially impacts k nearest neighbor instance based learning algorithms greatly. So, it is important for us to remove extra features because if you have a very high dimensional phase two items which are similar may still differ in some unimportant attributes and the differences in distance between different pair items will be almost similar. So, it will be difficult to find good representative training examples for a given test example. So, feature reduction is very important.

(Refer Slide Time: 32:39)

**Distance-Weighted kNN**

- tradeoff between small and large k can be difficult
  - use large k, but more emphasis on nearer neighbors?

$$prediction_{test} = \frac{\sum_{i=1}^k w_i * class_i}{\sum_{i=1}^k w_i} \quad (or \quad \frac{\sum_{i=1}^k w_i * value_i}{\sum_{i=1}^k w_i})$$

$$w_i = \frac{1}{Dist(c_i, c_{test})}$$

We will talk about feature reduction in later class, may be in the next class, but we have to summarize in distance weighted k nearest variable, we have a trade-off between small and large k which can be difficult because large k means more emphasis on nearer neighbors. So, in the weighted k n n, the prediction is based on this weighted average, right. This weight can be based on that distance; the weight can be proportional to the inverse of the distance between the two examples.

So, for a particular example, suppose this is a test example, suppose this is a test example

and these are three nearest. Now, this closes. This is median distance; this is a little further, right. Now, we look at the class of all these three instances and take a weighted average, we gave larger weight to this smaller to this and smaller to this. Based on this, we get do the averaging. This is called distance weighted k nearest neighbor.

(Refer Slide Time: 33:54)

### Locally Weighted Averaging

- Let k = number of training points
- Let weight fall-off rapidly with distance

$$prediction_{test} = \frac{\sum_{i=1}^k w_i * class_i}{\sum_{i=1}^k w_i} \quad (or \quad \frac{\sum_{i=1}^k w_i * value_i}{\sum_{i=1}^k w_i})$$

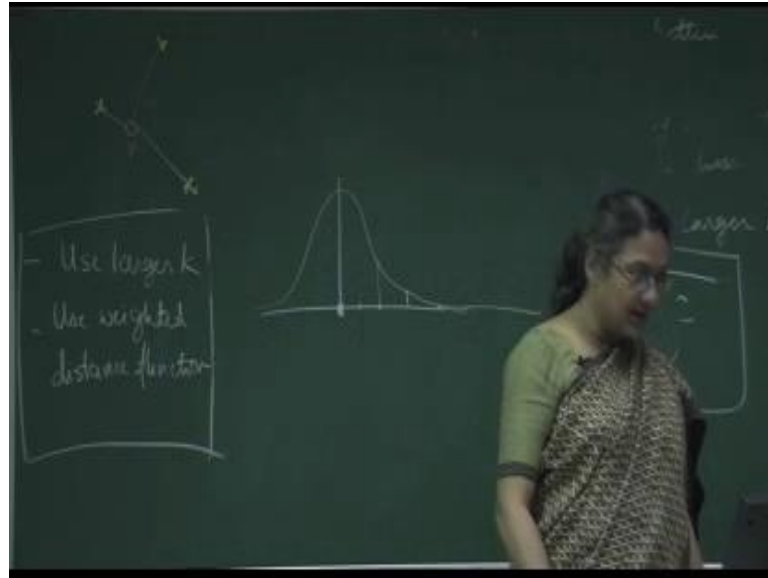
$$w_k = \frac{1}{e^{KernelWidth * Dist(c_x, c_{test})}}$$

- KernelWidth controls size of neighborhood that has large effect on value (analogous to k)

We can also instead of using this distance in you know the weight based on inverse of that distance, we can use other types of functions for this weighting. So, one of the ideas is locally weighted averaging. What locally weighted averaging does is that suppose k is the number of training points.

So, instead of you know we could also take instead of taking k as the 3 or 4 or 5, we could take very large value of k infect k could be entire training examples and then, we could give different weights to different training examples. So, if k is the total number of training points, we define weighting function, so that weight of nearby functions will reasonably will have some reasonable values, but for further training points also, some weight will be taken, but that weight function it will follow up rapidly weight distance. For example, you could think of the weighting function based on Gaussian function, right.

(Refer Slide Time: 35:15)



So, you know this is you know very rough. Secondly, Gaussian functions suppose this is your test point. So, any example which is here you know it will get this much of weight and example here will get this much of weight, example here will get this much of weight and example further away will get almost zero weight, but it will get some value of weight. So, we could use different weighting functions.

For example, here we are saying  $w_k$  is  $1/e$  to the power kernel with into distance between  $c_k$  nct. So, you using an exponential function and the weight are falling of rapidly after you know based on this kernel width. So, kernel width controls how much of the neighborhood you want to gave reasonable weighting. So, if kernel width is large, you are considering more in more area in the neighborhood. If it is small, you are using less area in the neighborhood, based on that you can do locally weighted averaging.

With this, we come close for this todays lecture. In the next lecture, we will discuss issues about feature dimensionality reduction and other issues.

Thank you very much.