Lecture 7: Support Vector Machine

ENEE 691 – Machine Learning and Photonics @ UMBC

Ergun Simsek, Ph.D. and Masoud Soroush, Ph.D.

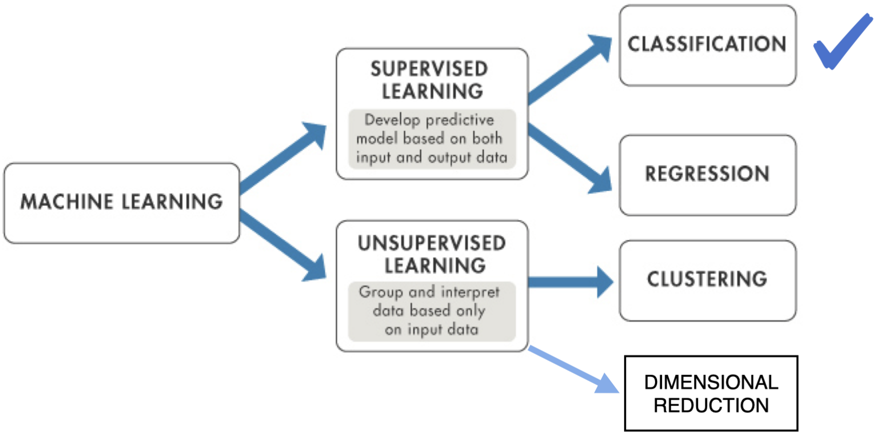
March 12, 2023

1 Support Vector Machine

In today's lecture[1], we introduce another *classification algorithm*, namely the **Support Vector Machine (SVM)**. SVM is one of the most robust prediction methods, being based on statistical learning frameworks. Although SVM is originally a non-probabilistic binary classifier, a clustering algorithm - known as the support vector clustering - applies the statistics of support vectors to categorize unlabeled data in the context of unsupervised learning. The focus of today's lecture is however on binary classifications in the context of supervised learning.

1.1 Background

Last week, we introduced the very first classification algorithm, namely the logistic regression. Recall that classification algorithms belong to the realm of supervised machine learning where the target is a categorical variable.



[1] This product is a property of UMBC, and no distribution is allowed. These notes are solely for your own use, as a registered student in this class.

Today, we focus on a new classification algorithm which takes a different approach than the logistic regression. Recall from last week that logistic regression takes a probabilistic approach toward predicting categorical target variables. In contrast, the support vector machine takes a purely **geometric approach**, and no probability calculation is involved in this classification algorithm. Support vector machine is primarily a *binary classifier*, and in this lecture we will learn how to train support vector classifier and predict the categorical target variable.

1.2 Formulation of Support Vectors

The basic idea behind SVM is to partition the feature space into two components, one for each class of a binary classification problem. The partitioning in the feature space is done in one of the two following ways. In the first method, the feature space is partitioned into two regions by inserting a *hyperplane*. This is the easiest way to partition the feature space, and the boundaries of each region are straight and have no curvature. The classifier that partitions the feature space by inserting a hyperplane is referred to as the **Support Vector Classifier**. In the second partitioning method, the boundaries of the two regions are allowed to have curvature, and the classifier that partitions the feature space in this manner is said to be a **Support Vector Machine**. Note that data scientists are typically not very careful about the distinction between support vector classifiers and support vector machines, and they may use them interchangeably.

To set the stage, let us assume that we have a *binary classification* problem with target variable y, and d continuous features $\vec{x} \in \mathbb{R}^d$. The target variable y takes values $\{-1, +1\}$ corresponding to the two classes. The values of the features $\vec{x}$ and the target y are recorded for n onservations to form a dataset for the statistical learning.

1.2.1 Support Vector Classifier

A support vector classifier simply divides the feature space into two parts by inserting a a hyperplane. A hyperplane in a 2-dimensional space is simply a straight line. In the 3-dimensional case, a hyperplane is simply a Euclidean plane. In higher dimensions ($d > 3$), a hyperplane is a generalization of the concept of plane in three dimensions. Hyperplanes can be conveniently described in terms of the dot product. Recall that for two vectors $\vec{v}$ and $\vec{w}$ in the d-dimensional space $\mathbb{R}^d$, the dot product $\vec{v} \cdot \vec{w}$ is defined by

$$\vec{v} \cdot \vec{w} = v_1 w_1 + v_2 w_2 + \cdots + v_d w_d = \sum_{i=1}^{d} v_i w_i \, . \tag{1}$$

As is evident from (1), the result (output) of the dot product is a *scalar* (*i.e.* a real number). You can easily calculate the dot product through numpy if you will.

Example: Let $\vec{v} = \langle 1, 2, -3, 3, 5 \rangle$ and $\vec{w} = \langle -1, 3, 1.5, 2.5, -4 \rangle$ be vectors in $\mathbb{R}^5$. What is $\vec{v} \cdot \vec{w}$?

[1]:

```
import numpy as np

v = np.array([1, 2, -3, 3, 5])
w = np.array([-1, 3, 1.5, 2.5, -4])

print('v.w =', np.dot(v, w))
```

```
v.w = -12.0
```

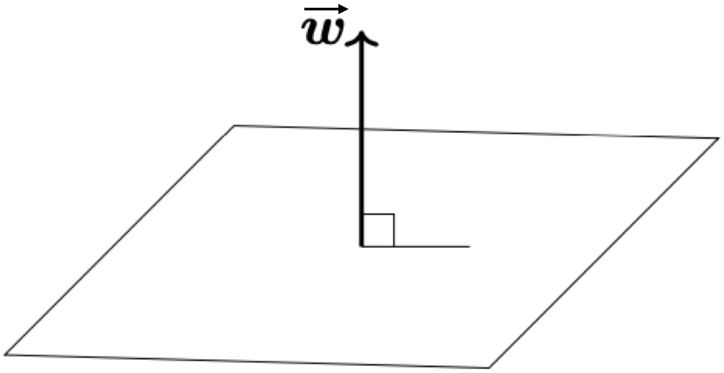
Now let $\vec{w}$ be a fixed vector in the feature space, $\mathbb{R}^d$. We define the following scalar function f

$$
\begin{aligned}
&f: \ \mathbb{R}^d \longrightarrow \mathbb{R} \\
&f(\vec{x}) = \vec{w} \cdot \vec{x} + b \,,
\end{aligned}
\tag{2}
$$

where b is a scalar ($b \in \mathbb{R}$). A hyperplane in $\mathbb{R}^d$ is now defined as the set of all points $\vec{x}$ in feature space $\mathbb{R}^d$ such that $f(\vec{x}) = 0$:

$$
\text{hyperplane } = \left\{ \vec{x} \in \mathbb{R}^d \,|\, f(\vec{x}) = 0 \right\} . \tag{3}
$$

Note that the fixed vector $\vec{w}$ is a vector *normal (perpendicular) to the hyperplane*. The following picture depicts a hyperplane in three dimensions (*i.e.* a Euclidean plane) with the normal vector $\vec{w}$.



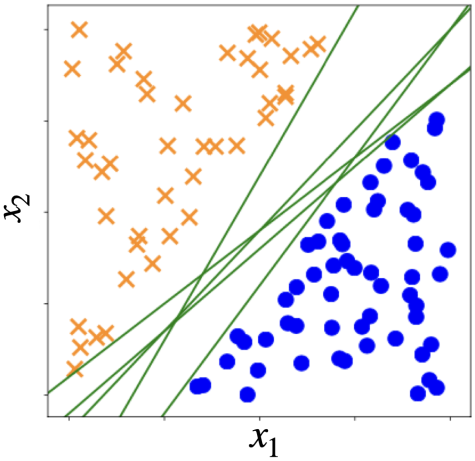
In order to classify an instance $\vec{x}^{(i)}$, we have to see on what side of the hyperplane the instance is located. Note that (3) not only defines a hyperplane, it additionally defines a direction ($\vec{w}$). This direction defines the positive and negative sides of the hyperplane. To classify the i-th example, we calculate $f(\vec{x}^{(i)})$. If $f(\vec{x}^{(i)}) > 0$, then the i-th instance is labeled positive, and if $f(\vec{x}^{(i)}) < 0$, it is labeled negative. When we train the supoort vector classifier, we want to ensure that

$$
\begin{cases}
f(\vec{x}^{(i)}) > 0 & \text{when} \quad y^{(i)} = +1 \,, \\
f(\vec{x}^{(i)}) < 0 & \text{when} \quad y^{(i)} = -1 \,.
\end{cases}
\tag{4}
$$

The two equations above can be combined into one single equation represented below

$$
y^{(i)}\big(f(\vec{x}^{(i)})\big) > 0 \quad \text{for} \quad i = 1, 2, \cdots, n \,. \tag{5}
$$

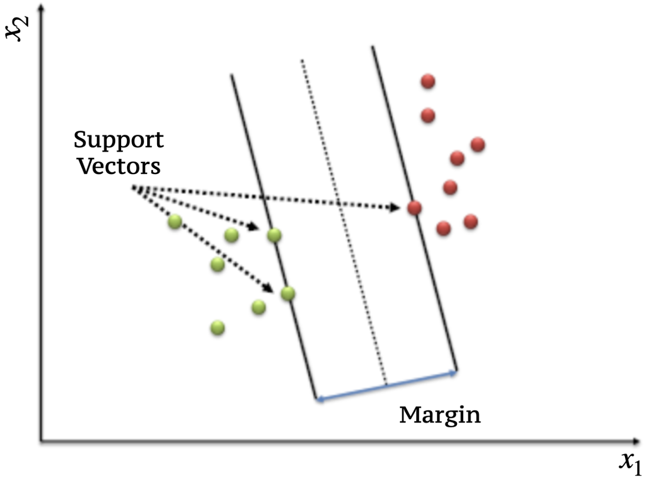
The above requirement (*i.e.* equation (5)) does not determine a hyperplane as a classifier uniquely. To see why this is the case, consider a linearly separable classification problem with only two features x_1 and x_2. In this case, the feature space is a copy of $\mathbb{R}^2$, and a hyperplane in two dimensions simply corresponds a straight line. The following figure depicts the two classes (blue and orange), and the green lines are all hyperplanes that satisfy the requirement in (5).



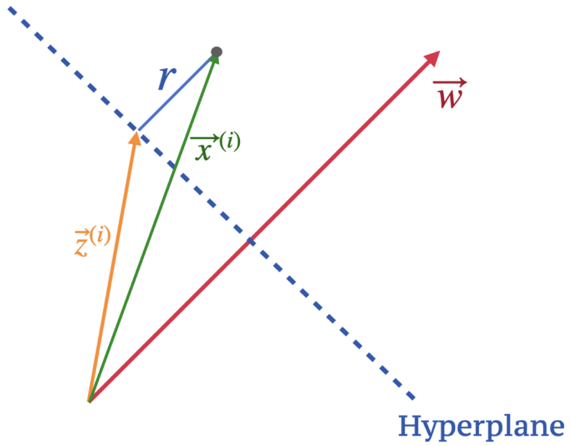
In fact, it turns out that there are infinitely many choices for the classifying hyperplane! The important question now is: **Which hyperplane should we choose?** To answer this question and find a *unique solution*, we first need to come up with a criterion by which we can compare different hyperplanes satisfying (5). We then need to go through an optimization process the pick the best hyperplane.

In order to be able to compare different hyperplanes with one another, we need to define the concept of **margin**. Intuitively, margin is the distance of the separating hyperplane from the closest instance in the dataset. But with this definition, we encounter a conceptual problem! How do we make sense of distance when different features carry different physical dimensions? Consider the following binary classification problem: The features are the weights and heights of individuals, and the target variable is gender. The feature space in this case is a copy of $\mathbb{R}^2$, with the horizontal axis being the weight and the vertical axis being the height. How do we measure distance in this space? Can we naively apply the Euclidean distance formula $\big($*i.e.* $d_{ij} = \sqrt{(x_1^{(i)} - x_1^{(j)})^2 + (x_2^{(i)} - x_2^{(j)})^2}\big)$ for instance? You should note that adding physical quantities with different dimensions is problematic! Moreover, you may decide to change the unit of one of the features (for instance, you may decide to measure the height in terms of meters rather than feet). In that case, one column of your dataset changes, and you find a new dataset (and hence new predictions), whereas the classification problem has not changed at all (it is the same problem)! There are several ways to address this issue. One way to remedy the issue is *data standardization*. Instead of working with $\vec{x}^{(i)}$, we work with the standardized instances $\frac{x_j^{(i)} - \mu_j}{\sigma_j}$ for each feature $j = 1, 2, \cdots, d$, where μ_j and σ_j represent the mean and the standard deviation of the j-th feature, respectively. The standardized instances are all dimensionless, and are independent of the units they are measured in. Moreover, we can apply distance formulas to the standardized dataset with no concerns. For the rest of the discussion, we assume that we are working with the standardized dataset, but we do not introduce any new notations (we simply use $\vec{x}^{(i)}$ to denote instances of the standardized dataset).

Suppose a separating hyperplane is given. The *distance of the closest instance of the dataset to the* *given hyperplane* is said to be the **margin** of the hyperplane. Moreover, each instance of the dataset which possesses the least distance (*i.e.* the margin) to the separating hyperplane is said to be a **support vector**. The following image depicts a two dimensional feature space with a separating hyperplane (dashed line). As is seen in the picture, there are 3 support vectors (two belonging to the green class and one belonging to the red class) in this dataset.



Since the notion of margin will play a central role in the rest of the discussion, we need to see how it is calculated. Suppose a hyperplane and its normal vector $\vec{w}$ are given. Assume that the *i*-th instance, $\vec{x}^{(i)}$, of the dataset is a support vector. Hence, its distance to the hyperplane is the margin (indicated by r in the following image).



We can easily find a formula for the margin r. Note that the blue segment indicating r is parallel with the normal vector $\vec{w}$. Therefore, we have

$$\vec{x}^{(i)} = \vec{z}^{(i)} + r\hat{w} = \vec{z}^{(i)} + r\frac{\vec{w}}{|\vec{w}|} \ , \tag{6}$$

where $\hat{w} = \frac{\vec{w}}{|\vec{w}|}$ is the unit vector along the normal vector $\vec{w}$. Since the margin is the closest distance to the separating hyperplane, we can express the constraints in (5) as

$$y^{(i)}\left(f(\vec{x}^{(i)})\right) \geq r \quad \text{for} \quad i = 1, 2, \cdots, n\,. \tag{7}$$

The concept of margin now allows us to compare different hyperplanes. Between two separating hyperplanes, the better one is the one which comes with the greater margin. More precisely, in order to find the best hyperplane, we should *maximize the margin associated with a separating hyperplane*. Thus, we arrive at the following constrained optimization problem:

$$\begin{aligned} &\underset{\vec{w},b}{\text{argmax}} \quad r \\ &\text{subject to: } \begin{cases} y^{(i)}\left(f(\vec{x}^{(i)})\right) \geq r, & \text{for} \quad i = 1, 2, \cdots, n \\ |\vec{w}| = 1 \\ r > 0 \end{cases} \end{aligned} \tag{8}$$

Note that the role of the normal vector $\vec{w}$ is to merely introduce a direction in the feature space, and its magnitude has no significance. That is why we have set $|\vec{w}| = 1$ in the set of constraints to be fulfilled in (8). We can rewrite the above optimization problem in a slightly different form. For that first perform dot product on the two sides of (6) with vector $\vec{w}$ and add scalar b to get

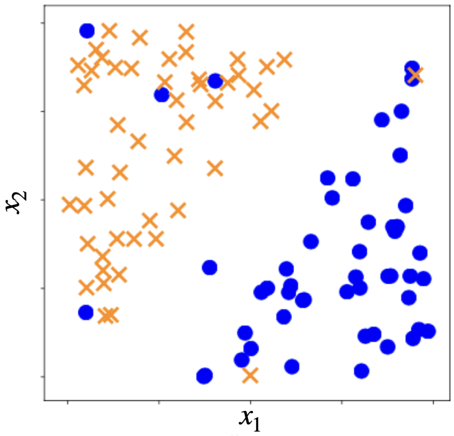
$$\begin{aligned} & \vec{w} \cdot \vec{x}^{(i)} + b = \vec{w} \cdot \left(\vec{z}^{(i)} + r\frac{\vec{w}}{|\vec{w}|}\right) + b = \left(\vec{w} \cdot \vec{z}^{(i)} + b\right) + r\frac{\vec{w} \cdot \vec{w}}{|\vec{w}|} \\ \Longrightarrow \quad & f(\vec{x}^{(i)}) = f(\vec{z}^{(i)}) + r|\vec{w}| \\ \Longrightarrow \quad & r = \frac{C^{(i)}}{|\vec{w}|}\,, \end{aligned} \tag{9}$$

where $C^{(i)}$ is a constant of the support vector (*i.e.* $C^{(i)} = f(\vec{x}^{(i)})$). In getting the final result in (9), we have used the fact that $f(\vec{z}^{(i)}) = 0$, as $\vec{z}^{(i)}$ is a point on the separating hyperplane. Therefore, instead of maximizing r in (8), we can maximize $\frac{1}{|\vec{w}|}$, and relax the constraint $|\vec{w}| = 1$ in (8). Since in machine learning, we typically prefer to minimize a function (rather than maximizing), we can minimize $\frac{1}{2}|\vec{w}|^2$ (instead of maximizing $\frac{1}{|\vec{w}|}$), as the minima of $\frac{1}{2}|\vec{w}|^2$ are exactly the maxima of $\frac{1}{|\vec{w}|}$. One final thing that we can do for further convenience is to divide the two sides of the inequality $y^{(i)}\left(f(\vec{x}^{(i)})\right) \geq r$ in (8) by r, and absorb the r in the definition of the normal vector w and the scalar b. We finally arrive at the following optimization problem:

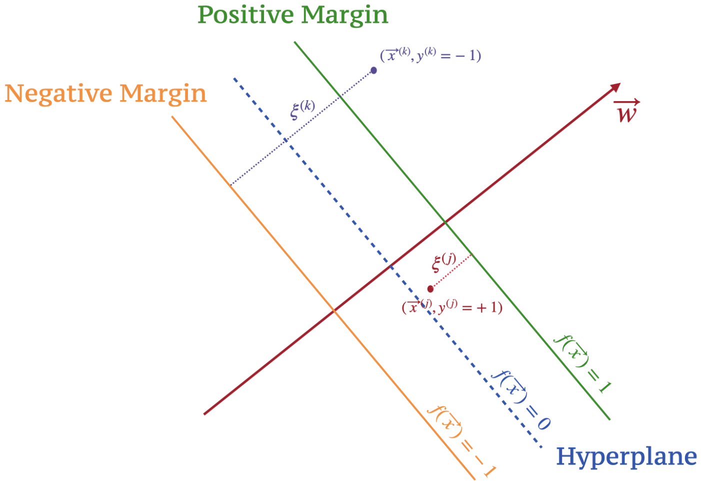
$$\begin{aligned} &\underset{\vec{w},b}{\text{argmin}} \quad \frac{1}{2}|\vec{w}|^2 \\ &\text{subject to: } y^{(i)}\left(f(\vec{x}^{(i)})\right) \geq 1, \quad \text{for} \quad i = 1, 2, \cdots, n\,. \end{aligned} \tag{10}$$

Question: *SVM is computationally a very expensive algorithm*! Looking at equation (10), explain why this is the case.

The above formulation is effective and can offer an optimal separating hyperplane when the dataset associated with the binary classification is *linearly separable*. But what if the dataset is not linearly separable? In that case, there won't exist a hyperplane which separates the two classes perfectly. The following image presents a dataset for a binary classification problem with two features which is *not linearly separable* (*i.e.* there exists no hyperplane which separates the two classes perfectly).



Most of the real-world problems we deal with are actually not linearly separable, and it is important to think how to implement the idea of the support vector machine in those situations. Roughly speaking, to rectify the situation, we would like to develop a *tolerance for misclassifications*, and this brings us to the concept of **soft margin**. The concept of margin we introduced earlier in this section is sometimes called the **hard margin** to distinguish it from the soft one. To formalize the soft margin SVM, we introduce a **slack variable**, $\xi^{(i)}$, associated with each instance $(\vec{x}^{(i)}, y^{(i)})$ of the dataset. Geometrically, $\xi^{(i)}$ measures the distance of the i-th instance in the feature space from the positive margin if the actual label of the i-th instance is positive (*i.e.* $y^{(i)} = +1$), and from the negative margin if the actual label of the i-th instance is negative (*i.e.* $y^{(i)} = -1$).



The slack variable $\xi^{(i)}$ is positive semidefinite (*i.e.* $\xi^{(i)} \geq 0$). If $0 < \xi^{(i)} \leq 1$, then the i-th instance violates the margin, but is still on the right side of the hyperplane. On the other hand, if $\xi^{(i)} > 1$, then the i-th instance is on the right side of the hyperplane. In other words, although $0 < \xi^{(i)} < 1$

does violate the margin, but it does not lead to a misclassification, whereas $\xi^{(i)} > 1$ leads to a misclassification for the algorithm.

Now, with the help of slack variables, we can introduce a new version of the optimization (10) that allows for misclassifications when we deal with nonlinearly separable cases. The new optimization problem in the presence of the slack variables is given by

$$\begin{aligned} &\underset{\vec{w},b,\xi^{(1)},\xi^{(2)},\cdots,\xi^{(n)}}{\text{argmin}} \quad \frac{1}{2}|\vec{w}|^2 \\ &\text{subject to: } \begin{cases} y^{(i)}\big(f(\vec{x}^{(i)})\big) \geq 1 - \xi^{(i)}, \quad \text{for} \quad i = 1, 2, \cdots, n \\ \sum_{i=1}^{n} \xi^{(i)} \leq C\,, \\ \xi^{(i)} \geq 0, \quad \text{for} \quad i = 1, 2, \cdots, n \end{cases} \end{aligned} \tag{11}$$

where C is a hyperparameter that determines the number and severity of the violations to the margin that we will tolerate. In other words, C in (11) is the *total budget for the margin violation*. When $C = 0$, no violation is tolerated and $\xi^{(1)} = \xi^{(2)} = \cdots = \xi^{(n)} = 0$. When $C > 0$, the algorithm learns (through the training process) how to optimally spend the margin violation to get the minimum $|\vec{w}|^2$. The greater C is, the more margin violation is tolerated.

Remark: Note that the optimization problem in (11) is computationally even more costly than (10), as the number of constraints is doubled.

Side Remark: As you have observed, for each of the previous ML algorithms we discussed in this course, we introduced a cost function. You may wonder whether it would be possible to introduce a cost function for the soft margin SVM algorithm (*i.e.* in the presence of the slack variables). It turns out that one can introduce a cost function for the SVM algorithm using the Lagrange multiplier technique. In here, we sketch very briefly the SVM cost function, but we do not attempt to present all technical details. One can incorporate the constraints in (11) into a cost function $\mathcal{L}$ as follows:

$$\mathcal{L}(\vec{w}, b, \xi^{(i)}, \alpha_i, \gamma_i) = \frac{1}{2}|\vec{w}|^2 - \sum_{i=1}^{n} \alpha_i \big(y^{(i)} f(\vec{x}^{(i)}) - 1 + \xi^{(i)}\big) + C \sum_{i=1}^{n} \xi^{(i)} - \sum_{i=1}^{n} \gamma_i \xi^{(i)}\,, \tag{12}$$

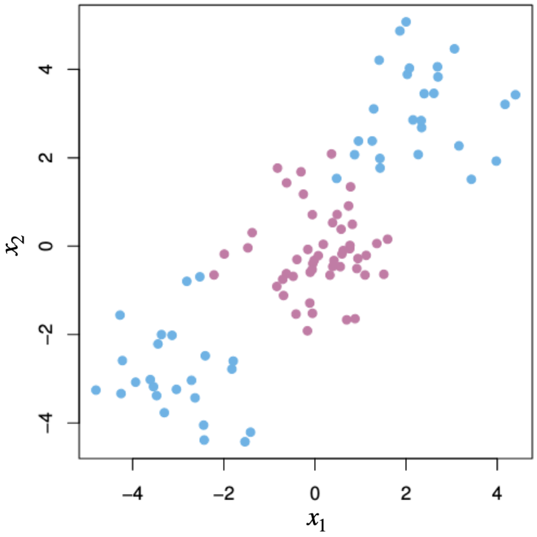
where the last three terms in above incorporate the three constraints in (11). Setting $\frac{\partial \mathcal{L}}{\partial w_i} = 0$, $\frac{\partial \mathcal{L}}{\partial b} = 0$, and $\frac{\partial \mathcal{L}}{\partial \xi^{(i)}} = 0$, we can find a cost function, $\mathcal{D}(\alpha_i, \gamma_i)$, purely in terms of the Lagrange multipliers

$$\begin{aligned} &\mathcal{D}(\alpha_i, \gamma_i) = \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y^{(i)} y^{(j)} \vec{x}^{(i)} \cdot \vec{x}^{(j)} - \sum_{i=i}^{n} \alpha_i\,, \\ &\text{subject to: } \sum_{i=1}^{n} \alpha_i y^{(i)} = 0\,. \end{aligned} \tag{13}$$

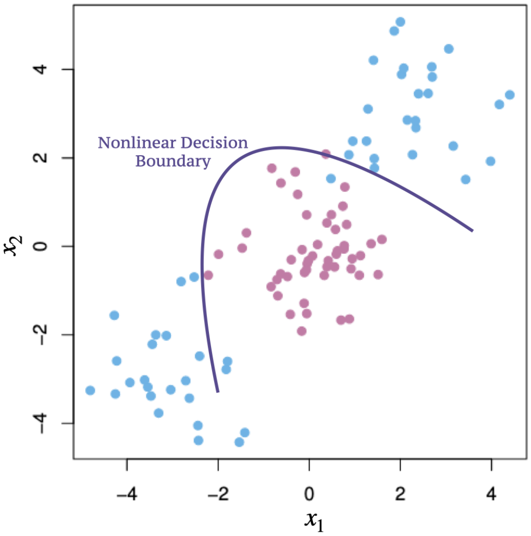
The cost function $\mathcal{D}(\alpha_i, \gamma_i)$ is sometimes referred to as the **dual SVM cost function**. The solution of the dual SVM problem (*i.e.* solutions for α_i and γ_i through minimization of $\mathcal{D}(\alpha_i, \gamma_i)$) would then determine the solution to the original SVM problem (12) through $\vec{w} = \sum_{j=1}^{n} \alpha_j y^{(j)} \vec{x}^{(j)}$. Note that α_j is nonzero *only when the j-th instance is a support vector*.

1.2.2 Support Vector Machine (Kernel Method)

In the previous section, we studied the support vector classifier in which the boundary between classes is given by a hyperplane. However, we often encounter cases where a hyperplane cannot perform an reasonably well classification job, exactly because the boundary between the two classes is linear. The following image depicts a binary dataset with two features x_1 and x_2 that is not linearly separable.



As observed above, no hyperplane (in this case a straight line) can perform a reasonable classification job even when margin violation is allowed (through slack variables). On the other hand, if we allow for nonlinear boundaries between classes, the above binary classification can be immediately rectified. The following image depicts a reasonably well classifier with nonlinear boundary between classes for the above dataset.



There are many ways to incorporate nonlinearity in the definition of boundaries between different classes. Support vector machine involves nonlinearity in the defining boundaries in a peculiar way, known as the kernel method. The key observation is that in the case of the support vector classifier, the defining equation of the hyperplane (3) is solely based on concept of *dot product*. The dot product is a special type of a more general class of products, known as the **inner product**. The *kernel trick* allows one to consider higher monomials in the feature space without explicit calculations of coordinates, but rather by simply computing the inner products between the images of all pairs of data in the feature space. The kernel trick is *computationally much cheaper* than the explicit computation of the coordinates! Now, let us see how the kernel method works. First, we define a function g which considers the dot product of the argument vector $\vec{x}$ with all data points as follows:

$$\begin{aligned} &g : \mathbb{R}^d \longrightarrow \mathbb{R} \\ &g(\vec{x}) = b + \sum_{i=1}^{n} \alpha_i \, \vec{x} \cdot \vec{x}^{(i)} = b + \vec{x} \cdot \big(\sum_{i=1}^{n} \alpha_i \, \vec{x}^{(i)} \big) \, . \end{aligned} \tag{14}$$

Note that (14) reduces to defining equation of the hyperplane (3) upon identification $\vec{w} = \sum_{i=1}^{n} \alpha_i \, \vec{x}^{(i)}$. Then we define the *linear*, *polynomial*, *radial*, and *sigmoid* kernel functions, $K : \mathbb{R}^d \times \mathbb{R}^d \longrightarrow \mathbb{R}$,for a pair of vectors $\vec{x}^{(i)}$ and $\vec{x}^{(j)}$

$$\begin{aligned} &K_{lin}(\vec{x}^{(i)}, \vec{x}^{(j)}) = \vec{x}^{(i)} \cdot \vec{x}^{(j)} \, , \\ &K_{poly}(\vec{x}^{(i)}, \vec{x}^{(j)}) = \big(r + \gamma \, \vec{x}^{(i)} \cdot \vec{x}^{(j)} \big)^{\ell} \, , \\ &K_{rad}(\vec{x}^{(i)}, \vec{x}^{(j)}) = e^{-\gamma |\vec{x}^{(i)} - \vec{x}^{(j)}|^2} \, , \\ &K_{sig}(\vec{x}^{(i)}, \vec{x}^{(j)}) = \tanh(r + \gamma \, \vec{x}^{(i)} \cdot \vec{x}^{(j)}) \, , \end{aligned} \tag{15}$$

where γ in the radial kernel is a *positive constant* (hyperparameter), and for the rest of the kernels, γ and r are real-valued hyperparameters. Now to define the boundary between classes, we generalize the function f in (2) to

$$\begin{aligned} f_{nonlin} :\ & \mathbb{R}^d \longrightarrow \mathbb{R} \\ f_{nonlin}(\vec{x}) &= b + \sum_{i=1}^{n} \alpha_i K(\vec{x}, \vec{x}^{(i)}) \ , \end{aligned} \tag{16}$$

where α_i are constants (We typically set $\alpha_i \neq 0$ only for the support vectors). The nonlinear boundary between classes is then defined by the same equation as (3) except f is replaced by f_{nonlin} defined in (16)

$$\text{boundary between classes } = \left\{ \vec{x} \in \mathbb{R}^d \mid f_{nonlin}(\vec{x}) = 0 \right\} . \tag{17}$$

There are a couple of comments in order in regards to different kernels introduced above:

1. The linear kernel, K_{lin}, is equivalent to the support vector classifier where the boundary between different classes is introduced by a hyperplane.

2. The polynomial kernel, K_{poly}, is a polynomial function of degree ℓ (Note that for $\ell = 1$, K_{poly} reduces to K_{lin}). Note that the greater the dot product $\vec{x}^{(i)} \cdot \vec{x}^{(j)}$ is, the greater the value of the kernel function will be.

3. The radial kernel works in a local manner: When the Euclidean distance between the pair of points $\vec{x}^{(i)}$ and $\vec{x}^{(j)}$ is large, the value of the radial kernel function is negligible. This implies that for a test point $\vec{x}^*$, the farther points to $\vec{x}^*$ do not play any role in predicting the class of $\vec{x}^*$ (Recall that the class of $\vec{x}^*$ is determined by the sign of $f_{nonlin}(\vec{x}^*)$). This means that the radial kernel has a very *local behavior*, and only nearby observations to a test point play a role in determining the class of the test point.

4. The sigmoid kernel is used when one desires to restrict the kernel function to vary in a bounded interval (Recall that $\tanh x \in (-1, 1),\ \forall x \in \mathbb{R}$).

1.3 Some Final Remarks on SVM

We conclude this theory part of the lecture by some useful remarks on SVM. The very first point to wonder is whether there exists a more generalized version of the SVM algorithm that can be applied to multinomial classification problems. Unfortunately, the concept of separating hyperplanes does not lend itself naturally to more than two classes. Nonetheless, a number of simple proposals have been made to extend SVM to the multinomial classification problems. In here, we mention the two most popular generalizations of SVM. Now, assume that we are dealing with a multinomial classification problem with K classes.

One-vs-one: In this approach, one considers all possible pairs of classes (*i.e.* $\frac{1}{2}K(K-1)$ distinct pairs of classes). For each pair, one constructs a binary SVM classifier. A test observation is classified using each of the $\frac{1}{2}K(K-1)$ SVM classifiers, and we tally the number of times that the test observation is assigned to each of the K classes. The final class of the test observation is determined through a voting process (*i.e.* the most frequently assigned class).

One-vs-all: In this approach, we first choose a class k (from the total K classes). We then construct an SVM classifier to decide whether a test observation belongs to class k or to the rest of $K-1$

classes (Note that this is a binary classification, and SVM can be readily applied). We can repeat this process for any of the K classes by constructing K SVM classifiers. Note that each of the K SVM classifiers has its own separating hyperplane $f_k(\vec{x})$. To assign the ultimate class of a test point $\vec{x}^*$, we calculate $f_k(\vec{x}^*)$ for $k = 1, 2, \cdots, K$. The final class of $\vec{x}^*$ is the class that leads to the greatest value for $f_k(\vec{x}^*)$.

Remark: Note that if SVM is used for a multinomial classification problem in scikit-learn, then the library automatically chooses the *one-vs-one* approach to complete the multinomial classification.

Finally, let's end this section with some of the *disadvantages of the SVM algorithm* that you should have in mind:

- SVM is computationally very expensive, and hence it is not a suitable classification algorithm for very large datasets.
- SVM does not perform well when there is a considerable amount of noise in the dataset.
- SVM typically underperforms when the number instances of the training dataset is less than the number of features.
- For situations where a probabilistic approach is more suitable, SVM cannot be helpful, as it takes a purely geometric approach towards the classification problem.