# **Introduction to SVM**

Support Vector Machines, or SVMs, are a class of extremely popular classification models. Besides their ability to solve complex machine learning problems, they have numerous other advantages over other classification problems, such as the ability to deal with computationally heavy data sets, classifying nonlinearly separable data, etc. We will discuss some of these problems in detail in the upcoming lectures.

It is important to remember that SVMs belong to the class of **linear machine learning models** (logistic regression is also a linear model). This concept will be very important in the upcoming lectures.

A linear model uses a linear function (i.e. of the form y = ax +b) to model the relationship between the input x and output y. For example, in logistic regression, the log(odds) = ln(P/1-P) of an outcome (say, defaulting on a credit card) is linearly related to the attributes x1, x2, etc.

log(odds of default)=ln(p/1-p)=β0+β1X1+β2X2+...βnXn

SVM is a supervised machine learning algorithm which can be used for classification or regression problems. It uses a technique called the kernel trick to transform your data and then based on these transformations it finds an optimal boundary between the possible outputs. Simply put, it does some extremely complex data transformations, then figures out how to separate your data based on the labels or outputs you've defined.

|  |  |
| --- | --- |
|  | SVMs need attributes in the **numeric form**. In our (somewhat simplified) example of email classification, the features are the ‘frequency of words’ in an email. Word frequency is a very common type of feature in text classification, and reasonably so. For example, words such as ‘Hurry’, ‘FREE’ and ‘Discount’ are likely to be more frequent in spam emails than ‘meeting’, ‘PPT’, ‘weekly report’, and others that are characteristic of office emails. |

# Concept of a Hyperplane in 2D

Before you move on to **support vector machines**, you need to understand the **concept of hyperplanes**. Essentially, it is a boundary which 'separates' the data set into its classes (in this case, separates spam emails from the ham ones). It could be lines, 2D planes, or even n-dimensional planes that are beyond our imagination. A**line**that is used to classify one class from another is also called a **hyperplane**. In fact, it is the model you're trying to build as shown in the figure-1 below

|  |  |  |
| --- | --- | --- |
|  | The equation of any such **line** is ax+by+c = 0, where a,b and c are the three co-efficient. Any point on this line will satisfy the above equation. | |
|  | | We could generalize it as W0+W1x1+W2x2=0, where x1 and x2 are the features — such as 'word\_freq\_technology' and 'word\_freq\_money' — and W1 and W2 are the coefficients.  A **positive value** (blue points in the plot above) would mean that the set of values of the features is in one class; however, a **negative value** (red points in the plot above) would imply it belongs to the other class. A value of zero would imply that the point lies on the line (hyperplane) because any point on the line will satisfy the equation: W0+W1x1+W2x2=0. |

**Hyperplane**

Let's say that a straight line, L, as shown in the plot below, is given by x2=w1x1+w0, and divides the points belonging to two classes, C1  and C2, in a 2D space. If the points (a,b) and (p,q) belong to C1 and C2 respectively, then:

|  |  |
| --- | --- |
| https://images.upgrad.com/cca654c4-0834-44ae-be79-3b8a888127d7-12.PNG | Top of Form  ​(b−w1a−w0)∗(q−w1p−w0)>0​  ​(b−w1a−w0)∗(q−w1p−w0)<0​  **Feedback :** *Since the two points belong to different classes, the terms*(b−w1a−w0)*​ ​and*(q−w1p−w0)*will have different signs for instance, if you observe the red points which lies below the line(hyperplane), give the negative value of expression*x2−w1x1−w0*. On the other hand, blue points which lies above the line(hyperplane), give the positive value of same expression*x2−w1x1−w0*. Thus, if you multiply both the expression's value, you will get a negative value.*  **Correct**  **​**(a−w1b−w0)∗(p−w1q−w0)>0**​**  **Feedback :** *Plug in the two given points in the equation of L and analyse the signs of the two terms. Will the signs be the same or different?* |

Bottom of Form

|  |  |
| --- | --- |
| In the vector space shown in the plot below, if there are various hyperplanes plotted, which hyperplane do you think is the right classifier?  **https://images.upgrad.com/b6dae2ae-97ec-4293-94ea-bd185071b0dc-11.PNG** | **Red**  **Feedback :**  *Yes. Only the hyperplane in red correctly classifies both the classes.* |

Top of Form

Bottom of Form

**Practice Questions**

Consider a data set with two independent variables, say **X1** and **X2,** and one dependent binary variable,**Y**, with two classes, +1 or -1. The data set is plotted in the figure below.

|  |  |
| --- | --- |
|  |  |

Based on the information above, can the data set be separated by a **hyperplane,**i.e. a straight line?

Top of Form

**Yes. It is possible to divide the classes by a straight line**

**Feedback :** *As shown in the plot above, points 1, 2, 3 and 4 can be separated from 5, 6 and 7 by the hyperplane or line.*

Bottom of Form

If the equation of the **optimal hyperplane** is represented by X2=W1.X1+W0, then (Hint: W1 is the slope of the line and W0 is the intercept, i.e. the point where it cuts the X2 axis) imagine the hyperplane and say whether the slope and intercept will be positive or negative.

Top of Form

​W1​< 0, ​W0​ = 0

​W1 > 0, ​W0​ < 0

**Feedback :** *‘W1’ is the slope of the line and ‘W0’ is the intercept (the point where it cuts the X2 axis). The slope will be positive since the hyperplane or line is tilted towards the right. The constant 'W0' is where it cuts the X2 axis, which is a point below X2 = 0.*

**Correct**

Bottom of Form

The Y labels for classes 1 and 2 are 1 and -1 respectively. If the optimal separating hyperplane is represented by X2=W1∗X1+W0, then the classification rule to classify a general point (p, q) is given by:

Top of Form

​(W1.p−q+W0).Y>=0​

**Feedback :** *The equation of the hyperplane is given as ​*W1X1−X2+W0=0*​.If any new point lies below the line, the value of Y will be -1, and the value of the expression ​*W1X1−X2+W0*​ is less than zero; this means the product will be positive. Similarly, if the point lies above the line, the value of Y will be +1, and the expression W1X1 - X2 + W0 is greater than zero. Thus, the product is positive. Also, in the case of the point on the line, the value of the expression ​*W1X1−X2+W0*​ is equal to zero. Thus, the expression ​*(W1.p−q+W0).Y*​ gives a positive value or zero.*

**Correct**

Bottom of Form

# Maximal Margin Classifier

The Maximal Margin Classifier ensures a margin of safety that the normal classifier (hyperplane) doesn’t. But what is the advantage of this?

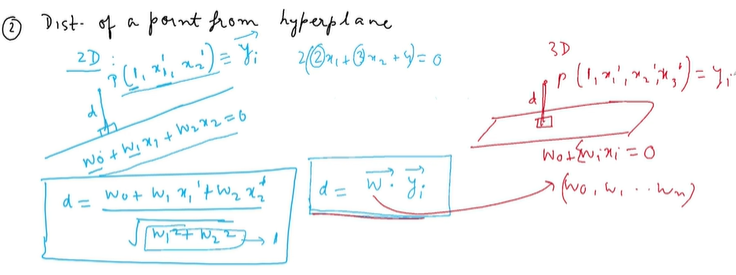
|  |  |
| --- | --- |
|  | *The Maximal Margin Classifier divides the data set in such a way that it is equidistant from both the classes. Thus, it maintains an equal distance from both classes, making the model less biased to the training data. Also, training errors are reduced.* |

The best line is the one that maintains the **largest possible equal distance**from the nearest points of both the **classes**. It is also referred to as a **maximal margin classifier**. You can think of the margin as a 'band' that the hyperplane has on both its sides. There can be several lines (hyperplanes) possible in the same data set, as shown in the figure below, with different values of margins. Among these, the line with the **maximum margin**would be considered the best fit line for the given data

|  |  |
| --- | --- |
|  | It's obvious, from the given margin values, that the **third hyperplane** is the best of the lot. Why? Because it maintains the maximum margin value.  But finding the maximal margin hyperplane needs a mathematical formulation. Let's now understand the formulation for an optimal **maximal margin classifier.** |

**Note**: In the coming lectures, you will need to recall the **concept of dot product** of two vectors. Say there are two vectors **A = [2, 3, 5]** and **B = [1, 0, 4]**, then the dot product of A and B is the element-wise multiplication of A and B, i.e. **transpose(A). B = A'B** = 2.1 + 3.0 + 5.4 =22. Note that the dot product of two vectors returns a **scalar number**, in this case, 22.

|  |  |
| --- | --- |
|  | To simplify the notations, first we added a col ‘one’ or ‘1’ with all 1’s. We also add a label column L/l. Now the equation can be represented as vector dot product of coefficient ‘w’ with each data point xi as y­i vector. |

****

**While defining the maximal margin classifier formulation, we are trying to:**

Find the weights corresponding to the hyperplane having the maximum possible weights. A hyperplane basically defined by its weights. We are trying to find the hyperplane, or the weights, such that the margin is maximum.

|  |  |
| --- | --- |
|  | The condition is multiplied by the dot (.) product of vector w and vector corresponds to the ith data point yi must be > 0.  Ii \* (vecW, vecYi) > 0 , i.e. w correctly classifies the ith data point. If it correctly classifies the ith data point then this condition is true. An important note here the coefficients (w) forms a unit vector. |

Scaling a hyperplane by some means geometrically it’s the same object here it is a straight line. Even though algebraically these are different as the coefficients are different.

* 2 \* (x + 2x +3 = 0)
* 2x + 4y +6 =0

Both the above represents the same hyperplane geometrically. Hence, we can say that the **coefficients can be written in normalized form and it’s a normalized vector**.

The mathematical formulation requires **two major constraints** that needs to be taken into account while **maximising the margin**. They are

* The standardisation of coefficients such that the summation of the square of the coefficients of all the attributes is equal to 1. For example, if you have 20 attributes, then the summation of square of the coefficients should be ∑20i=1(W2i)=1
* Along with the first constraint, the maximal margin hyperplane should also follow the constraint given below:

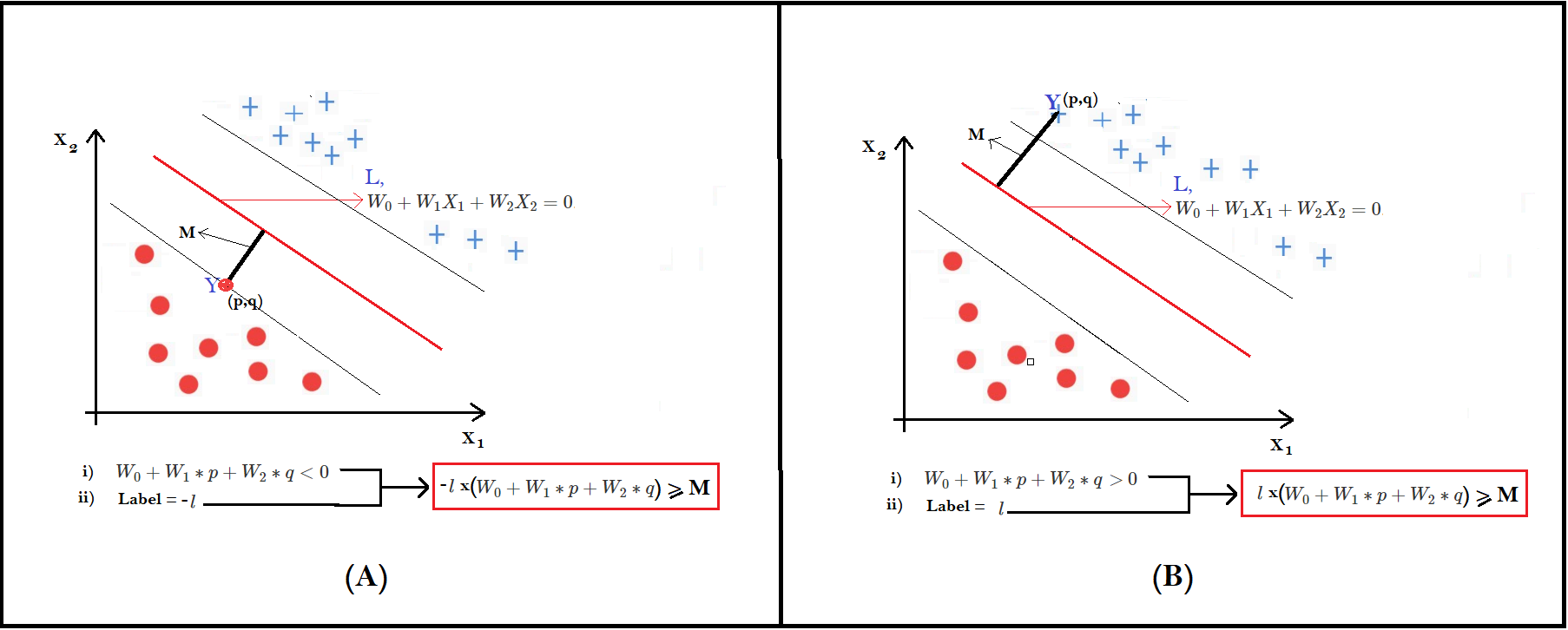
Vector dot (.) product **(li\*(Wi.Yi))⩾M** where, **li** = label (1, -1) , **Wi** = coefficient of attributes, **Yi**  = data points of all the attributes in each row

Let's understand these constraints through an example, since you already know how to write an equation for a hyperplane in 2D. Now say that the hyperplane is represented as W0+W1X1+W2X2=0. So what you need to do is apply both the constraints in this equation such that you get a maximal margin hyperplane equation that maintains an equal distance from both the labels, as shown in figure 2 below.

The**first constraint** should be valid if**W20**+ **W21 + W22 = 1**

For the **second constraint**, the hyperplane should be a safe distance of more than or equal to the margin 'M' from both the labels. If you look at **figure 2(A)**, all the red dots are below the hyperplane that is labelled (*-L*). Here, the multiplication of (*-L*), with the dot product of Y(p,q) and the equation W0+W1X1+W2X2, will give you a positive value that is greater than or equal to **'M'**.

Similarly, if you look at **figure 2(B)**, all the blue dots are above the hyperplane that is labelled (*+L*). Here, the multiplication of (*+L*), with the dot product of Y(p,q) and the equation W0+W1X1+W2X2, will also give you a positive value that is greater than or equal to 'M'.



**Among the multiple possible hyperplanes, why is one being called 'better' than the others?**

Top of Form

The better one has the least number of misclassifications.

The better one lies closer to the points than any one class and thus, classifies the other class accurately.

The better one is a 'safe' distance away from both the classes and thus, will minimize the chances of incorrect identification.

**Feedback :** *The Maximal Margin Classifier is better than the others because it maintains an equal distance from both the classes; this performs better on the test set.*

**State whether true or false. The maximal margin hyperplane is equidistant from both the classes, where 'distance' implies the distance of the closest point to the hyperplane.**

Top of Form

**Feedback :** *Yes. The margin should be selected in such a way that it has the maximum distance from both the classes. For example, if you want to categories spam and ham by a plane, then the plane should be drawn in such a way that it is equally set apart from both the classes, i.e. 'spam' and 'ham'.*

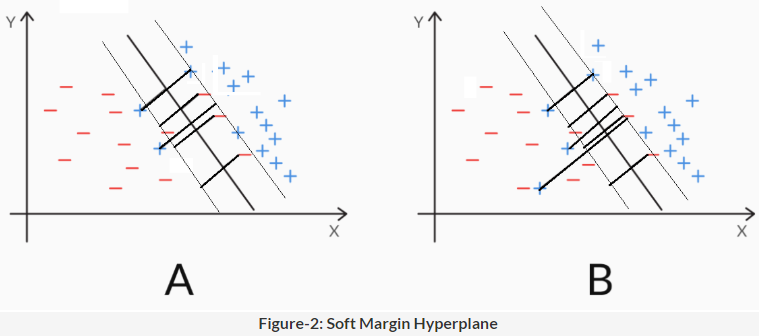
The maximal margin line (hyperplane), although it separates the two classes perfectly, is very sensitive to the training data. This means that the **Maximal Margin Classifier** will perform perfectly on the training data set. But on the unseen data, it may perform poorly. Also, there are cases where the classes cannot be perfectly separated. For example, if you look at figure 3 below, do you think the maximal margin can classify the data points perfectly?

|  |  |
| --- | --- |
|  | It seems that the maximal margin line (hyperplane) is not even possible in this case. In this case, if you want to create a linear hyperplane, you will inevitably need to misclassify a few data points. In other words, some **points will need to fall on the wrong side** of the hyperplane. |

# The Soft Margin Classifier – Support Vector Classifier (SVC)

**Support vectors**are the points that **lie close to the hyperplane**. In fact, they are **the only points that are used in constructing the hyperplane**. The SVCs(Support vector Classifiers) are relatively immune to outliers. Because SVCs are formulated from the support vector points. It implies that SVC (i.e. hyperplane) will not be changed if we do not change the support vectors.

The hyperplane that allows certain points to be **deliberately** misclassified is also called the **Support Vector Classifier**. The support vector classifier works well when the data is **partially intermingled** (i.e. most of the data can be classified correctly with some misclassifications). Similar to the Maximal Margin Classifier, the Support Vector Classifier also maximises the margin; but it will also allow some points to be misclassified, as shown in **figure 2(A & B)** below.



The Maximal Margin Classifier has certain limitations and drawbacks. Due to this, we are now moving towards the Support Vector Classifier.

1. It can be extremely sensitive to individual observations. In other words, the model can drastically change if a few points are changed.
2. It cannot classify data that is linearly inseparable, i.e. if the classes cannot be divided by a straight line. The Maximum Margin Classifier does not misclassify any point. So it cannot be built on linearly inseparable data i.e. intermingled data.

To differentiate between soft margin and hard margin is a soft margin is used in constructing the Soft Margin Classifier (Support Vector Classifier). SVC allows some points to be misclassified, whereas the hard margin ensures no point ae misclassified.

|  |  |
| --- | --- |
|  | With introducing epsilon (1− εi) we are allowing the data points to be closure to hyperplane. Even εi > 1, then 1- εi becomes negative. When these terms become negative we are allowing the data points to fall on the other side. So   * The one which are correctly classified but are uncomfortably close to the hyperplane would have their epsilon values 0 < ε< 1 * The one which are clearly correctly classified would have epsilon value εi = 0 * The one which are incorrectly classified for those data points the εi would be ε> 0 |

**Each data point has a slack value** associated to it, according to where the point is located.

The value of slack lies **between 0 and +infinity. Lower values of slack are better** than higher values (slack = 0 implies a correct classification, but slack > 1 implies an incorrect classification, whereas slack within 0 and 1 classifies correctly but violates the margin).

Once you understand the notion of the slack variable, you can easily compare any two Support Vector Classifiers. You can measure the summation of all the epsilons(**ϵ**) of both the hyperplanes and choose the best one that gives you the least sum of epsilons(**ϵ**). The summation of all the epsilons of each data point is denoted by cost or 'C', i.e. ∑ εi ≤C.

When **C is large**, the slack variables can be large, i.e. you allow a larger number of data points to be misclassified or to violate the margin. So you get a hyperplane where the margin is wide and misclassifications are allowed. In this case, the model is**flexible, more generalizable, and less likely to overfit**. In other words, it has a **high bias.**

On the other hand, when **C is small**, you force the individual slack variables to be small, i.e. you do not allow many data points to fall on the wrong side of the margin or the hyperplane. So the margin is narrow and there are few misclassifications. In this case, the model is **less flexible, less generalisable, and more likely to overfit.** In other words, it has a **high variance.**

# Comprehension-1: Notion of Slack Variables

As you have already learnt, the mathematical formulation for the Maximal Margin Classifier can be expressed as

li\*(W.Yi)>=M, where

* li represents the label of the ith observation (such as spam(+1) and ham(-1));
* **W** represents the vector of the coefficients (or weights) of each attribute (for example, if you have 3 attributes, W = [w0, w1, w2, w3]).
* Yi represents the vector of the attribute values for the ith row, e.g. Y = [y1, y2, y3] for 3 attributes.

Thus, the dot product W.Yi is simply the value of the expression obtained by putting the ith data point in the hyperplane equation, i.e. W.Yi=w0+w1y1+w2y2+w3y3.

Thus, W.Yi is lesser than, equal to or greater than 0, depending on the location of the **ith data point** with respect to the hyperplane. Also, note that the value of W.Yi gives you the distance of the**ith data point from the hyperplane**.

**M** represents the margin, i.e. the distance of the closest data point from the hyperplane.

If you impose the condition (li\*(W.Yi)>=M) on the model, then you are implying that you want each point to be at least **a distance M away from the hyperplane**. But unfortunately, few real datasets will not be so easily, perfectly separable. Thus, to relax the constraint, you include a ‘**slack variable**’ epsilon εi for each data point **i**. Thus, you modify the formulation to li\*(W.Yi)>=M(1− εi), Where the **slack variable epsilon (**εi**)**takes a value between **0 to infinity**. Depending on the value of epsilon εi, the ith data point can now take any position - it can fall on the**correct side of the margin** (and a safe distance away), or **inside the margin** (but still correctly classified), or **even stray on the wrong side of the hyperplane** itself.

**If the ith data point has epsilon(**εi**) = 0, then:** The point falls on the correct side of the margin, i.e. correctly classified and at a positive distance away from the margin

**Feedback:** *Yes, In this case, the value of ​*li\*(Y.Wi)>=M*​, which is exactly same as the maximal margin classifier. In this case, the data point falls on the correct side of the hyperplane*

**Notion of Slack Variables**

As you learnt, the summation of all the epsilon is equal to cost “C” So, what happens when C is large, say 100, and what happens when C is small, say 5?

Compare the two cases on the basis of:

* In which case would a higher number of points be misclassified?
* In which case is the model more likely to overfit?

**Suggested Answer**

If C is large, then the slack variables epsilon ​εi​ can take higher values. And you know that when slack(​​εi ​) > 1, the point is misclassified, between 0 and 1, it falls inside the margin, and when slack(​​εi) = 0, it is correctly classified. Thus, for higher C, more points will be allowed to get misclassified or fall inside the margin (compared to a lower C).

On the other hand, a lower C implies that each slack(​εi ​) will have to take a lower value, and thus not be allowed to stray on the other side of the margin or the hyperplane. This is a more strict condition. In other words, a lower C does not give the model freedom to misclassify even a few points, and thus the model tries to overfit the data.

|  |  |
| --- | --- |
|  | **Which of the hyperplanes shown below corresponds to a higher C?**  Hyperplane 1.  *If C is large, the slack variables (epsilons( ϵ)) can be large, i.e. you allow a larger number of data points to be misclassified, that indicates wider margin. So you get a hyperplane where the margin is wide and misclassifications are allowed. And hence, it will work well on unseen data.*  *Even though hyperplane 1 allows a few misclassifications, it maximises the margin so the test points are classified correctly and the model is not biased.* |

The professor pointed out the importance of reading documentation before using libraries. In fact, it is quite important to do that before implementing SVM in python, since the meaning of the hyperparameter C in sklearn is quite different. Please note that the parameter C that you used in the SVM formulation (in the theory lectures) and the C in the SVC() function are the**inverse of each other.**

Previously, we discussed how C is the sum of all the slack variables, and thus, a **large value of C**implies that you are willing to **accommodate a higher number of misclassifications** (and thus, the model is **more generalisable**since it does not have to worry about classifying all the points correctly). According to this definition, a **large C ensures that the model will not overfit.**In other words, a **high value of the parameter C**, as discussed earlier, **regularizes the SVM model.**

On the other hand, in the SVC() implementation of python that you will use, the **hyperparameter** C is analogous to the **penalty imposed for misclassification**, i.e. a higher C will force the model to classify most (training) data points correctly (and thus, overfit). In sklearn, **high value of C** implies a **high cost of making errors**or misclassifications.

# **Introduction to Kernels**

Kernels are one of the most interesting inventions in machine learning, partly because they were born through the creative imagination of mathematicians, and partly because of their utility in **dealing with non-linear datasets**. This session is meant to give you an intuitive understanding of kernels.

In this session, you will study

* The problem: Linear and nonlinear feature spaces
* Why we need kernels: To reduce computational cost
* How a kernel works: The kernel trick
* SVM Lab - Letter recognition

|  |  |
| --- | --- |
|  | We can transform nonlinear boundaries to linear boundaries by applying certain functions to the original attributes. The original space (X, Y) is called the original **attribute space,** and the transformed space (X’, Y’) is called the **feature space.**  To do this transformation here we calculated the squares of each point. |

Similarly, we can classify the emails into ‘spam’ or ‘ham’ on this basis of two attributes 'word\_freq\_office' (Y) and 'word\_freq\_lottery' (X). The following plot shows the data set (figure 1), which is clearly nonlinear.

|  |  |
| --- | --- |
|  | To convert this data set into a linearly separable one, a simple transformation into a new feature space (X’, Y’) can be made. For now, don’t worry about the math behind the transformation. You may almost never need to manually transform data sets. Just assume that some appropriate transformation from (X, Y) to  (X’, Y’) can make the data linearly separable. |

The shiny app given below will help you visualise the data set in the **original attribute space** and the **transformed feature space**. In the original attribute space, notice that the observations are distributed in a circular fashion. This gives you a hint that the transformation should convert the circular distribution to a linear distribution.

|  |  |
| --- | --- |
|  | Under the **feature space tab** in the app, the transformation given below is used:     * word\_freq\_lottery(X′)=(word\_freq\_lottery(X)−a)2 * word\_freq\_office(Y′)=(word\_freq\_office(Y)−b)2   You can play around with ‘a’ and ‘b’ to change the transformation and observe how the data changes in the new feature space (X’, Y’). For example, If you take **a= 0 and b= 0**, you will get the same feature trasformation i.e (X2,Y2)  from the attribute space i.e (X,Y) as explained by the professer in the lecture. |

The process of transforming the original attributes into a new feature space is called ‘**feature transformation**’. As the number of attributes increases, there is an **exponential increase** in the number of dimensions in the transformed feature space.

|  |  |
| --- | --- |
|  | Suppose we have a quadratic function with attributes (x,y), now we can transform this into 6-Dimentional space and get 6 feature variables. Few of these are non-linear. These x2,xy,y2,x,y,c (cont) attributes we have extracted from the original function and mapped each data point to their corresponding point in the 6-Dimentional space. Once we transform the data points we can run any of our linear algorithm or SVM on the transformed set of data points in 6-Dimentional space and it will give us a separator (hyperplane) on these transformed coordinates. Then we can map it back to the original attribute space. In general this is what we need to do as we do not know the exact functional form of transformed features. |

Similarly, suppose you have four variables in your original data set, then considering only a polynomial transformation with **degree 2**, you end up making **15 features** in the new feature space**,** as shown in the figure below.

Note that the terms xyz, xzw etc. do not appear here because it is a transformation of degree 2, and xyz, xzw etc. are third degree terms.

|  |  |
| --- | --- |
| https://cdn.upgrad.com/UpGrad/temp/b6f3c6fd-45e7-4fa0-8d0f-b63e750ba260/equation_image.PNG | The transformation from the 2-D attribute space resulted in a linearly separable feature space which is 6-D, i.e. the number of dimensions have increased. In theory, this looks fine, but practically a higher number of features will increase the computational cost. |

# The Kernel Trick

Think of a kernel as a **black box**, as shown in the figure below. The original attributes are passed into the black box, and it returns the transformed attributes (in a higher dimensional feature space). The SVM algorithm is shown only the transformed, linear feature space, where it builds the linear classifier as usual.

|  |  |
| --- | --- |
| https://cdn.upgrad.com/UpGrad/temp/3e8f34f3-0202-4c00-941d-4bc1f92c007a/summary.PNG | However, what makes kernels special is that they **don't do this transformation explicitly** (which is a computationally difficult task, as we discussed), but they **use a mathematical hack to do this implicitly.** |

Recall the concept ofthe **inner product (or dot product) of two vectors**. Say there are two vectors A = [2, 3, 5] and B = [1, 0, 4], then the dot product (or the inner product) of A and B is the element-wise multiplication of A and B, i.e. transpose(A). B = A'B = 2.1 + 3.0 + 5.4 = 22. Note that the inner product of two vectors returns a scalar number, in this case 22. Similarly, in any given dataset with n attributes, you can think of each data point (i.e. each row) as a vector of length n. Thus, the dot product of two rows, or observations, will return a scalar value.

You know that kernels transform nonlinear datasets into linear ones **without doing it explicitly**. This is possible because of the fact thatto find a best fit model, **the learning** **algorithm** **only needs the inner products** of the observations XTi.Xj.

Let's understand this through an example. Suppose you have two nonlinearly separable attributes (A1 and A2) along with the label (Y) in your data set, as shown below.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **Row\_name** | **A1** | **A2** | **Y** | | --- | --- | --- | --- | | **X1** | 2 | 4 | 1 | | **X2** | 2 | 7 | 0 | | **X3** | 3 | 1 | 1 | | **X4** | 5 | 8 | 0 | | The SVM algorithm only needs the pairwise inner products of the form (XTi.Xj). For example, in the above data set, the dot product of X1 = (2,4) and X2 = (2, 7) results in a scalar value of  (24).(27)  = (2\*2+4\*7) = 32.    Similarly, one can easily calculate the inner products of all the pairs of data points (XTi.Xj). Note that here, the vectors (XTi.Xj) are in the original nonlinear attribute space. |

Now let's say that phi ϕ is a function which transforms the nonlinear attribute vectors into linear feature vectors, (function phi/phee symbol) ϕ(X1),ϕ(X2),...ϕ(Xn). Now, you can calculate the pairwise inner products of the feature vectors  
ϕ(XTi).ϕ(Xj), using which the learning algorithm can build a model **in the linear feature space**. The SVM() algorithm does not know about the original attribute X1, X2…Xn and it does not care about as well. It works with the transformed attribute in transformed attribute feature space. The original algorithm is running into transformed attribute feature space with vector dot (.) product. The problem is that you do not know what phi ϕ is. For instance, in the example of linear transformation from (x, y) to (x2,y2), the transformation function ϕ is ϕ(x,y)=(x2,y2), but it is harder (or sometimes impossible) to figure out what ϕ is when you have a large number of attributes.

This is exactly where kernels come in  - they bypass the whole business of finding a suitable ϕ using a clever mathematical hack!

|  |  |
| --- | --- |
| For example take any function ‘k’ represent by some exponential function which calculates the Euclidean distance between two data points Xi, Xj. The SVM algorithm only cares about the value of this dot (.) product. We are not really computing the feature vector, taking their dot (.) product and giving it to the SVM algorithm rather we are taking a short cut and computing this Fn and value return by this function is what we give to SVM. | In most of the cases we may not know the actual feature transformation function phi ϕ, but the dot (.) product of such function value is exactly same as the example function **K (kernel function)**.  The kernel Fn value for any pair of vectors is actually equal to the dot (.) product of corresponding feature vectors for some feature transformation Fn pfi ϕ. So to run SVM algorithm to generate the appropriate boundaries (hyperplane) we need to pick up an appropriate kernel Fn.  The example kernel function here is called **RBF** (**Radial Basis Function**). |

Kernel functions use this fact to **bypass the explicit transformation process** from the attribute space to the feature space, and rather **do it implicitly**. The benefit of implicit transformation is that now you do not need to:

* Manually find the mathematical transformation needed to convert a nonlinear to a linear feature space
* Perform computationally heavy transformations

In practice, you only need to know that **kernels are functions** which help you transform non-linear datasets. Given a dataset, you can try various kernels, and choose the one that produces the best model. The three most popular types of kernel functions are:

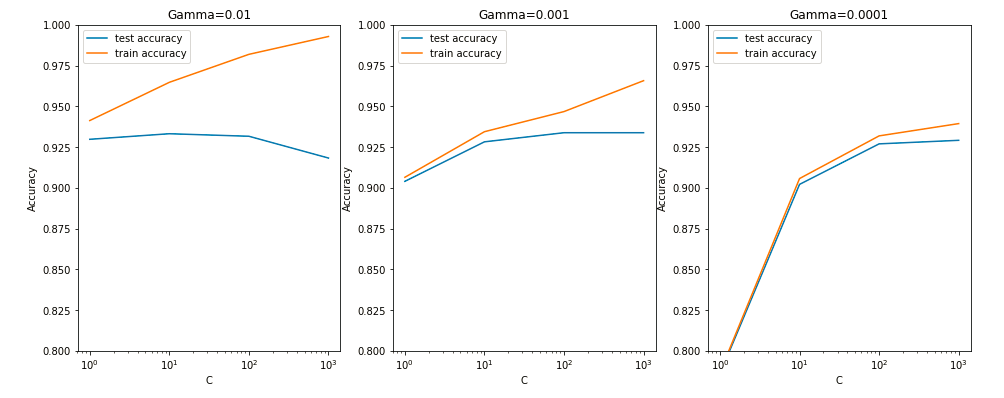
|  |  |  |
| --- | --- | --- |
| https://images.upgrad.com/cca654c4-0834-44ae-be79-3b8a888127d7-12.PNG | **The linear kernel:** This is the same as the support vector classifier, or the same as vanilla hyperplane, without any transformation at all | |
| https://cdn.upgrad.com/UpGrad/temp/377cd38a-7231-492e-9005-01436faca165/polynomial%20kernel.PNG  https://cdn.upgrad.com/UpGrad/temp/e9694930-1ae8-4a1e-9284-68682903cd14/RBF_kernel.PNG | **The polynomial kernel:** It is capable of creating nonlinear, polynomial decision boundaries. The polynomial kernel can produce polynomial shaped nonlinear boundaries  **The radial basis function (RBF) kernel:** This is the most complex one, which is capable of transforming highly nonlinear feature spaces to linear ones. It is even capable of creating elliptical (i.e. enclosed) decision boundaries | |
| In Scikit learn SVM() library as you have learnt, the **linear kernel,** also known as the hyperplane, requires only one tuning parameter, i.e. 'C' to select the best-fit linear model.  # specify range of parameters (C) as a list  params = {"C": [0.1, 1, 10, 100, 1000]}  model = SVC() | | # set up grid search scheme  # note that we are still using the 5 fold CV scheme we set up earlier  model\_cv = GridSearchCV(estimator = model, param\_grid = params, scoring= 'accuracy', cv = folds, verbose = 1,  return\_train\_score=True)  # fit the model - it will fit 5 folds across all values of C  model\_cv.fit(X\_train, y\_train) |

In a **non-linear kernel**, such as the RBF kernel, you'll need to choose two tuning parameters: **gamma** and '**C**'. The hyperparameter **gamma controls  the amount of non-linearity** in the model - as gamma increases, the model becomes more non-linear, and thus model complexity increases.

|  |  |
| --- | --- |
| Also, now since you have two hyperparameters to optimise (C and gamma), you would need to define a range of values of 'C' and gamma. As you would have guessed, **grid search cross-validation** is the best way to choose the **best combination** of these hyperparameters.  # creating a KFold object with 5 splits  folds = KFold(n\_splits = 5, shuffle = True, random\_state = 4)  # specify range of hyperparameters  # Set the parameters by cross-validation  hyper\_params = [ {'gamma': [1e-2, 1e-3, 1e-4],  'C': [1, 10, 100, 1000]}] | # specify model  model = SVC(kernel="rbf")  # set up GridSearchCV()  model\_cv = GridSearchCV(estimator = model,  param\_grid = hyper\_params,  scoring= 'accuracy', cv = folds,  verbose = 1, return\_train\_score=True)  # fit the model  model\_cv.fit(X\_train, y\_train)  # printing the optimal accuracy score and hyperparameters  best\_score = model\_cv.best\_score\_  best\_hyperparams = model\_cv.best\_params\_  print("The best test score is {0} corresponding to hyperparameters {1}".format(best\_score, best\_hyperparams)) |

﻿The best test score is 0.9338509316770186 corresponding to hyperparameters {'C': 100, 'gamma': 0.001}

The image below shows how the training and test accuracies vary with C and gamma. Answer whether true or false based on the image - the model tends to overfit at higher values of gamma (keeping C constant).



  Top of Form

**True - Feedback** : *Yes, at higher values of gamma, the performance (accuracy) on training data is much better than that on test data - a clear sign of overfitting.Bottom of Form*

Increasingly complex (non-linear) models result in a higher training accuracy, though the test accuracy does not increase significantly with increasing model complexity.

**True Feedback** : *Yes, this is correct - as you increase gamma (i.e. make a more complex model), the training accuracy goes up, though the test accuracy does not improve with an equal amount.*

* High values of gamma lead to overfitting (especially at high values of C); note that the training accuracy at gamma=0.01 and C=1000 reaches almost 99%
* The training score increases with higher gamma, though the test scores are comparable (at sufficiently high cost, i.e. C > 10)
* The least amount of overfitting (i.e. the difference between train and test accuracy) occurs at low gamma, i.e. a quite simple non-linear model

﻿Though sklearn suggests the optimal scores mentioned above (gamma=0.001, C=100), one could argue that it is better to choose a simpler, more non-linear model with gamma=0.0001. This is because the optimal values mentioned here are calculated based on the average test accuracy (but not considering subjective parameters such as model complexity).

We can achieve comparable average test accuracy (~92.5%) with gamma=0.0001 as well, though we'll have to increase the cost C for that. So to achieve high accuracy, there's a tradeoff between:

* High gamma (i.e. high non-linearity) and average value of C
* Low gamma (i.e. less non-linearity) and high value of C

We argue that the model will be simpler if it has as less non-linearity as possible, so we choose gamma=0.0001 and a high C=100.

Although you can increase the nonlinearity using kernels, some problems do not require nonlinearity at all and are solved just fine using a linear kernel. Even in a problem decently complex, such as spam detection, you did not need a nonlinear kernel. However, there are datasets that are highly nonlinear, where you will need nonlinear kernels such as RBF.

In nonlinear kernels such as the **RBF**, you use the parameter gamma to control the amount of nonlinearity in the model. The higher the value of gamma, the more is the nonlinearity introduced; the lower the value of gamma, the lesser is the nonlinearity. It is also denoted as sigma in some texts and packages.

Apart from gamma, you also have the hyperparameter C, or the cost (with all types of kernels).

|  |  |
| --- | --- |
| https://cdn.upgrad.com/UpGrad/temp/7b351948-3b91-4261-885e-ca98a551a983/svm.png  Figure-1: RBF Kernels | The plot shows three RBF kernels with different values of gamma. When gamma is high, more nonlinearity is added. When you increase the gamma from 10 to 100, the nonlinearity is further increased, and all the training points are correctly mapped, as is visible in the highly complex decision boundaries in figure 1(C). However, this results in a low biased model that may overfit the training set.    Like most other hyperparameters, it is advisable to tune the value of gamma using cross-validation. |

**Note**: **Please note that the hyperparameter 'C', used in this comprehension refers to the 'C' used in the SVC() function, i.e. higher the C, more complex the model.**

**Overfitting can be controlled by** : *'C', gamma, and the types of kernels, all have an effect on constructing the decision boundary.*

High value of C

Feedback : A higher value of 'C' ( tuning parameter i.e used in SVC() in Python lab) will not allow any points to be misclassified. The SVM model will be overfitting when there is no misclassification.

High value of gamma

Feedback : A higher value of gamma will add more nonlinearity to the decision surface. The SVM model will be overfitting when no points are misclassified, and the nonlinearity is highly introduced than required.