**“These notes would help you all understand the topics of Module 9—- Gyanendra”**

**Naive Bayes classifier**

It is an classification algorithm and based on Bayes Theorem.

One important assumptions is all the feature or input columns must be independent from each other.

When you have more than two classes to predict or classify the data into then you can use NBC.

———-

Bayes theorem provides a way of calculating the posterior probability, *P*(*c|x*), from *P*(*c*), *P*(*x*), and *P*(*x|c*). Naive Bayes classifier assume that the effect of the value of a predictor (*x*) on a given class (*c*) is independent of the values of other predictors. This assumption is called class conditional independence.

Explain the formula of NBC

*P*(*c|x*) is the posterior probability of *class* (*target*) given *predictor* (*attribute*).

* *P*(*c*) is the prior probability of *class*.
* *P*(*x|c*) is the likelihood which is the probability of *predictor* given *class*.
* *P*(*x*) is the prior probability of *predictor*

|  |
| --- |
| The posterior probability can be calculated by first, constructing a frequency table for each attribute against the target. Then, transforming the frequency tables to likelihood tables and finally use the Naive Bayesian equation to calculate the posterior probability for each class. The class with the highest posterior probability is the outcome of prediction. |

Outlook, Humidity and Wind are independent variables and Play is a dependent variable.

Frequency table: identify how many unique values are present in the column.

**————————————————————————**

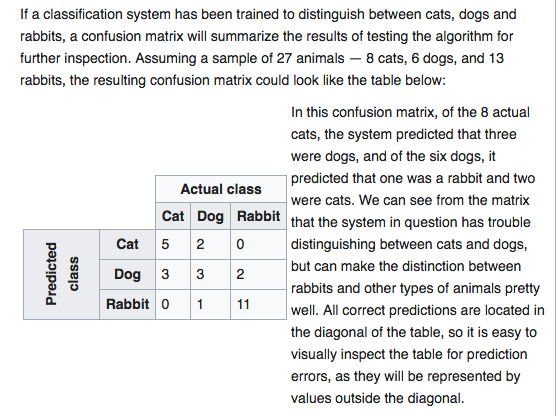
**Confusion Matrix:**

A confusion matrix is a summary of prediction results on a classification problem.

The number of correct and incorrect predictions are summarised with count values and broken down by each class. This is the key to the confusion matrix.

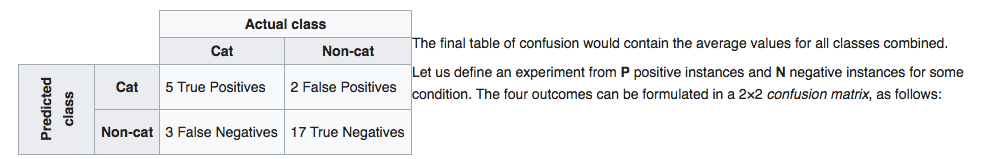
The confusion matrix shows the ways in which your classification model is confused when it makes predictions.

It gives us insight not only into the errors being made by a classifier but more importantly the types of errors that are being made.

From Wiki:

**Table of confusion:**

In predictive analytics, a table of confusion (sometimes also called a confusion matrix), is a table with two rows and two columns that reports the number of false positives, false negatives, true positives, and true negatives. This allows more detailed analysis than mere proportion of correct classifications (accuracy). Accuracy is not a reliable metric for the real performance of a classifier, because it will yield misleading results if the data set is unbalanced (that is, when the numbers of observations in different classes vary greatly). For example, if there were 95 cats and only 5 dogs in the data, a particular classifier might classify all the observations as cats. The overall accuracy would be 95%, but in more detail the classifier would have a 100% recognition rate (sensitivity) for the cat class but a 0% recognition rate for the dog class. F1 score is even more unreliable in such cases, and here would yield over 97.4%, whereas Informedness removes such bias and yields 0 as the probability of an informed decision for any form of guessing (here alway guessing cat).

Assuming the confusion matrix above, its corresponding table of confusion, for the cat class, would be:

**Definition of the Terms:**

• Positive (P) : Observation is positive (for example: is an apple).

• Negative (N) : Observation is not positive (for example: is not an apple).

• True Positive (TP) : Observation is positive, and is predicted to be positive.

• False Negative (FN) : Observation is positive, but is predicted negative.

• True Negative (TN) : Observation is negative, and is predicted to be negative.

* False Positive (FP) : Observation is negative, but is predicted positive.

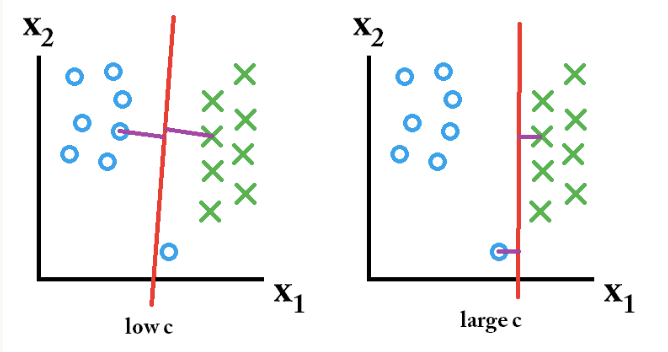
**Impact of ‘C' in SVM**

https://stats.stackexchange.com/questions/31066/what-is-the-influence-of-c-in-svms-with-linear-kernel

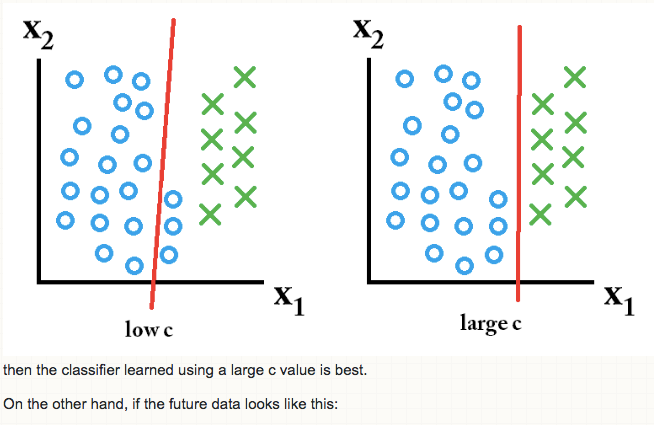
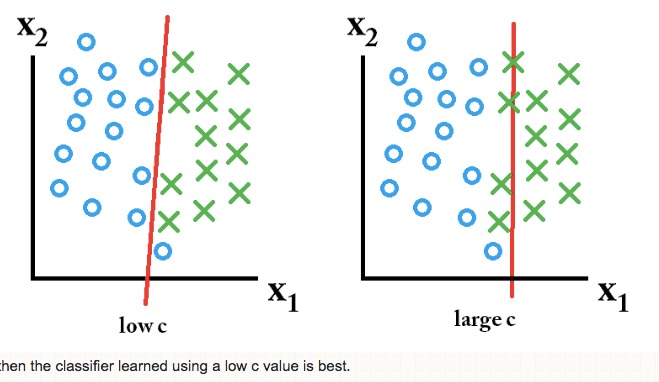
Penalty parameter C of the error term also called ‘**cost of miscalculation**’

The C parameter tells the SVM optimisation how much you want to avoid misclassifying each training example. For large values of C, the optimization will choose a smaller-margin hyperplane if that hyperplane does a better job of getting all the training points classified correctly. Conversely, a very small value of C will cause the optimizer to look for a larger-margin separating hyperplane, even if that hyperplane misclassifies more points. For very tiny values of C, you should get misclassified examples, often even if your training data is linearly separable.

**In a SVM you are searching for two things: a hyperplane with the largest minimum margin, and a hyperplane that correctly separates as many instances as possible. The problem is that you will not always be able to get both things**. The c parameter determines how great your desire is for the latter. I have drawn a small example below to illustrate this. To the left you have a low c which gives you a pretty large minimum margin (purple). However, this requires that we neglect the blue circle outlier that we have failed to classify correct. On the right you have a high c. Now you will not neglect the outlier and thus end up with a much smaller margin.



So which of these classifiers are the best? That depends on what the future data you will predict looks like, and most often you don't know that of course. If the future data looks like this:



**Kernel:**

**What are kernels?**

A kernel is a **similarity** function. It is a function that you, as the domain expert, provide to a machine learning algorithm. It takes two inputs and spits out how similar they are.

Suppose your task is to learn to classify images. You have (image, label) pairs as training data. Consider the typical machine learning pipeline: you take your images, you compute features, you string the features for each image into a vector, and you feed these "feature vectors" and labels into a learning algorithm.

Data --> Features --> Learning algorithm

Kernels offer an alternative. Instead of defining a slew of features, you define a single kernel function to compute similarity between images. You provide this kernel, together with the images and labels to the learning algorithm, and out comes a classifier.

Of course, the standard SVM/ logistic regression/ perceptron formulation doesn't work with kernels : it works with feature vectors. How on earth do we use kernels then? Two beautiful mathematical facts come to our rescue:

Under some conditions, every kernel function can be expressed as a dot product in a (possibly infinite dimensional) feature space ( Mercer's theorem ).

Many machine learning algorithms can be expressed entirely in terms of dot products.

These two facts mean that I can take my favorite machine learning algorithm, express it in terms of dot products, and then since my kernel is also a dot product in some space, replace the dot product by my favorite kernel. Voila!

**Why kernels?**

Why kernels, as opposed to feature vectors? One big reason is that in many cases, computing the kernel is easy, but computing the feature vector corresponding to the kernel is really really hard. The feature vector for even simple kernels can blow up in size, and for kernels like the RBF kernel ( k(x,y) = exp( -||x-y||^2), see Radial basis function kernel) the corresponding feature vector is infinite dimensional. Yet, computing the kernel is almost trivial.

Many machine learning algorithms can be written to only use dot products, and then we can replace the dot products with kernels. By doing so, we don't have to use the feature vector at all. This means that we can work with highly complex, efficient-to-compute, and yet high performing kernels without ever having to write down the huge and potentially infinite dimensional feature vector. Thus if not for the ability to use kernel functions directly, we would be stuck with relatively low dimensional, low-performance feature vectors. This "trick" is called the kernel trick ( Kernel trick ).

**Endnote:**

I want to clear up two confusions which seem prevalant on this page:

A function that transforms one feature vector into a higher dimensional feature vector is not a kernel function. Thus f(x) = [x, x^2 ] is not a kernel. It is simply a new feature vector. You do not need kernels to do this. You need kernels if you want to do this, or more complicated feature transformations without blowing up dimensionality.

A kernel is not restricted to SVMs. Any learning algorithm that only works with dot products can be written down using kernels. The idea of SVMs is beautiful, the kernel trick is beautiful, and convex optimization is beautiful, and they stand quite independent.

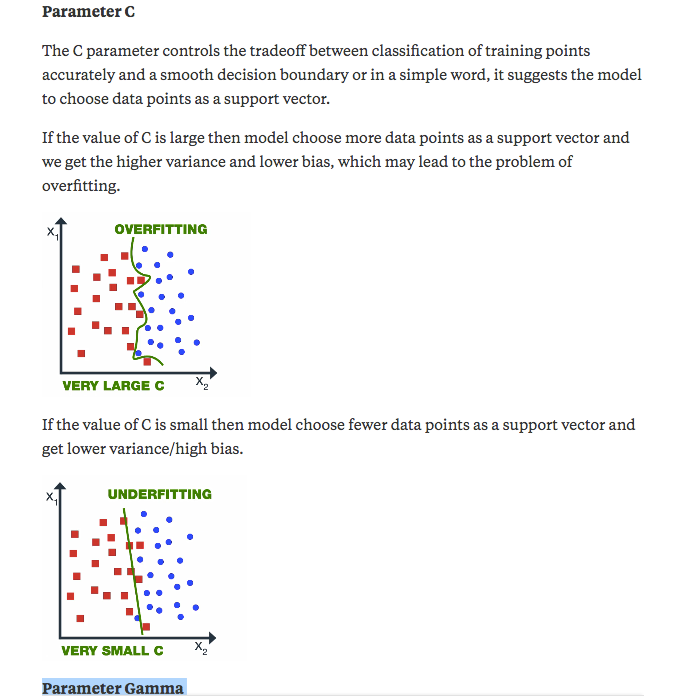
**Gamma**

**Parameter Gamma:**

K (xi, xj) = exp (-γ||xi - xj||2)

This is the equation of RBF kernel. Here γ is a positive constant and known as Gamma. Gamma defines how far the influence of single training example reaches.

If the value of Gamma is high, then our decision boundary will depend on points close to the decision boundary and nearer points carry more weights than far away points due to which our decision boundary becomes more wiggly.

If the value of Gamma is low, then far away points carry more weights than nearer points and thus our decision boundary becomes more like a straight line.

**Conclusion: Refer below screenshot as well.**

The value of gamma and C should not be very high because it leads to the overfitting or it shouldn’t be very small (underfitting). Thus we need to choose the optimal value of C and Gamma in order to get a good fit.