In [4]:

```
from IPython.core.display import display, HTML
display(HTML("<style>.container { width:65% !important; }</style>"))
```

Primer to Support Vector Machines

By Dr. Jonne Pohjankukka

- Note, that github does not necessarily render all the equations in this tutorial correctly. To guarantee best presentation, open the .ipynb-file in Jupyter notebook.

Preface: to whom is this tutorial for?

I think it was back in 2014 or 2015 when I first came across with support vector machines and I remember being frustrated for the difficulty of finding good practical tutorials on the subject. I could find many tutorials online describing the theory and general ideas, but everytime after reading them I had one question: okay, now what? What do I type in my IDE? I could understand the idea, but when I needed to do some programming I found out I actually did not understand it that well. So what I did next was to search again online for some advices, and I found out these guides repeating themselves: "use a package", "download and apply a package". I understand that this is a perfectly valid solution if your goal is simply to apply support vector machines. In my case, I wanted to understand **every step going on under the hood**. I wanted to understand how one constructs a support vector machine model, how does one train it et cetera **without using any packages at all** (excluding trivial packages of course such as NumPy). How does one implement the model, the mathematical optimization, kernel tricks, everything by yourself?

This tutorial is an attempt to provide readers a primer into the subject who have similar problems and frustrations that I had when first running into the subject. I will provide the reader with the general idea of support vector machines, its basic theretical background, practical pen-and-paper examples, pseudocode and Python codes which do not apply any packages related to machines learning or mathematical optimization. I will assume the reader has some understanding on probability, linear algebra and mathematical optimization. Enjoy the ride =)

1. Machine learning

What is machine learning (ML)? ML is a subfield of computer science focused on the research and desing of models, which aim to discover and learn patterns from data. Applications of ML could be for example the prediction of stock price values, classification of soil bearing capacity, or forecasting the effects of drinking milk to the acidity levels in human stomach. ML combines techniques from many fields of science, such as probability theory, statistics, physics, mathematical optimization and neuroscience.

One of the most important (or maybe better say **the most important**) issues in ML is the concept of generalizability, which measures how well a ML model performs in making predictions in new situations (that is, with new data). A model probably fails to generalize well, if it is "fitted" too much to the data (this claim is also backed up theoretically). The notion of "fitting a model to data", usually means that we minimize some error function, which describes the goodness-of-fit of our model to the data. The lower the error, the better the fit. In fields such as mathmetical optimization or calculus of variation, the goal is many times to find the absolute minimum of this error (say, the optimum trajectory of a particle). It has however been shown, both from a theoretical and practical perspective, that if you train a model too much *overfitting* will occur.

Overfitting means that the model has learned not only the intrinsic phenomena in the data, but also an additional non-existing relationship called *noise*, which can not be learned by definition. Noise is present in all data you ever measure and can be caused e.g. by measurement errors, weather or malfunctioning sensors. Thus by overfitting a model, we have learned an incorrent relationship from the data, and are more likely to generalize worse. A common way to tackel overfitting is to apply a method known as *regularization*, which basically means restricting the learning process by preventing it from learning too complicated functions. There are many ways to tackle overfitting such as using penalty term or early-stopping methods, but we will not go deeper into this subject in this tutorial. Readers interested with overfitting can find more information from standard ML literature.

2. Linear vs. nonlinear models

All methods of ML can be divided into two groups: linear or nonlinear. Linear models are simple and effective methods in many applications describing real world phenomena, but sometimes their expressive power is not enough to learn more complicated relationships in the data. In cases like this, nonlinear methods are usually applied due to their higher expressive power. However, due to its simplicity a linear model is less likely to overfit than nonlinear model. Also in general, nonlinear models require more data than linear models to achieve succesful generalization. There is therefore a trade-off between the expressive power of a model and its likelihood of overfitting to the data. Because of the higher expressive power, nonlinear models are more easily fitted to the noise in the data. With this in mind, it begs now the question: does there exist a model which contains both the resistance towards noise (as in linear models) and high expressive power (as in nonlinear models). A clever method called *support vector machines* (SVM) was proposed for achieving this by Vladimir Vapnik and Alexey Chervonenkis in 1963, which we will discuss next. In what follows, we will go through the motivation, theory, examples and a self-made implementation (in pseudo- and Python code) of the SVM method.

3. Support vector machines

We will begin with a simple geometric illustration, which best explains the intuition behind SVM. In figure 1 is presented data from two different classes denoted by blue crosses and red circles. The lines depict three competing decision lines which we use to determine the classification regions. The data is *linearly separable*

which means that the data can be divided into two distinct subspaces by the line (or a hyperplane in higher dimensions). Which of these three lines would the best choice and why? Or would you say they are all equally good? After all, all the lines perfectly classify the data. I'm betting however, that you would probably select the line in the rightmost image. Why would we choose this line? What is the intuition behind our choice? Remember that all data we ever measure contains noise. It would be good therefore to select such a decision line which would be most robust against this noise. You can think of the effect of noise in the four data points by shifting them randomly into arbitrary directions a tiny bit. We would want the decision line to allow as much of this random shifting (due to noise) as possible and still achieve correct classification. You would agree that decision line in the rightmost image allows the largest amount of random shifting in the points in the image, right?

In [31]:

```
# Make Figure 1 plot
drawClassifierLines(showRadius=False)
```

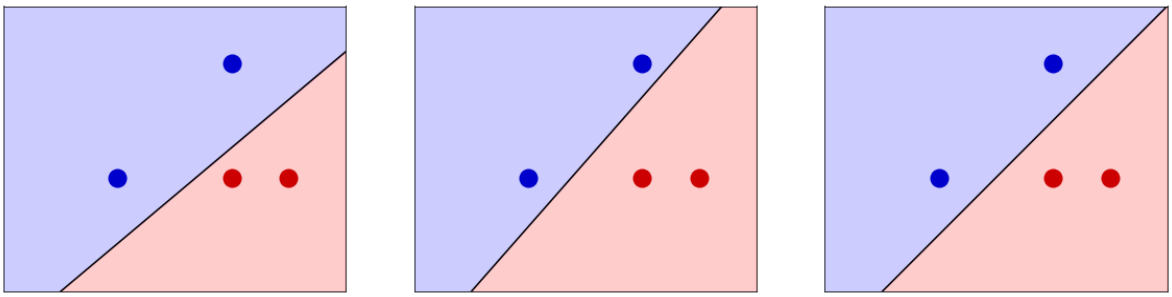


Figure 1: Geometric intuition behind the SVM. Which line is the best classifier?

As we discussed, we would like to select a decision line which allows as much as possible this random shifting in the data (i.e. noise) without affecting the classification. We can picture this random shifting caused by noise in terms of circles or spheres with radius *r* around the data points (see figure 2). Notice that in the rightmost image the circles have the largest radius, and so we can think that the data points are allowed to move within this circle (i.e. we have uncertainty) and still we get correct classification. What we would like to do, is to select a linear classifier which maximizes the radius of these spheres. In this way, we have maximized the model's robustness against noise in the data. Notice that in the rightmost image, the distance from the line to the closest data points is maximized. The data points at which the spheres first touch the decision line are called *support vectors*, from which the name of the SVM comes from. The support vectors play a special role in the SVM since they are solely responsible in determining the classifier line. If we would for example remove the rightmost data point from the below images, it would not affect the choice of the classifier line, only the support vectors have impact on this. Lastly, since in SVM the point is to maximize the distance of the line to the support vectors (called the *margin*), SVMs are also called *maximum margin models*.

In [32]:

```
# Make Figure 2 plot
drawClassifierLines(showRadius=True)
```

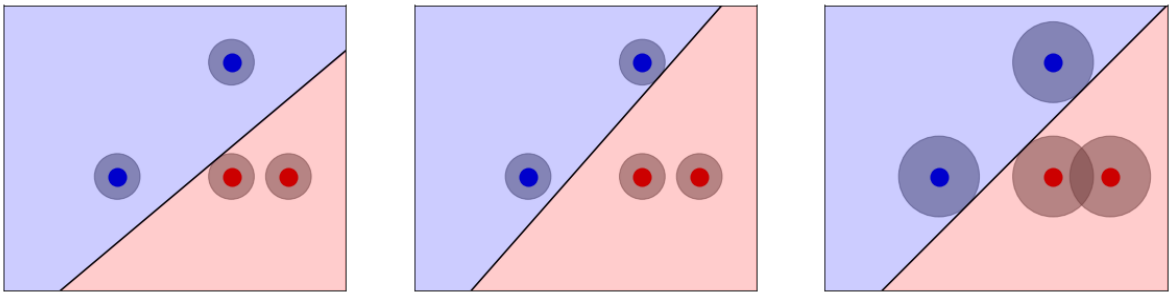


Figure 2: Uncertainties around the data points in figure 1 illustrated. The line in the rightmost plot has the highest tolerance against uncertainty (noise) in the data.

3.1 How to find the maximum margin model?

Lets now proceed to formalize the concepts of the SVM and find out how can we solve the maximum margin hyperplane. Denote by $\mathbf{x} \in \mathcal{X} \subset \mathbb{R}^d$ an input vector, $\mathbf{w} \in \mathbb{R}^d$ as hyperplane weight values, $y \in \{-1, 1\}$ as label value and $b \in \mathbb{R}$ as a constant intercept term. A hyperplane h defined by vector $\mathbf{w}$ and constant b separates the data points $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)$ if and only if:

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) > 0 \quad (i = 1, 2, \ldots, n), \qquad (1)$$

where $h(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$ is called the *signal* of input $\mathbf{x}$. That is, for data points having $y = -1$ we want the hyperplane to have a negative signal $h(\mathbf{x}) < 0$ and for data points with $y = 1$ we want $h(\mathbf{x}) > 0$ correspondingly. Notice next that the equations in (1) are always satisfied also for any $h(\mathbf{x})/\rho$, where $\rho > 0$. Define next (for reasons later coming clear):

$$\rho := \min_{i=1,2,\ldots,n} y_i(\mathbf{w}^T\mathbf{x}_i + b),$$

and redefine the separating hyperplane as $h(\mathbf{x})/\rho$. For this redefined hyperplane we have:

$$\min_{i=1,2,\ldots,n} y_i \left(\frac{h(\mathbf{x})}{\rho}\right) = \min_{i=1,2,\ldots,n} y_i \left(\frac{\mathbf{w}^T\mathbf{x}_i}{\rho} + \frac{b}{\rho}\right) = \frac{1}{\rho} \min_{i=1,2,\ldots,n} y_i \left(\mathbf{w}^T\mathbf{x}_i + b\right) = \frac{\rho}{\rho} = 1,$$

that is, the hyperplane separates all the data points if and only if:

$$\min_{i=1,2,\ldots,n} y_i \left(\mathbf{w}^T\mathbf{x}_i + b\right) = 1. \qquad (2)$$

So we have now learned that a hyperplane h separates the data points only if the condition in (2) is met. This condition is not yet though sufficient enough to define the problem of finding a maximum margin hyperplane. Recall that one of the defining factors of the SVM was also the fact that the margin (distance to the closest, a.k.a support vectors) needs to maximized. In other words, we want to maximize the distance between the hyperplane h and the vector $\mathbf{x}$ closest to it.

To start, lets first figure out how one generally calculates the distance between a hyperplane and a vector point. To solve this distance we need to calculate the perpendicular distance between h and $\mathbf{x}$. Let $\mathbf{u}$ be a unit vector perpendicular to h and $\mathbf{x}'$ some point on h, i.e. $h(\mathbf{x}') = \mathbf{w}^T\mathbf{x}' + b = 0$. Then, from basic linear algebra (make e.g. a Google search) we know that the distance between h and point $\mathbf{x}$ is the projection $d(h, \mathbf{x}) = |\mathbf{u}^T(\mathbf{x} - \mathbf{x}')|$. Note that $\mathbf{w}$ is perpendicular to the plane h, since for two points $\mathbf{x}', \mathbf{x}''$ on the plane h we have:

$$\mathbf{w}^T(\mathbf{x}'' - \mathbf{x}') = \mathbf{w}^T\mathbf{x}'' - \mathbf{w}^T\mathbf{x}' = -b + b = 0,$$

that is $\mathbf{w}$ is perpendicular to any vector $\mathbf{x}'' - \mathbf{x}'$ on h. We can now set $\mathbf{u} = \mathbf{w}/||\mathbf{w}||$, where $||\mathbf{w}||$ is the magnitude of vector $\mathbf{w}$. Hence, we get the distance from an arbitrary point $\mathbf{x}$ to plane h as:

$$d(h, \mathbf{x}) = \frac{\left|\mathbf{w}^T(\mathbf{x} - \mathbf{x}')\right|}{||\mathbf{w}||} = \frac{\left|\mathbf{w}^T\mathbf{x} - \mathbf{w}^T\mathbf{x}'\right|}{||\mathbf{w}||} = \frac{\left|\mathbf{w}^T\mathbf{x} + b\right|}{||\mathbf{w}||}.$$

Furthermore, since for a binary SVM classifier we have $y_i \in \{-1, 1\} \forall i$, we get:

$$\left|\mathbf{w}^T\mathbf{x}_i + b\right| = y_i(\mathbf{w}^T\mathbf{x}_i + b) \ \forall i,$$

and because of (2) we have that:

$$\min_{i=1,2,...,n} d(h, \mathbf{x}_i) = \min_{i=1,2,...,n} \frac{y_i(\mathbf{w}^T\mathbf{x}_i + b)}{||\mathbf{w}||} = \frac{1}{||\mathbf{w}||} \min_{i=1,2,...,n} y_i(\mathbf{w}^T\mathbf{x}_i + b) = \frac{1}{||\mathbf{w}||}. \quad (3)$$

The nice and simple form of equation (3) is the reason why we redefined the hyperplane h earlier by scaling it with the constant ρ, giving us thus a nice numerator of 1. We have now found all the ingredients we need to define the problem of solving the maximum margin model: A SVM model (binary classification) is such a hyperplane, which maximizes the value of $1/||\mathbf{w}||$ (margin) and satisfies the condition in equation (2). This can expressed in the following optimization problem:

$$\begin{aligned} &\underset{\mathbf{w},b}{\text{minimize:}} && \frac{1}{2}\mathbf{w}^T\mathbf{w} \\ &\text{subject to:} && \min_{i=1,2,...,n} y_i\left(\mathbf{w}^T\mathbf{x}_i + b\right) = 1. \end{aligned} \quad (4)$$

From the perspective of mathematical optimization, it is easier to solve the problem:

$$\begin{aligned} &\underset{\mathbf{w},b}{\text{minimize:}} && \frac{1}{2}\mathbf{w}^T\mathbf{w} \\ &\text{subject to:} && y_i\left(\mathbf{w}^T\mathbf{x}_i + b\right) \geq 1, \quad (i = 1, 2, \ldots, n) \end{aligned} \quad (5)$$

which is equivalent to problem (4) at the optimal solution, given that the data set contains samples with negative and positive labels. To show that equations (4) and (5) are equivalent at the optimum, suppose the solution $(\mathbf{w}^*, b^*)$ of equation (5) has:

$$\rho^* = \min_{i=1,2,...,n} y_i\left(\mathbf{w}^{*T}\mathbf{x}_i + b^*\right) > 1,$$

which means that $(\mathbf{w}^*, b^*)$ is not a solution to equation (4). Consider now a hyperplane $(\mathbf{w}, b) = \frac{1}{\rho^*}(\mathbf{w}^*, b^*)$ which satisfies the constraints in equation (5). But now we have that $||\mathbf{w}|| = \frac{1}{\rho^*}||\mathbf{w}^*|| < ||\mathbf{w}^*||$, which means that $\mathbf{w}^*$ can not be optimal for equation (5) unless $\mathbf{w}^* = \mathbf{0}$. This however is not possible, because the hyperplane $(\mathbf{0}, b)$ can not correctly classify the negative and positive data samples □. So, once we have solved the optimal solution $(\mathbf{w}^*, b^*)$ to equation (5), we get the maximum margin classifier model (a.k.a SVM) as:

$$g(\mathbf{x}) = \text{sign}\left(\mathbf{w}^{*T}\mathbf{x} + b^*\right) \in \{-1, 1\}. \quad (6)$$

In the upcoming sections, we will see how to solve the optimization problem of equation (5). First however, we will have a short discussion on why the maximum margin model is better than a non-maximum margin model in terms of generalizability.

3.2 Is maximum margin model really better?

In earlier sections, we had the intuition that a linear classifier with a larger margin would probably perform better than one with a smaller margin. In fact, our intuition is justified also mathematically by a special number called the *Vapnik-Chervonenkis-dimension* $d_{\text{VC}} \in \mathbb{N}_{>0}$ (VC-dimension), which quantifies a probabilistic bound for the classification error of the SVM model. Generally speaking, a model with a smaller VC-dimension is more likely to achieve succesful generalization than one with a higher VC-dimension. And it is indeed the case (see the works of e.g. V. Vapnik), that the VC-dimension of a SVM model is smaller than the VC-dimension of an unrestricted linear classifier (unrestricted by the margin that is). This fact is a result from an inequality called the *VC-inequality* (related closely to the Hoeffding inequality), which states that:

$$P\left[\sup_{h\in\mathcal{H}} |E_{\text{in}}(h) - E_{\text{out}}(h)| > \varepsilon\right] \leq 4m_{\mathcal{H}}(2n)\exp\left(-\frac{1}{8}\varepsilon^2 n\right), \quad (7)$$

where $\mathcal{H}$ is the hypothesis set, that is, a set of functions from which we are searching for the model h, $\varepsilon > 0$ is an error bound and $n \in \mathbb{N}_{>0}$ is the number of data points. The functions $E_{\text{in}}(h), E_{\text{out}}(h) : \mathcal{H} \to \mathbb{R}$ denote the training and generalization errors of hypothesis h. That is, $E_{\text{in}}(h)$ describes how well we were able to fit model h to the observed data $\mathcal{D} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\}$, and $E_{\text{out}}(h)$ describes how well the model h performs with new yet unseen data. Our goal is to have $E_{\text{in}}(h) \approx E_{\text{out}}(h)$, because in this case the we can trust that our model's performance estimated with the data set $\mathcal{D}$ reflects its performance in a general situation.

The function $m_{\mathcal{H}}(2n) : \mathbb{N}_{>0} \to \mathbb{N}$ maps $2n$ to a number, which describes the maximum number of dichotomies that can be generated by the hypothesis set $\mathcal{H}$ on any $2n$ data points. In other words, it gives a quantifying number on how many ways the hypothesis set $\mathcal{H}$ can split $2n$ data points into two categories. If $m_{\mathcal{H}}(c) = 2^c$ for c data points, then we say that the hypothesis set $\mathcal{H}$ is able *shatter* the c data points, that is find all possible dichotomies (classifications) for the data. To put the relationship between d_{VC} and $m_{\mathcal{H}}(c)$ explicit, the VC-dimension is defined as $d_{\text{VC}} \equiv \max\{c \in \mathbb{N} \mid m_{\mathcal{H}}(c) = 2^c\}$. In other words, VC-dimension is the maximum number of data points the hypothesis set $\mathcal{H}$ is able to shatter. The next question now is, what has the VC-dimension got to do with the VC-inequality? It is known, that if a hypothesis set $\mathcal{H}$ has a finite d_{VC}, then it holds that:

$$m_{\mathcal{H}}(2n) \leq \sum_{i=0}^{d_{\text{VC}}} \binom{2n}{i},$$

and therefore a hypothesis set with a smaller VC-dimension has a smaller multiplying factor in the right side of equation (7) resulting in a smaller probability bound. This is also somewhat intuitive if you think about. If you are using a hypothesis set with unrestricted (by the margin) classifier models, then you would probably be able to find more dichotomies for the data set $\mathcal{D}$, than with the hypothesis set consisting from models with the restriction to maximize the margin. With a large number of equally good hypotheses to choose from, you would have smaller chances (a higher probability bound in the VC-inequality) to pick the right one unless you are very lucky.

3.3 What if data is not linearly separable?

So far, we have been talking about cases where the data set can be separated by a linear model. In many practical situations however the data set can not be separated by a linear model. In Figure 3 I have illustrated two examples of data sets which can not be separated by a linear model.

In [80]:

```
# Make Figure 3 plot
drawNonSeparableCases()
```

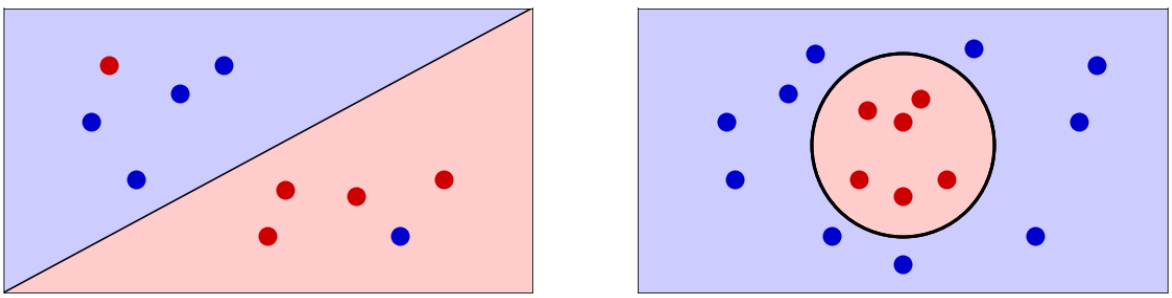


Figure 3: Data which is not linearly separable. In both plots a nonlinear classifier is clearly needed. In the right plot we see a circular classifier.

So how should we proceed from this? Well, fortunately we two options two consider. Firstly, we can loosen up the condition on the SVM which requires all the data points to be classified correctly. We do this by allowing few misclassifications for the model to occur. An SVM with a looser condition like this is called the *soft margin SVM*,

which does not force a perfect classification and allows some mistakes to occur. On the left side of figure 3, we see an example of a soft margin SVM model. Our second option to the separability problem, is to first apply a nonlinear transformation to the data making it linearly separable, and then fit a hard margin linear SVM into this transformed data. In the right side of figure 3 is an example of this second approach. Note that the nonlinear model you seen here is a representation of the linear model of the transformed space in the original data space. The nonlinear transformation is achieved by a function $\Phi : \mathcal{X} \to \mathcal{Z} \subset \mathbb{R}^q$, which transforms the input vectors into $\mathbf{z}_i = \Phi(\mathbf{x}_i)$. After the transformation, the optimization problem in equation (5) becomes:

$$\begin{aligned} &\underset{\mathbf{w},b}{\text{minimize:}} && \frac{1}{2}\mathbf{w}^T\mathbf{w} \\ &\text{subject to:} && y_i\left(\mathbf{w}^T\mathbf{z}_i + b\right) \geq 1, \quad (i = 1, 2, \ldots, n), \end{aligned} \tag{8}$$

and similarly to equation (6) we get the nonlinear hard margin SVM model as:

$$g(\mathbf{x}) = \text{sign}\left(\mathbf{w}^{*T}\Phi(\mathbf{x}) + b^*\right) \in \{-1, 1\}. \tag{9}$$

What happens to the generalization capability if we use nonlinear transformation for the data? Does it get worse? In general, the price we pay for using nonlinear models with higher expressive power is that we have a larger risk to overfit the data and that the probability of successful generalization decreases. Fortunately, a neat mathematical theorem exists (see e.g. Mostafa et al. for proofs) which helps us tackle the generalization problem of SVMs using nonlinear transformation. The theorem states that:

Theorem: the VC-dimension of maximum margin classifier SVM with bounded input data $||\mathbf{x}|| \leq R$ and margin $r \in \mathbb{R}^+$ follows the inequality:

$$d_{\text{VC}} \leq \lceil R^2/r^2 \rceil + 1,$$

regardless on the nonlinear transformation Φ. This means that even if SVM incorporates an infinite dimensional transformation, generalization is achieved as long as we use large enough margin r.

3.4 Kernel methods

In earlier sections, we went through formulating the mathematical optimization problem of solving a SVM model. This problem was presented explicitly in equations (5) and (8). It will in later sections become clear that when solving these problems, we need to calculate inner products $\mathbf{x}_i^T\mathbf{x}_j$ (linear SVM) and $\mathbf{z}_i^T\mathbf{z}_j = \Phi(\mathbf{x}_i)^T\Phi(\mathbf{x}_j)$ (nonlinear SVM). This seems simple enough, as it is in many cases but one can find transformations Φ which map the inputs $\mathbf{x}$ into infinite dimensional vectors (i.e. $d = \infty$). It is obvious, that we can not calculate the inner products of infinite dimensional vectors $\mathbf{z}$, since this would require us first to explicitly calculate the infinite vectors $\mathbf{z}$. Fortunately though, there exists cool functions which allow us to calculate these infinite inner products without explicitly knowing the vectors $\mathbf{z}$. These functions are called *kernel functions*, which describe the inner products of vectors in some (possibly infinite dimensional) space $\mathcal{Z}$. To be explicit, the kernel functions defined by transformation Φ are defined as:

$$K_\Phi(\mathbf{x}, \mathbf{x}') \equiv \Phi(\mathbf{x})^T\Phi(\mathbf{x}').$$

In other words, kernel function K_Φ takes two vectors $\mathbf{x}, \mathbf{x}' \in \mathcal{X}$ as input and returns the inner product of Φ-transformed vectors in $\mathcal{Z}$-space. This process is called the *kernel trick*, where the trick part comes from the fact that we do not need to explicitly calculate the vectors in $\mathcal{Z}$-space in order to calculate their inner product. To give few examples of common kernel functions, a first example is the *polynomial kernel function* of Q-degree defined as:

$$K(\mathbf{x}, \mathbf{x}') \equiv (\lambda + \gamma\mathbf{x}^T\mathbf{x}')^Q, \tag{10}$$

where $\lambda, \gamma > 0$. Another common example is the *Gaussian radial basis function kernel* defined as:

$$K(\mathbf{x}, \mathbf{x}') \equiv \exp\left(-\frac{||\mathbf{x} - \mathbf{x}'||^2}{2\sigma^2}\right), \qquad (11)$$

where $\sigma > 0$. It is straightforward to show, that if we set $\sigma = 1$ in equation (11), then we have

$$K(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{1}{2}||\mathbf{x} - \mathbf{x}'||^2\right) = \sum_{k=0}^{\infty} \frac{(\mathbf{x}^T\mathbf{x}')^k}{k!} \exp\left(-\frac{1}{2}||\mathbf{x}||^2\right) \exp\left(-\frac{1}{2}||\mathbf{x}'||^2\right),$$

which represents an inner product in an *infinite dimensional* $\mathcal{Z}$-space (the Φ-function produces an infinite dimensional vector). It turns out also, that one can construct his own kernel functions K, if it is valid that the symmetric matrix:

$$K_M = \begin{bmatrix} K(\mathbf{x}_1, \mathbf{x}_1) & K(\mathbf{x}_1, \mathbf{x}_2) & \cdots & K(\mathbf{x}_1, \mathbf{x}_n) \\ K(\mathbf{x}_2, \mathbf{x}_1) & K(\mathbf{x}_2, \mathbf{x}_2) & \cdots & K(\mathbf{x}_2, \mathbf{x}_n) \\ \vdots & \vdots & \ddots & \vdots \\ K(\mathbf{x}_n, \mathbf{x}_1) & K(\mathbf{x}_n, \mathbf{x}_2) & \cdots & K(\mathbf{x}_n, \mathbf{x}_n) \end{bmatrix}, \qquad (12)$$

is positive semidefinite (i.e. $\mathbf{x}_i^T K_M \mathbf{x}_i \geq 0 \ \forall \ i$) for all vectors $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$. This condition is called *Mercer's condition*. In other words, if your K satisfies Mercer's condition, then you have a valid kernel function.

4. Solving the optimal SVM

We have so far gone through the core ideas of SVM models: the maximum margin classifier, VC-inequality, linearly/non-linearly separable data and the kernel methods. We have observed, that in order to solve the SVM classifier $(\mathbf{w}, b)$ we need to solve the problem in equation (5):

$$\begin{aligned} &\underset{\mathbf{w},b}{\text{minimize:}} && \frac{1}{2}\mathbf{w}^T\mathbf{w} \\ &\text{subject to:} && y_i\left(\mathbf{w}^T\mathbf{x}_i + b\right) \geq 1 \quad (i = 1, 2, \ldots, n). \end{aligned}$$

Lets now go on and solve this problem in a toy example. Let the input data be defined in the following way:

$$X = \begin{bmatrix} 0 & 0 \\ 2 & 2 \\ 2 & 0 \\ 3 & 0 \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} -1 \\ -1 \\ +1 \\ +1 \end{bmatrix},$$

where matrix X represents the container of input vectors of $\mathbf{x}$ and $\mathbf{y}$ represents the class labels correspondingly. Each row in X and $\mathbf{y}$ correspond to a single data point. Formulating the problem of equation (5) in terms of this data we get:

$$\begin{aligned} &\underset{w_1,w_2,b}{\text{minimize:}} && \frac{1}{2}(w_1^2 + w_2^2) \\ &\text{subject to:} && -b \geq 1 \quad \text{(i)} \\ & && -(2w_1 + 2w_2 + b) \geq 1 \quad \text{(ii)} \\ & && 2w_1 + b \geq 1 \quad \text{(iii)} \\ & && 3w_1 + b \geq 1 \quad \text{(iv)}. \end{aligned}$$

By combining inequalities (i), (iii) and (ii), (iii) together we get the conditions $w_1 \geq 1$ and $w_2 \leq -1$. By now squaring both these two new conditions and combining them we get the condition $\frac{1}{2}(w_1^2 + w_2^2) \geq 1$. From this we see that the minimum is achieved when (with conditions satisfied) $w_1 = 1$ ja $w_2 = -1$. Substituting these values of w_1, w_2 into inequalities (ii)-(iv) we get that $b = -1$ and so the SVM model in this case is:

$$(w_1^*, w_2^*, b^*) = (1, -1, -1),$$

and so in terms of equation (6) we get the SVM classifier as:

$$g(\mathbf{x}) = \operatorname{sign}(x_1 - x_2 - 1).$$

Notice that the width of the margin in this case is $\frac{1}{||\mathbf{w}^*||} = \frac{1}{\sqrt{2}} \approx 0.707$. This is illustrated in figure X, with the distance from the line to the support vectors is $\frac{1}{\sqrt{2}}$. It was fairly easy to solve this particular SVM model but in general it is not this easy. The problem might have a lot more parameters and inequalities in which case we need to use more sophisticated methods to solve the SVM model. We will next consider *quadratic programming*, which we use for solving more general SVM models.

4.1 Quadratic programming

In this section we will aim to solve the parameters $(\mathbf{w}, b)$ of the SVM model in equation (6) using the methods of quadratic programming (QP). In a general QP-problem, we aim to solve problem:

$$\begin{aligned} \underset{\mathbf{u}\in\mathbb{R}^l}{\text{minimize:}} \quad & \frac{1}{2}\mathbf{u}^T Q\mathbf{u} + \mathbf{p}^T\mathbf{u} \\ \text{subject to:} \quad & \mathbf{a}_i^T\mathbf{u} \geq c_i \qquad (i = 1, \ldots, m), \end{aligned} \tag{13}$$

where Q is a $l \times l$ symmetric positive semidefinite matrix, $\mathbf{u}, \mathbf{p}, \mathbf{a}_i$ are l-dimensional vectors and c_i is some constant. The optimization problem in (13) contains a qudratic term $\frac{1}{2}\mathbf{u}^T Q\mathbf{u}$ and a linear term $\mathbf{p}^T\mathbf{u}$. The constraints $\mathbf{a}_i^T\mathbf{u} \geq c_i$ are linear. Because Q is positive semidefinite the problem in (13) is a convex optimization problem. If the vectors $\mathbf{a}_i$ and constants c_i are grouped into matrix A and vector $\mathbf{c}$:

$$A = \begin{bmatrix} \mathbf{a}_1^T \\ \vdots \\ \mathbf{a}_m^T \end{bmatrix} \quad \mathbf{c} = \begin{bmatrix} c_1 \\ \vdots \\ c_m \end{bmatrix},$$

then the optimization problem (13) can be presented in the form:

$$\begin{aligned} \underset{\mathbf{u}\in\mathbb{R}^l}{\text{minimize:}} \quad & \frac{1}{2}\mathbf{u}^T Q\mathbf{u} + \mathbf{p}^T\mathbf{u} \\ \text{subject to:} \quad & A\mathbf{u} \geq \mathbf{c}. \end{aligned}$$

The constraints of problem (13) can also be presented in the form $A\mathbf{u} \leq \mathbf{c}$ by multiplying both sides with -1. Next we will show that the SVM optimization problem in (5) can be represented as a QP-problem. In order to do this, we must identify $A, Q, \mathbf{u}, \mathbf{p}$ ja $\mathbf{c}$ from problem (5). This requires finding the optimal values of $(\mathbf{w}, b)$ so we have now:

$$\mathbf{u} = \begin{bmatrix} b \\ \mathbf{w} \end{bmatrix} \in \mathbb{R}^{d+1},$$

so $l = d + 1$. Our task is to minimize the term $\frac{1}{2}\mathbf{w}^T\mathbf{w}$, which must be presented in the form $\frac{1}{2}\mathbf{u}^T Q\mathbf{u} + \mathbf{p}^T\mathbf{u}$. We note that:

$$\mathbf{w}^T\mathbf{w} = \begin{bmatrix} b & \mathbf{w}^T \end{bmatrix} \begin{bmatrix} 0 & \mathbf{0}_d^T \\ \mathbf{0}_d & \mathbf{I}_d \end{bmatrix} \begin{bmatrix} b \\ \mathbf{w} \end{bmatrix},$$

where $\mathbf{I}_d$ is a $d \times d$ identity matrix and $\mathbf{0}_d$ is a d-dimensional zero vector. From this we see that:

$$Q = \begin{bmatrix} 0 & \mathbf{0}_d^T \\ \mathbf{0}_d & \mathbf{I}_d \end{bmatrix} \qquad \mathbf{p} = \mathbf{0}_{d+1},$$

where Q is positive semidefinite. Because there are total of n $y_i \left(\mathbf{w}^T \mathbf{x}_i + b\right) \geq 1$ inequalities we get that $m = n$. Also note that these inequalities are equivalent with the inequalities:

$$\begin{bmatrix} y_i & y_i \mathbf{x}_i^T \end{bmatrix} \mathbf{u} \geq 1,$$

and so by setting $\mathbf{a}_i^T = y_i \begin{bmatrix} 1 & \mathbf{x}_i^T \end{bmatrix}$ and $c_i = 1$ in problem (13) we get:

$$A = \begin{bmatrix} y_1 & y_1 \mathbf{x}_1^T \\ \vdots & \vdots \\ y_n & y_n \mathbf{x}_n^T \end{bmatrix} \qquad \mathbf{c} = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}.$$

Therefore we have shown that problem (5) is indeed a QP-problem of form (13). Next, we will take a look at the Lagrangian duality form for hard margin SVM, which makes solving the problem (13) easier and brings in the kernel methods introduced in section 3.4. We will also go through why the problem (5) includes calculating the inner products $\mathbf{x}_i^T \mathbf{x}_j, \forall i, j \in \{1, \ldots, n\}$ as we promised earlier in section 3.4.

4.2 Lagrangian dual

In section 3, we discussed about applying the nonlinear transformation $\Phi : \mathcal{X} \to \mathcal{Z} \subset \mathbb{R}^q$ to the input vectors and we ended up into the optimization problem (8). In this problem, we have $q + 1$ variables to solve since $\mathbf{u} = [\tilde{\mathbf{w}}, \tilde{b}] \in \mathbb{R}^{q+1}$. It is quite difficult to solve this problem if q is very large or even infinite $(q = \infty)$. By transforming the problem (8) called the *primal* into another form called the *Lagrangian dual form* the SVM problem is transformed into a QP-problem with n variables to be solved with $n + 1$ constraints. The dual problem is independent of the dimensionality of the space $\mathcal{Z}$, and depends only from the amount of data points n. This is a very useful method to apply especially when q is very large. We will next see how to transform the QP-problem in (13) into a dual form. To begin, we start from the primal form of the problem:

$$\begin{aligned} &\underset{\mathbf{u} \in \mathbb{R}^l}{\text{minimize:}} && \frac{1}{2} \mathbf{u}^T Q \mathbf{u} + \mathbf{p}^T \mathbf{u} \\ &\text{subject to:} && \mathbf{a}_i^T \mathbf{u} \geq c_i \qquad (i = 1, \ldots, m). \end{aligned}$$

We proceed next by dropping out the constraints in the above problem and adding a *penalty term* in the following way:

$$\underset{\mathbf{u} \in \mathbb{R}^l}{\text{minimize:}} \qquad \frac{1}{2} \mathbf{u}^T Q \mathbf{u} + \mathbf{p}^T \mathbf{u} + \max_{\boldsymbol{\alpha} \geq \mathbf{0}} \sum_{i=1}^{m} \alpha_i \left(c_i - \mathbf{a}_i^T \mathbf{u}\right), \qquad (14)$$

where $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_m)$. It can be shown (see e.g. literature on nonlinear programming) that the primal problem (13) is equivalent with this new problem (14), as long as there exists at least one solution which satisfies the constraints of problem (13). The penalty term in (14) encourages the optimization to choose vectors $\mathbf{u}$ such that $c_i - \mathbf{a}_i^T \mathbf{u} \leq 0$ (because $\alpha_i \geq 0$), satisfying thus the constraints in (13). Notice that in (14), at the optimum solution $(\mathbf{u}^*, \boldsymbol{\alpha}^*)$ we have either $\alpha_i^* = 0$ or $c_i - \mathbf{a}_i^T \mathbf{u}^* = 0 \; \forall \, i$. Thus, we can instead of problem (13) solve the simpler unconstrained problem (14). The function in (14) is called the *Lagrangian* which is defined as:

$$\mathcal{L}(\mathbf{u}, \boldsymbol{\alpha}) = \frac{1}{2} \mathbf{u}^T Q \mathbf{u} + \mathbf{p}^T \mathbf{u} + \sum_{i=1}^{m} \alpha_i \left(c_i - \mathbf{a}_i^T \mathbf{u}\right), \qquad (15)$$

so our task is to solve the optimization task:

$$\min_{\mathbf{u}\in\mathbb{R}^l} \max_{\boldsymbol{\alpha}\geq\mathbf{0}} \mathcal{L}\left(\mathbf{u}, \boldsymbol{\alpha}\right). \qquad (16)$$

In convex quadratic programming we can take advantage of the so-called *strong duality*:

$$\min_{\mathbf{u}\in\mathbb{R}^l} \max_{\boldsymbol{\alpha}\geq\mathbf{0}} \mathcal{L}\left(\mathbf{u}, \boldsymbol{\alpha}\right) = \max_{\boldsymbol{\alpha}\geq\mathbf{0}} \min_{\mathbf{u}\in\mathbb{R}^l} \mathcal{L}\left(\mathbf{u}, \boldsymbol{\alpha}\right), \qquad (17)$$

which can be shown to be true if the Lagrangian $\mathcal{L}\left(\mathbf{u}, \boldsymbol{\alpha}\right)$ has the form as in (15) and there exists a solution satisying the constraints $\mathbf{a}_i^T\mathbf{u} \geq c_i$. A proof for these facts can be found from literature related to quadratic programming and convex optimization. With the help of the Lagrangian and strong duality we have transformed the original problem (13) into an easier unconstrained optimization problem:

$$\max_{\boldsymbol{\alpha}\geq\mathbf{0}} \min_{\mathbf{u}\in\mathbb{R}^l} \mathcal{L}\left(\mathbf{u}, \boldsymbol{\alpha}\right),$$

which is called the *Lagrangian dual problem*. To wrap up this section, we will state the necessary conditions a solution of the QP-problem in (13) must satisfy.

Necessary optimality conditions for the QP-problem (13)**.** If the primal convex QP-problem in (13):

$$\begin{aligned} \underset{\mathbf{u}\in\mathbb{R}^l}{\text{minimize:}} \quad & \frac{1}{2}\mathbf{u}^T Q\mathbf{u} + \mathbf{p}^T\mathbf{u} \\ \text{subject to:} \quad & \mathbf{a}_i^T\mathbf{u} \geq c_i \qquad (i = 1, \ldots, m), \end{aligned}$$

has the corresponding Lagrangian function in (15):

$$\mathcal{L}\left(\mathbf{u}, \boldsymbol{\alpha}\right) = \frac{1}{2}\mathbf{u}^T Q\mathbf{u} + \mathbf{p}^T\mathbf{u} + \sum_{i=1}^{m} \alpha_i \left(c_i - \mathbf{a}_i^T\mathbf{u}\right),$$

then the solution $\mathbf{u}^*$ is optimal for the primal problem (13), if and only if $(\mathbf{u}^*, \boldsymbol{\alpha}^*)$ is a solution for the dual problem in (14):

$$\max_{\boldsymbol{\alpha}\geq\mathbf{0}} \min_{\mathbf{u}\in\mathbb{R}^l} \mathcal{L}\left(\mathbf{u}, \boldsymbol{\alpha}\right).$$

The optimal solution $(\mathbf{u}^*, \boldsymbol{\alpha}^*)$ of problem (14) must satisfy the following conditions:

$$\text{(i)} \quad \mathbf{a}_i^T\mathbf{u}^* \geq c_i \quad \text{and} \quad \alpha_i^* \geq 0 \quad \forall\, i.$$

$$\text{(ii)} \quad \alpha_i^*\left(\mathbf{a}_i^T\mathbf{u}^* - c_i\right) = 0 \quad \forall\, i.$$

$$\text{(iii)} \quad \nabla_{\mathbf{u}}\mathcal{L}\left(\mathbf{u}, \boldsymbol{\alpha}\right)|_{\mathbf{u}=\mathbf{u}^*,\ \boldsymbol{\alpha}=\boldsymbol{\alpha}^*} = \mathbf{0},$$

where $\nabla_{\mathbf{u}}$ denotes the gradient with respect to $\mathbf{u}$. The conditions (i)-(iii) are called the Karush-Kühn-Tucker (KKT)-conditions. The KKT-conditions give us an analytical way to either check or solve the optimal solution of the QP-problem (13). At the last sections of this tutorial, we will produce an iterative algorithm for solving the SVM model and we therefore do not directly apply the KKT-conditions for solving the optimal SVM model. We can however use the KKT-conditions to check whether our iteratively solved solution is indeed optimal or not.

4.3 The dual of hard margin SVM

Lets now next get back to solving the SVM model. In section 4.1, we identified the factors $A, Q, \mathbf{u}, \mathbf{p}, \mathbf{c}$ in order to incorporate the SVM problem (5) into the form in (13). We will now proceed to find the Lagrangian dual function $\mathcal{L}\left(\mathbf{u}, \boldsymbol{\alpha}\right)$ of the SVM problem. To recall, our original problem was:

$$\begin{aligned} &\underset{b,\mathbf{w}}{\text{minimize:}} && \frac{1}{2}\mathbf{w}^T\mathbf{w} \\ &\text{subject to:} && y_i\left(\mathbf{w}^T\mathbf{x}_i + b\right) \geq 1 \qquad (i = 1, \ldots, n). \end{aligned}$$

The optimal solution we aim to find for this problem is the vector $\mathbf{u} = [b, \mathbf{w}]^T$. Lets go on and solve the Lagrangian function. Because there are n contraints, we will get n penalty terms into the Lagrangian with coefficients α_i. The Lagrangian function is therefore:

$$\begin{aligned} \mathcal{L}(b, \mathbf{w}, \boldsymbol{\alpha}) &= \frac{1}{2}\mathbf{w}^T\mathbf{w} + \sum_{i=1}^{n} \alpha_i \left(1 - y_i\left(\mathbf{w}^T\mathbf{x}_i + b\right)\right) \\ &= \frac{1}{2}\mathbf{w}^T\mathbf{w} - \sum_{i=1}^{n} \alpha_i y_i \mathbf{w}^T\mathbf{x}_i - b\sum_{i=1}^{n} \alpha_i y_i + \sum_{i=1}^{n} \alpha_i. \qquad (18) \end{aligned}$$

Our first step is to minimize the Lagrangian function with respect to the vector $\mathbf{u}$ and then maximize with respect to $\boldsymbol{\alpha} \geq \mathbf{0}$. So lets take the derivative of $\mathcal{L}$ with respect to b and $\mathbf{w}$. We get:

$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_{i=1}^{n} \alpha_i y_i \qquad \text{and} \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \mathbf{w} - \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i,$$

and setting these to zero we get:

$$\sum_{i=1}^{n} \alpha_i y_i = 0 \qquad (19)$$

$$\mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i. \qquad (20)$$

By plugging these into the Lagrangian function $\mathcal{L}(b, \mathbf{w}, \boldsymbol{\alpha})$ above we get:

$$\begin{aligned} \mathcal{L}(b, \mathbf{w}, \boldsymbol{\alpha}) &= \frac{1}{2}\mathbf{w}^T\mathbf{w} - \sum_{i=1}^{n} \alpha_i y_i \mathbf{w}^T\mathbf{x}_i - b\sum_{i=1}^{n} \alpha_i y_i + \sum_{i=1}^{n} \alpha_i \\ &= \frac{1}{2}\sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i^T \sum_{j=1}^{n} \alpha_j y_j \mathbf{x}_j - \sum_{i=1}^{n} \alpha_i y_i \sum_{j=1}^{n} \alpha_j y_j \mathbf{x}_j^T \mathbf{x}_i \\ &\quad - b\sum_{i=1}^{n} \alpha_i y_i + \sum_{i=1}^{n} \alpha_i \\ &= \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} y_i y_j \alpha_i \alpha_j \mathbf{x}_i^T \mathbf{x}_j - \sum_{i=1}^{n}\sum_{j=1}^{n} y_i y_j \alpha_i \alpha_j \mathbf{x}_i^T \mathbf{x}_j + \sum_{i=1}^{n} \alpha_i \\ &= -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} y_i y_j \alpha_i \alpha_j \mathbf{x}_i^T \mathbf{x}_j + \sum_{i=1}^{n} \alpha_i, \end{aligned}$$

and so we get that:

$$\mathcal{L}(\boldsymbol{\alpha}) = -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} y_i y_j \alpha_i \alpha_j \mathbf{x}_i^T \mathbf{x}_j + \sum_{i=1}^{n} \alpha_i,$$

which is a function of only the vector $\boldsymbol{\alpha} \geq \mathbf{0}$. Thus, in order to solve the SVM model we need to maximize the function $\mathcal{L}(\boldsymbol{\alpha})$ so that $\boldsymbol{\alpha} \geq \mathbf{0}$. Note that we have now new constraints for the variables α_i by the equation (19), from which we get a total of $n + 1$ constraints. Instead of maximizing the function $\mathcal{L}(\boldsymbol{\alpha})$ we can also

equivalently minimize the function $-\mathcal{L}(\boldsymbol{\alpha})$, and so we have transformed the original optimization problem (5) into the problem:

$$\begin{aligned} \underset{\boldsymbol{\alpha}\in\mathbb{R}^n}{\text{minimize:}} \quad & \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} y_i y_j \alpha_i \alpha_j \mathbf{x}_i^T \mathbf{x}_j - \sum_{i=1}^{n}\alpha_i \qquad (21) \\ \text{subject to:} \quad & \sum_{i=1}^{n}\alpha_i y_i = 0 \\ & \alpha_i \geq 0 \qquad (i = 1, \ldots, n). \end{aligned}$$

From problem (21) we see why the calculation of inner products $\mathbf{x}_i^T \mathbf{x}_j, \forall\, i, j \in \{1, \ldots, n\}$ is important as we discussed before in section 3.4. The problem (21) can also be rewritten equivalently as the convex QP-problem:

$$\begin{aligned} \underset{\boldsymbol{\alpha}\in\mathbb{R}^n}{\text{minimize:}} \quad & \frac{1}{2}\boldsymbol{\alpha}^T Q_D \boldsymbol{\alpha} - \mathbf{1}_n^T \boldsymbol{\alpha} \\ \text{subject to:} \quad & A_D \boldsymbol{\alpha} \geq \mathbf{0}_{n+2}, \qquad (22) \end{aligned}$$

where

$$Q_D = \begin{bmatrix} y_1 y_1 \mathbf{x}_1^T \mathbf{x}_1 & \cdots & y_1 y_n \mathbf{x}_1^T \mathbf{x}_n \\ y_2 y_1 \mathbf{x}_2^T \mathbf{x}_1 & \cdots & y_2 y_n \mathbf{x}_2^T \mathbf{x}_n \\ \vdots & \vdots & \vdots \\ y_n y_1 \mathbf{x}_n^T \mathbf{x}_1 & \cdots & y_n y_n \mathbf{x}_n^T \mathbf{x}_n \end{bmatrix} \quad A_D = \begin{bmatrix} \mathbf{y}^T \\ -\mathbf{y}^T \\ \mathbf{I}_{n\times n} \end{bmatrix}$$

and $\mathbf{I}_{n\times n}, \mathbf{y}^T, \mathbf{0}_{n+2}, \mathbf{1}_n$ stand for $n \times n$ identity matrix, row vector of labels $\{y_1, y_2, \ldots, y_n\}$, $(n+2)$-dimensional zero vector and n-dimensional vector of ones respectively. It can be shown, that if Q_D is positive semidefinite, then the problem (21) is a convex optimization problem (again, to verify check the literature). When the optimal solution $\boldsymbol{\alpha}^*$ for this problem has been solved, we get the parameters $(\mathbf{w}^*, b^*)$ of the optimal SVM hyperplane as (see equations 2 and 20):

$$\mathbf{w}^* = \sum_{i=1}^{n} y_i \alpha_i^* \mathbf{x}_i \qquad (23)$$

$$\begin{aligned} b^* &= \frac{1}{y_s} - \mathbf{w}^{*T}\mathbf{x}_s \\ &= \frac{1}{y_s} - \sum_{i=1}^{n} y_i \alpha_i^* \mathbf{x}_i^T \mathbf{x}_s, \qquad (24) \end{aligned}$$

where $\mathbf{x}_s$ is any support vector satisfying $\alpha_s^* > 0$. For all non-support vectors it holds that $\alpha_i^* = 0$. Because of the constraint in equation (2), it holds for support vectors $\mathbf{x}_s$ that:

$$y_s\left(\mathbf{w}^{*T}\mathbf{x}_s + b^*\right) = 1.$$

The optimal SVM hyperplane model $(\mathbf{w}^*, b^*)$ is therefore:

$$\begin{aligned} g(\mathbf{x}) &= \text{sign}\left(\mathbf{w}^{*T}\mathbf{x} + b^*\right) \\ &= \text{sign}\left(\sum_{i=1}^{n} y_i \alpha_i^* \mathbf{x}_i^T \mathbf{x} + b^*\right) \\ &= \text{sign}\left(\sum_{i=1}^{n} y_i \alpha_i^* \mathbf{x}_i^T \left(\mathbf{x} - \mathbf{x}_s\right) + y_s\right). \end{aligned} \tag{25}$$

Also, because only the support vectors $\mathbf{x}_s$ affect the selection of the SVM hyperplane $(\alpha_s^* > 0)$ we can write the model in (25) in terms of the support vectors as:

$$g(\mathbf{x}) = \text{sign}\left(\sum_{\alpha_i^* > 0} y_i \alpha_i^* \mathbf{x}_i^T \mathbf{x} + b^*\right). \tag{26}$$

4.4 Example: solving the SVM classifier via the Lagrangian dual form

Let the observed data in this example be the same as earlier, that is:

$$X = \begin{bmatrix} 0 & 0 \\ 2 & 2 \\ 2 & 0 \\ 3 & 0 \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} -1 \\ -1 \\ +1 \\ +1 \end{bmatrix}.$$

Here we have that $n = 4$. The matrices Q_D and A_D are identified as:

$$Q_D = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 8 & -4 & -6 \\ 0 & -4 & 4 & 6 \\ 0 & -6 & 6 & 9 \end{bmatrix} \quad A_D = \begin{bmatrix} -1 & -1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

By taking advantage of (22) we get our optimization problem as minimizing the Lagrangian function:

$$\mathcal{L}(\boldsymbol{\alpha}) = 4\alpha_2^2 + 2\alpha_3^2 + \frac{9}{2}\alpha_4^2 - 4\alpha_2\alpha_3 - 6\alpha_2\alpha_4 + 6\alpha_3\alpha_4 - \alpha_1 - \alpha_2 - \alpha_3 - \alpha_4$$

$$\begin{aligned} \text{subject to:} \quad & \alpha_1 + \alpha_2 = \alpha_3 + \alpha_4; \\ & \alpha_1, \alpha_2, \alpha_3, \alpha_4 \geq 0. \end{aligned}$$

By plugging the first constraint equation into the Lagrangian function we get:

$$\mathcal{L}(\boldsymbol{\alpha}) = 4\alpha_2^2 + 2\alpha_3^2 + \frac{9}{2}\alpha_4^2 - 4\alpha_2\alpha_3 - 6\alpha_2\alpha_4 + 6\alpha_3\alpha_4 - 2\alpha_3 - 2\alpha_4.$$

Now we calculate the partial derivative with respect to parameter α_2 and setting this to zero we get:

$$\frac{\partial \mathcal{L}}{\partial \alpha_2} = 8\alpha_2 - 4\alpha_3 - 6\alpha_4 = 0 \quad \leftrightarrow \quad \alpha_2 = \frac{1}{2}\alpha_3 + \frac{3}{4}\alpha_4.$$

It then follows that:

$$\alpha_1 = \alpha_3 + \alpha_4 - \alpha_2 = \frac{1}{2}\alpha_3 + \frac{1}{4}\alpha_4.$$

Note also that $\alpha_1, \alpha_2 \geq 0$. By plugging α_1 and α_2 into the Lagrangian we get:

$$\mathcal{L}(\boldsymbol{\alpha}) = \alpha_3^2 + \frac{9}{4}\alpha_4^2 + 3\alpha_3\alpha_4 - 2\alpha_3 - 2\alpha_4.$$

Next we note that:

$$\mathcal{L}(\boldsymbol{\alpha}) = \frac{1}{4}(2\alpha_3 + 3\alpha_4 - 2)^2 + \alpha_4 - 1,$$

and so taking into account the constraints $\alpha_3, \alpha_4 \geq 0$ we see that the minimum is achieved at the point $\alpha_3 = 1, \alpha_4 = 0$ and so:

$$\alpha_1 = \frac{1}{2} \quad \text{ja} \quad \alpha_2 = \frac{1}{2},$$

so the solution is:

$$\boldsymbol{\alpha}^* = \begin{bmatrix} 1/2 \\ 1/2 \\ 1 \\ 0 \end{bmatrix}.$$

Now by using equations (23) and (24) we get:

$$\mathbf{w}^* = -\frac{1}{2}\mathbf{x}_1 - \frac{1}{2}\mathbf{x}_2 + \mathbf{x}_3 = (0,0) - (1,1) + (2,0) = (1,-1),$$

$$b^* = -1 + \mathbf{w}^{*T}\mathbf{x}_1 = -1 + 0 = -1,$$

where $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$ are support vectors. According to equation (25) the optimal SVM classifier is therefore:

$$g(\mathbf{x}) = \operatorname{sign}(x_1 - x_2 - 1),$$

just like we obtained in the earlier example without using the Lagrangian dual form.

4.5 The dual in the $\mathcal{Z}$-space

So what part do the kernel methods play in the SVM model? If we incorporate them into the SVM model does the developed theory change much? The answer is no, it is actually very easy to add the kernel methods into the developed SVM models, namely into the problem in (21). We simply replace the inner products $\mathbf{x}_i^T\mathbf{x}_j$ with the inner products of space $\mathcal{Z}$, that is:

$$\mathbf{z}_i^T\mathbf{z}_j = \Phi(\mathbf{x}_i)^T\Phi\left(\mathbf{x}_j\right) = K\left(\mathbf{x}_i, \mathbf{x}_j\right),$$

and so the problem (21) becomes:

$$\begin{aligned} \underset{\boldsymbol{\alpha}\in\mathbb{R}^n}{\text{minimize:}} \quad & \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} y_i y_j \alpha_i \alpha_j K\left(\mathbf{x}_i, \mathbf{x}_j\right) - \sum_{i=1}^{n}\alpha_i \\ \text{subject to:} \quad & \sum_{i=1}^{n}\alpha_i y_i = 0 \\ & \alpha_i \geq 0 \qquad (i = 1, \ldots, n). \end{aligned} \tag{27}$$

The matrix Q_D in problem (22) becomes the so-called *kernel matrix*:

$$Q_D = \begin{bmatrix} y_1 y_1 K(\mathbf{x}_1, \mathbf{x}_1) & \cdots & y_1 y_n K(\mathbf{x}_1, \mathbf{x}_n) \\ y_2 y_1 K(\mathbf{x}_2, \mathbf{x}_1) & \cdots & y_2 y_n K(\mathbf{x}_2, \mathbf{x}_n) \\ \vdots & \vdots & \vdots \\ y_n y_1 K(\mathbf{x}_n, \mathbf{x}_1) & \cdots & y_n y_n K(\mathbf{x}_n, \mathbf{x}_n) \end{bmatrix},$$

and so, together with the nonlinear transformation Φ the SVM model of (26) becomes:

$$\boxed{g(\mathbf{x}) = \text{sign}\left(\sum_{\alpha_i^* > 0} y_i \alpha_i^* K(\mathbf{x}_i, \mathbf{x}) + b^* \right), \qquad (28)}$$

where now $b^* = y_s - \sum_{\alpha_i^* > 0} y_i \alpha_i^* K(\mathbf{x}_i, \mathbf{x}_s)$ and $\mathbf{x}_s$ is any support vector with $\alpha_s^* > 0$ (and corresponding y_s).

Phew, we have now gone through everything we need to solve the SVM model of problem (5). Now it's time to get hands dirty with the most interesting part of this tutorial, that is actually solving the model $(\mathbf{w}^*, b^*)$. We will also see (but not go through the proof), that with only slight modifications to the SVM problem we can solve either soft or hard margin SVM models. In fact, hard margin SVM is actually a special case of the soft margin SVM model.

5. Simple algorithm for solving the SVM model

Recall from previous sections, that the (hard margin) SVM model was solved by finding the optimal solution to the problem (22):

$$\begin{aligned} \underset{\boldsymbol{\alpha} \in \mathbb{R}^n}{\text{minimize:}} \quad & \frac{1}{2} \boldsymbol{\alpha}^T Q_D \boldsymbol{\alpha} - \mathbf{1}_n^T \boldsymbol{\alpha} \\ \text{subject to:} \quad & A_D \boldsymbol{\alpha} \geq \mathbf{0}_{n+2}, \end{aligned}$$

where

$$Q_D = \begin{bmatrix} y_1 y_1 K(\mathbf{x}_1, \mathbf{x}_1) & \cdots & y_1 y_n K(\mathbf{x}_1, \mathbf{x}_n) \\ y_2 y_1 K(\mathbf{x}_2, \mathbf{x}_1) & \cdots & y_2 y_n K(\mathbf{x}_2, \mathbf{x}_n) \\ \vdots & \vdots & \vdots \\ y_n y_1 K(\mathbf{x}_n, \mathbf{x}_1) & \cdots & y_n y_n K(\mathbf{x}_n, \mathbf{x}_n) \end{bmatrix} \quad A_D = \begin{bmatrix} \mathbf{y}^T \\ -\mathbf{y}^T \\ \mathbf{I}_{n \times n} \end{bmatrix}.$$

Note that above we have incorporated the transformation Φ into the game in order to make the problem as general as possible. In the case of a *linear kernel* (as in (22)), we simply have $K(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^T \mathbf{x}_j$. In order to also incorporate the soft margin SVM into the above problem, we modify it slightly by adding a *penalty rate* $C \in \mathbb{R}^+$ in the following way:

$$\boxed{\begin{aligned} \underset{\boldsymbol{\alpha} \in \mathbb{R}^n}{\text{minimize:}} \quad & \frac{1}{2} \boldsymbol{\alpha}^T Q_D \boldsymbol{\alpha} - \mathbf{1}_n^T \boldsymbol{\alpha} \\ \text{subject to:} \quad & \mathbf{y}^T \boldsymbol{\alpha} = 0 \qquad (29) \\ & \mathbf{0} \leq \boldsymbol{\alpha} \leq C \cdot \mathbf{1}. \end{aligned}}$$

It is interesting to see that the above most general SVM problem so far (general Φ, soft margin) in (29) differs only a little bit in the contraint part when compared to (22). To skip unnecessary complications, we leave the proofs of this fact for the reader to find out from related literature. The penalty term C controls the amount of strictness we put on perfect classification of the SVM model. The higher the value of C is, the 'harder' our model is. In fact, when we have $C = \infty$ then the problem in (29) is equal to the hard margin problem in (22).

To mention lastly, it is a bit more complicated to solve the parameter b^* in the case of a soft margin SVM, but I will get into that later. Next, we will have a short review on the techniques of nonlinear programming that our algorithm will be based upon.

5.1 Active set methods

The active set method, is a simple technique where the goal is to keep track of the so-called *active contraints* and proceed with the problem optimization in the subspace defined by these constraints. The contraints in our SVM problem in (29) form a closed convex subspace where we are to apply our optimization. We can think of the optimization problem in (29) as moving along a convex function with borders set up by a hyperplane defined by the vector $\boldsymbol{\alpha}$. In the active set method, we therefore proceed freely with 'moving' in the convex function until some constraints become active, at which point we change direction so that feasibility is preserved (that is we move only in the allowed subspace). In the SVM problem, our constraints are of the form $A\mathbf{x} \leq \mathbf{y}$, which are active at point $\mathbf{x}$ if $A\mathbf{x} = \mathbf{y}$ and inactive if $A\mathbf{x} < \mathbf{y}$. The inactive constraints are not relevant for the optimization and so we always need to keep track of only the active constraints. We have now set up the method that keeps track that all the contraints of our problem (30) are satisfied. Next, we will discuss about how we 'move' in the space of $\boldsymbol{\alpha}$s towards the optimum.

5.2 Gradient projection method of Rosen

If you are familiar with basic concepts of mathematical optimization, then you should have heard at some point about *gradient descent* based algorithms. Gradient descent algorithms work simply by firstly calculating the gradient $\mathbf{g}(\mathbf{x})$ of the (convex) function to $f(\mathbf{x})$ to be minimized (i.e. $\mathbf{g}(\mathbf{x}) = \nabla f(\mathbf{x})$), and secondly updating the current best solution $\mathbf{x}_i$ with the rule $\mathbf{x}_{i+1} = \mathbf{x}_i - \lambda \mathbf{g}(\mathbf{x}_i)$, where $\lambda \geq 0$ is sometimes called the *learning rate* which determines how much to move into the direction $-\mathbf{g}(\mathbf{x}_i)$. That is, we are moving into the direction of the negative gradient of the function $f(\mathbf{x})$, which is the direction of deepest decrement of the function value. The just described update rule is iteratively applied until convergence to optimum is achieved. Gradient descent is a standard technique of mathematical optimization and it is used in tons of applications and research fields. Many variates of this methods also exist, such as the *conjugate gradient method* or the *quasi-Newton methods*.

One the problems in gradient descent however is that it assumes the optimization process to be *unconstrained*, that is we are free to move in the parameter space. As we have seen above, this is not the case in the SVM problem where we have linear constraints on the $\boldsymbol{\alpha}$-vector. We are thus restricted to conduct the optimization problem in a restricted subspace as we discussed with active set methods above. Fortunately, many techniques exists which allow us to conduct gradient descent while maintaining the *feasibility* of the new solutions $\mathbf{x}_{i+1}$. A solution $\mathbf{x}$ is called *feasible*, if it is a solution which satisfies the constraints of the optimization problem. One of these techniques that maintain feasibility, is called the *gradient projection method of Rosen* (GP). GP does exactly what its name suggests, firstly, it project the gradient vector into a feasible subspace, and then an update step is conducted in this restricted space. In other words, we are simply doing gradient descent, but at every step we project the gradient vector into a subspace which preserves feasibility. We will next make this more explicit. Consider the general problem:

$$\begin{aligned} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize:}} \quad & f(\mathbf{x}) \\ \text{subject to:} \quad & A\mathbf{x} \leq \mathbf{b} \\ & Q\mathbf{x} = \mathbf{q}, \end{aligned} \tag{30}$$

where A is an $m \times n$ matrix, Q is an $l \times n$ matrix ($l \geq n$), b is an m-dimensional vector, $\mathbf{q}$ is an l-dimensional vector, and $f : \mathbb{R}^n \to \mathbb{R}$ is a differentiable function. It can be proven (e.g. nonlinear programming by Bazaraa) that the following lemma is true:

Lemma: Let $\mathbf{x}$ be a feasible point such that $A_1\mathbf{x} = \mathbf{b}_1$ and $A_2\mathbf{x} < \mathbf{b}_2$, where $A^T = (A_1^T, A_2^T)$ and $\mathbf{b}^T = (\mathbf{b}_1^T, \mathbf{b}_2^T)$. Furthermore, suppose that f is differentiable at $\mathbf{x}$ and that $P = \mathbf{I} - M^T\left(MM^T\right)^{-1}M$, where $M^T = (A_1^T, Q^T)$ is a full rank matrix. It then follows, that the vector $\mathbf{d} = -P\nabla f(\mathbf{x}) = -P\mathbf{g}(\mathbf{x})$ is an improving feasible direction of f at $\mathbf{x}$.

The matrix P is called a *projection matrix* and it is symmetric and positive semidefinite. So, we have all the required components at our disposal. We have formulated the SVM problem in (29), and we have defined the optimization techniques we need for solving this problem, namely GP combined with active set method. Next, we will formulate the pseudocode of the algorithm for solving the problem (29).

5.3 Constructing a gradient projection/active set-based SVM solver

Before constructing our gradient projection/active set-based algorithm (let us call it GPAS), we need to identify the key components of the above lemma in problem (29). First of all, lets rewrite problem (29) to make it resemble more the problem (30) as:

$$\begin{aligned} \underset{\boldsymbol{\alpha}\in\mathbb{R}^n}{\text{minimize:}} \quad & \frac{1}{2}\boldsymbol{\alpha}^T Q_D \boldsymbol{\alpha} - \mathbf{1}_n^T\boldsymbol{\alpha} \\ \text{subject to:} \quad & A_C\boldsymbol{\alpha} \leq \mathbf{b} \\ & \mathbf{y}^T\boldsymbol{\alpha} = 0 \end{aligned} \qquad (31)$$

where Q_D is the kernel matrix (see e.g. (27)) and

$$A_C = \begin{bmatrix} -\mathbf{I}_{n\times n} \\ \mathbf{I}_{n\times n} \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} \mathbf{0}_n \\ \mathbf{C}_n \end{bmatrix},$$

where $\mathbf{C}_n$ is a n-dimensional vector of values C. The problem (31) is equivalent to problem (29) and we have written it in the style of problem (30) identifying the corresponding factors. We now proceed with constructing the matrix M in the lemma of section 5.2 for problem (31). Let the set J be a set of indexes for which $\alpha_i = 0$ or $\alpha_i = C$ at point $\boldsymbol{\alpha}$, that is $J = \{j \in (1, 2, \ldots, n) \mid \alpha_j = 0 \text{ or } \alpha_j = C, \boldsymbol{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_n)\}$. For example, if $C = 1, n = 3$ and $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \alpha_3) = (0, 0.5, 1)$, then $J = \{1, 3\}$. Next, we define the matrix A_1 (as in the lemma) to be a $|J| \times n$ matrix with zeros everywhere else, with the following exceptions: Let the ith value in J be denoted as k. For all $i \in (1, 2, \ldots, |J|)$, we set $A_1(i, k) = 1$. So for the example $J = \{1, 3\}$ we have:

$$A_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Next, we easily note that $Q = \mathbf{y}^T$ and so we get that M is:

$$M = \begin{bmatrix} A_1 \\ \mathbf{y}^T \end{bmatrix}, \quad M^T = \begin{bmatrix} A_1^T & \mathbf{y} \end{bmatrix}.$$

For the same example above we have:

$$M = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ y_1 & y_2 & y_3 \end{bmatrix}, \quad M^T = \begin{bmatrix} 1 & 0 & y_1 \\ 0 & 0 & y_2 \\ 0 & 1 & y_3 \end{bmatrix}.$$

Now we simply apply $P = \mathbf{I} - M^T\left(MM^T\right)^{-1}M$ in the lemma and we have our projection matrix P, which projects any vector to the subspace specified by the set J and equation $\mathbf{y}^T\boldsymbol{\alpha} = 0$. The constraints $\mathbf{0} \leq \boldsymbol{\alpha} \leq C \cdot \mathbf{1}$ are called *box constraints*, and so we can think (in the GPAS) as moving along a hyperplane

$\mathbf{y}^T \boldsymbol{\alpha} = 0$ constrained inside a box $\mathbf{0} \leq \boldsymbol{\alpha} \leq C \cdot \mathbf{1}$.

We are now almost done! We have found the 'active' constraints and constructed the projection matrix P. So, given that our current best solution is $\boldsymbol{\alpha}_i$, our next best guess is therefore:

$$\boldsymbol{\alpha}_{i+1} = \boldsymbol{\alpha}_i - \lambda_d P\mathbf{g}(\boldsymbol{\alpha}_i),$$

where the $\lambda_d \geq 0$ (step size) will be determined by *line search*, which simply means solving λ_d from:

$$\frac{\partial f(\boldsymbol{\alpha}_i - \lambda P\mathbf{g}(\boldsymbol{\alpha}_i))}{\partial \lambda} = 0.$$

In the SVM problem (31), we can think of our objective function at this point only a function of λ (since $\boldsymbol{\alpha}_i$, P and $\mathbf{g}(\boldsymbol{\alpha}_i)$ are fixed), that is:

$$f(\lambda) = \frac{1}{2}(\boldsymbol{\alpha}_i + \lambda \mathbf{d}_i)^T Q_D(\boldsymbol{\alpha}_i + \lambda \mathbf{d}_i) - \mathbf{1}_n^T(\boldsymbol{\alpha}_i + \lambda \mathbf{d}_i),$$

where $\mathbf{d}_i = -P\mathbf{g}(\boldsymbol{\alpha}_i) = -P\mathbf{g}_i$. Lets now calculate the derivative of $f(\lambda)$, equate it to zero, and solve for λ_d:

$$\begin{aligned}
f(\lambda) &= \frac{1}{2}(\boldsymbol{\alpha}_i + \lambda \mathbf{d}_i)^T Q_D(\boldsymbol{\alpha}_i + \lambda \mathbf{d}_i) - \mathbf{1}_n^T(\boldsymbol{\alpha}_i + \lambda \mathbf{d}_i) \\
&= \frac{1}{2}(\boldsymbol{\alpha}_i + \lambda \mathbf{d}_i)^T \begin{bmatrix} \mathbf{q}_1^T \\ \vdots \\ \mathbf{q}_n^T \end{bmatrix} (\boldsymbol{\alpha}_i + \lambda \mathbf{d}_i) - \mathbf{1}_n^T(\boldsymbol{\alpha}_i + \lambda \mathbf{d}_i) \\
&= \frac{1}{2}(\boldsymbol{\alpha}_i + \lambda \mathbf{d}_i)^T \begin{bmatrix} \mathbf{q}_1^T \boldsymbol{\alpha}_i + \lambda \mathbf{q}_1^T \mathbf{d}_i \\ \vdots \\ \mathbf{q}_n^T \boldsymbol{\alpha}_i + \lambda \mathbf{q}_n^T \mathbf{d}_i \end{bmatrix} - \mathbf{1}_n^T(\boldsymbol{\alpha}_i + \lambda \mathbf{d}_i) \\
&= \frac{1}{2}\left[\sum_{i=1}^n \alpha_i(\mathbf{q}_i^T \boldsymbol{\alpha}_i + \lambda \mathbf{q}_i^T \mathbf{d}_i) + \lambda d_i(\mathbf{q}_i^T \boldsymbol{\alpha}_i + \lambda \mathbf{q}_i^T \mathbf{d}_i)\right] - \sum_{i=1}^n \alpha_i - \lambda \sum_{i=1}^n d_i \\
&= \frac{1}{2}\left[\sum_{i=1}^n \alpha_i \mathbf{q}_i^T \boldsymbol{\alpha}_i + \alpha_i \lambda \mathbf{q}_i^T \mathbf{d}_i + \lambda d_i \mathbf{q}_i^T \boldsymbol{\alpha}_i + \lambda^2 d_i \mathbf{q}_i^T \mathbf{d}_i\right] - \sum_{i=1}^n \alpha_i - \lambda \sum_{i=1}^n d_i.
\end{aligned}$$

And now we calculate the derivative:

$$\begin{aligned}
\frac{\partial f(\lambda)}{\partial \lambda} &= \frac{1}{2}\left[\sum_{i=1}^n \alpha_i \mathbf{q}_i^T \mathbf{d}_i + d_i \mathbf{q}_i^T \boldsymbol{\alpha}_i + 2\lambda d_i \mathbf{q}_i^T \mathbf{d}_i\right] - \sum_{i=1}^n d_i \\
&= \frac{1}{2}\sum_{i=1}^n (\alpha_i \mathbf{q}_i^T \mathbf{d}_i + d_i \mathbf{q}_i^T \boldsymbol{\alpha}_i) + \lambda \sum_{i=1}^n d_i \mathbf{q}_i^T \mathbf{d}_i - \sum_{i=1}^n d_i \\
&= \frac{1}{2}\boldsymbol{\alpha}_i^T Q_D \mathbf{d}_i + \frac{1}{2}\mathbf{d}_i^T Q_D \boldsymbol{\alpha}_i + \lambda \mathbf{d}_i^T Q_D \mathbf{d}_i - \sum_{i=1}^n d_i \\
&= \boldsymbol{\alpha}_i^T Q_D \mathbf{d}_i + \lambda \mathbf{d}_i^T Q_D \mathbf{d}_i - \mathbf{1}_n^T \mathbf{d}_i \\
&= \lambda \mathbf{d}_i^T Q_D \mathbf{d}_i + (\boldsymbol{\alpha}_i^T Q_D - \mathbf{1}_n^T)\mathbf{d}_i.
\end{aligned}$$

Before proceesing with $\frac{\partial f(\lambda)}{\partial \lambda}$, let us calculate the gradient $\mathbf{g}(\boldsymbol{\alpha}) = \nabla f(\boldsymbol{\alpha})$ (we need it soon). We have:

$$f(\boldsymbol{\alpha}) = \frac{1}{2}\boldsymbol{\alpha}^T Q_D \boldsymbol{\alpha} - \mathbf{1}_n^T \boldsymbol{\alpha}$$

$$= \frac{1}{2}\boldsymbol{\alpha}^T \begin{bmatrix} \mathbf{q}_1^T \\ \vdots \\ \mathbf{q}_n^T \end{bmatrix} \boldsymbol{\alpha} - \mathbf{1}_n^T \boldsymbol{\alpha}$$

$$= \frac{1}{2}\sum_{i=1}^{n} \alpha_i \mathbf{q}_i^T \boldsymbol{\alpha} - \sum_{i=1}^{n} \alpha_i.$$

$$\frac{\partial f}{\partial \alpha_k} = \frac{1}{2}\left[\sum_{j\neq k} \alpha_j q_{jk} + \sum_{j\neq k} q_{kj}\alpha_j + 2\alpha_k q_{kk}\right] - 1 = \sum_{i=1}^{n} q_{ki}\alpha_i - 1 = \mathbf{q}_k^T \boldsymbol{\alpha} - 1,$$

where $q_{ij} = y_i y_j K(\mathbf{x}_i, \mathbf{x}_j)$ and $q_{ij} = q_{ji}$ due to the symmetricity of Q_D. We thus have that:

$$\mathbf{g}(\boldsymbol{\alpha}) = \nabla f(\boldsymbol{\alpha}) = \begin{bmatrix} \frac{\partial f}{\partial \alpha_1} \\ \vdots \\ \frac{\partial f}{\partial \alpha_n} \end{bmatrix} = \begin{bmatrix} \mathbf{q}_1^T \boldsymbol{\alpha} - 1 \\ \vdots \\ \mathbf{q}_n^T \boldsymbol{\alpha} - 1 \end{bmatrix} = Q_D \boldsymbol{\alpha} - \mathbf{1}_n.$$

Now continue with $\frac{\partial f(\lambda)}{\partial \lambda} = 0$:

$$\lambda \mathbf{d}_i^T Q_D \mathbf{d}_i + (Q_D \boldsymbol{\alpha}_i - \mathbf{1}_n)^T \mathbf{d}_i = 0$$

$$\rightarrow \lambda_d = -\frac{\mathbf{g}_i^T \mathbf{d}_i}{\mathbf{d}_i^T Q_D \mathbf{d}_i} = \frac{-\mathbf{g}_i^T(-P\mathbf{g}_i)}{(-P\mathbf{g}_i)^T Q_D (-P\mathbf{g}_i)} = \frac{\mathbf{g}_i^T P \mathbf{g}_i}{\mathbf{g}_i^T P Q_D P \mathbf{g}_i} \geq 0,$$

where the last inequality follows from the symmetric positive semidefinite nature of matrices P and Q_D. Furthermore, we also need to mind that while we are moving along the direction $\mathbf{d}_i = -P\mathbf{g}(\boldsymbol{\alpha}_i)$, some other box constraints that were not active before, might become active for some $\lambda < \lambda_d$. That is, some new $\alpha_j \; j \notin \boldsymbol{J}$ might become $\alpha_j = 0$ or $\alpha_j = C$ at point $\boldsymbol{\alpha}_{i+1} = \boldsymbol{\alpha}_i + \lambda \mathbf{d}_i, \; \lambda < \lambda_d$.

To make sure that we do not violate the box constraints, we also determine the following bounding λ-values:

$$\lambda_0 = \min_{j\notin J} -\frac{\alpha_j}{d_j} \quad \lambda_C = \min_{j\notin J} \frac{C - \alpha_j}{d_j},$$

where α_j, d_j are the jth coordinates of vectors $\boldsymbol{\alpha}_i$ and $\mathbf{d}_i$ respectively. Note that in the two above definitions we do not need to be concerned about the indexes $j \in J$, because the projection matrix P guarantees that the corresponding α_j parameters stay constant. To further elaborate, the values λ_0, λ_C follow simply from the equations:

$$\alpha_j + \lambda d_j = 0 \;\text{ and }\; \alpha_j + \lambda d_j = C.$$

Now, the optimal step size λ^* is then defined as (assuming also that $\lambda_0, \lambda_C > 0$):

$$\lambda^* = \max\{0, \min\{\lambda_0, \lambda_C, \lambda_d\}\},$$

and then we proceed to set:

$$\boldsymbol{\alpha}_{i+1} = \boldsymbol{\alpha}_i + \lambda^* \mathbf{d}_i,$$

as our next solution. The whole above process from the beginning of this section is repeated until convergence criteria is met or KKT-conditions are satisfied. We have now constructed all the necessary theory for the GPAS-algorithm and we are ready to define the pseudocode.

5.4 Pseudocode for the GPAS SVM solver

Below I will list the pseudocode that we will implement later in Python code. Note that the below is a mathematical idealization and in practise we need to mind many sort of numerical problems caused by the finite memory limitations of the computer. I have attempted to tackle this problem in the code.

---

1. INPUT: Index $i = 1$, initial feasible solution $\boldsymbol{\alpha}_i$ to problem (31), $M = \mathbf{y}^T = [y_1, \ldots, y_n]$, convergence criteria $\varepsilon > 0$, penalty term $C > 0$.
2. OUTPUT: optimal solution $\boldsymbol{\alpha}^*$ to problem (31).
3.
4. do:
5. Set $\mathbf{g}_i = Q_D \boldsymbol{\alpha}_i - \mathbf{1}$ # Gradient vector
6. Set $P = I - M^T(MM^T)^{-1}M$ # Projection matrix
7. Set $\mathbf{d}_i = -P\mathbf{g}_i$ # Search direction
8. Set $J = \{j \,|\, (\alpha_j = 0 \wedge d_j < 0) \vee (\alpha_j = C \wedge d_j > 0)\}$ # Active indexes
9. while $J \neq \varnothing$: # Update projection matrix accordingly
10. Set $A_1 = \mathbf{0}_{|J| \times n}$
11. for $k = 1, \ldots, |J|$:
12. Set $A_1(k, J(k)) = 1$
13. Set $M = \begin{bmatrix} A_1 \\ \mathbf{y}^T \end{bmatrix}$
14. Set $P = I - M^T(MM^T)^{-1}M$
15. Set $\mathbf{d}_i = -P\mathbf{g}_i$
16. Set $J = \{j \,|\, (\alpha_j = 0 \wedge d_j < 0) \vee (\alpha_j = C \wedge d_j > 0)\}$
17. Set $\lambda_d = \frac{\mathbf{g}_i^T P \mathbf{g}_i}{\mathbf{g}_i^T P Q_D \mathbf{g}_i}$ # Solve optimal and bounding step values
18. Set $\lambda_0 = \min_{i \in \{1,\ldots,n\}} -\frac{\alpha_j}{d_j}$
19. Set $\lambda_C = \min_{i \in \{1,\ldots,n\}} \frac{C - \alpha_j}{d_j}$
20. Set $\lambda^* = \max\{0, \min\{\lambda_d, \lambda_0, \lambda_C\}\}$
21. Set $\boldsymbol{\alpha}_{i+1} = \boldsymbol{\alpha}_i + \lambda^* \mathbf{d}_i$ # Make update step
22. Set $i = i + 1$
23. while $|f(\boldsymbol{\alpha}_i) - f(\boldsymbol{\alpha}_{i-1})| > \varepsilon$: # Check convergence criteria (e.g. KKT-conditions)

FINISHED! We are now done with constructing the theory and algorithms and now it's time to make some examples with code. In the following, I will first present some examples which are then followed by the code based on everything we've been discussing in this tutorial. I hope this tutorial helped you!

I will continue to improve this tutorial as I find bugs etc.

You can contact me at jjepoh(at)utu(dot)fi

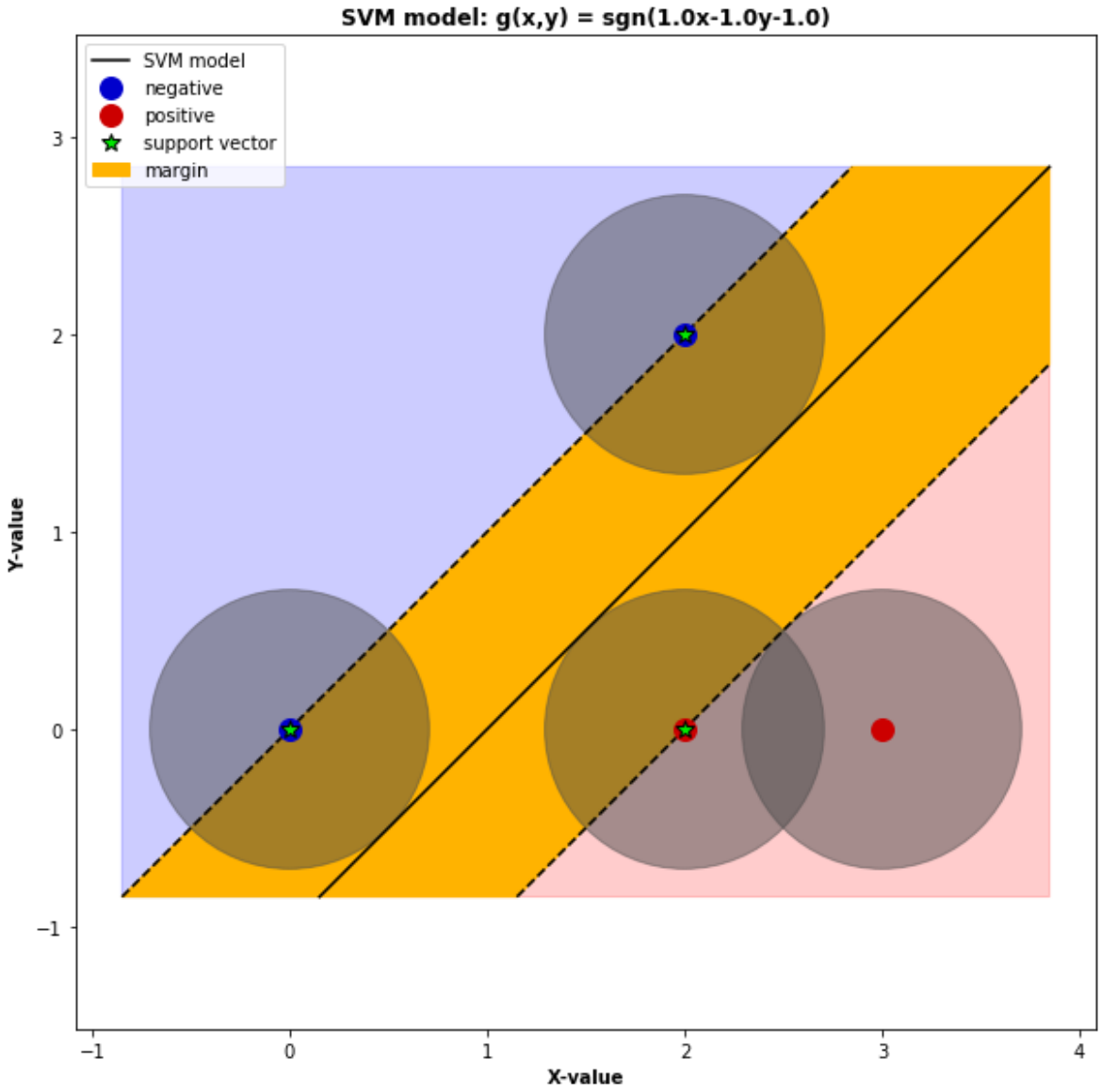
Best regards, Jonne Pohjankukka 31.5.2019

In [1]:

```
##########################################################################
#
# Copyright Jonne Pohjankukka 2019
#
# You can use the below code freely with reference requirement.
#
# Note! while experimenting with the code, it is good to avoid very
# large and small values, since these can cause numerical instabilities
# with the current method due to finite memory limitations of the computer.
#
##########################################################################
```

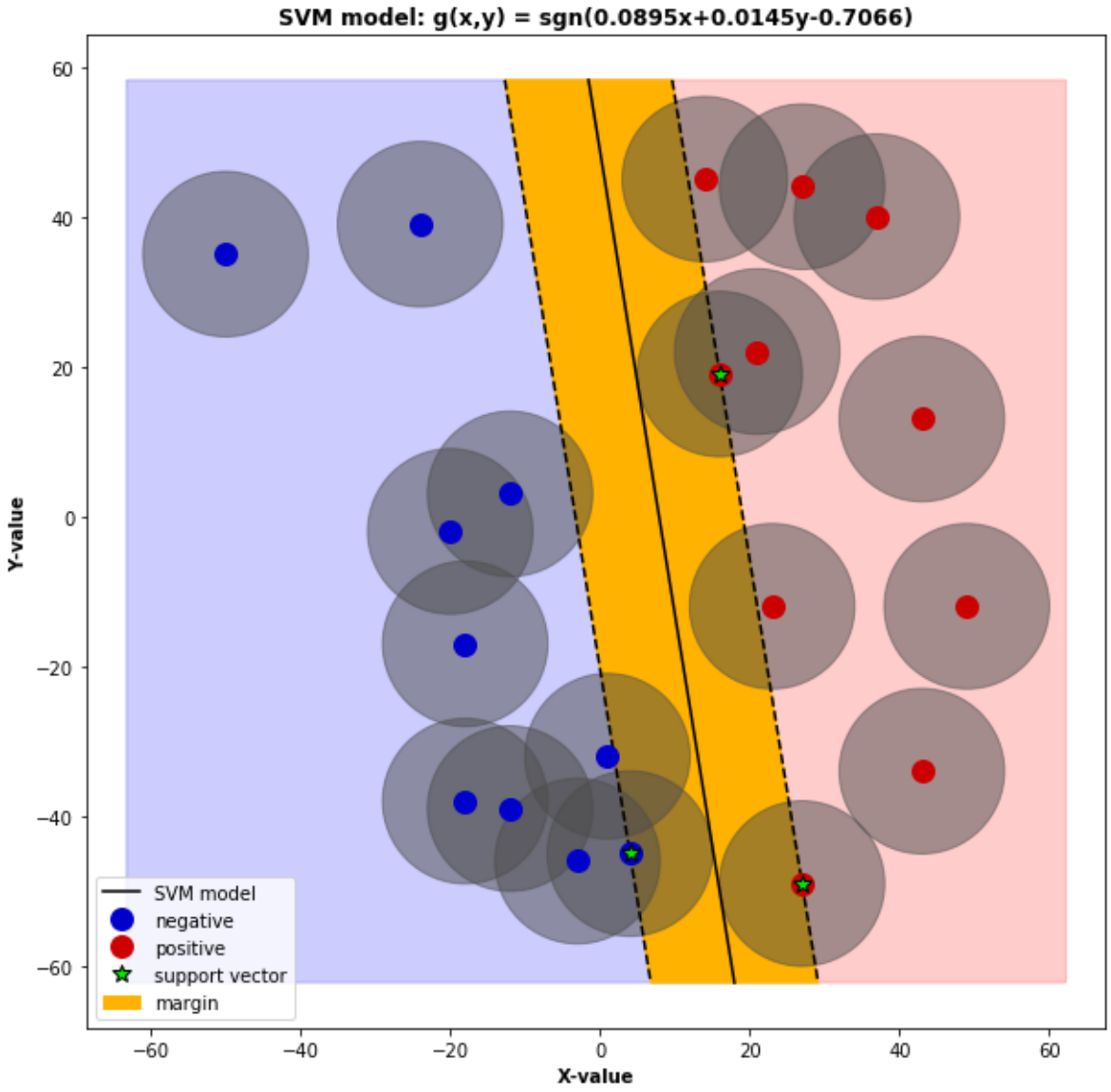
In [89]:

```
##############################################################################################
# *****
# *****
# ***** DEMONSTRATION I: Hard margin, linear kernel SVM with custom data set.
# *****
# *****
##############################################################################################

# Step 1: Create custom data set (same as in the tutorial)
X= np.array([[0,0], [2,2], [2,0], [3,0]])
y = np.array([[-1, -1, 1, 1]])
# Step 2: Train the SVM
# Set the tolerance for how close current solution needs to be to the previous one befo
error_tolerance = np.finfo(float).eps
# Set the number of training iterations
iterations = 100
# Set other parameters, sigma for Gaussian kernel; lambda, gamma and q for polynomial k
kwargs = {"sigma":1, "lambda":0, "gamma":1, "q":1, "printInfo":False}
# Create the SVM object with infinite penalty term (inf, hard-margin SVM) with the gene
SVM_model = SVM(X, y, "linear", iterations, error_tolerance, np.inf, kwargs)
# Begin training, GPAS-algorithm
SVM_model.train()
# Plot the resulting hyperplane and data
SVM_model.plot2D()
```


SVM model: g(x,y) = sgn(1.0x-1.0y-1.0)
SVM model
negative
positive
support vector
margin
Y-value
X-value
3
2
1
0
−1
−1
0
1
2
3
4

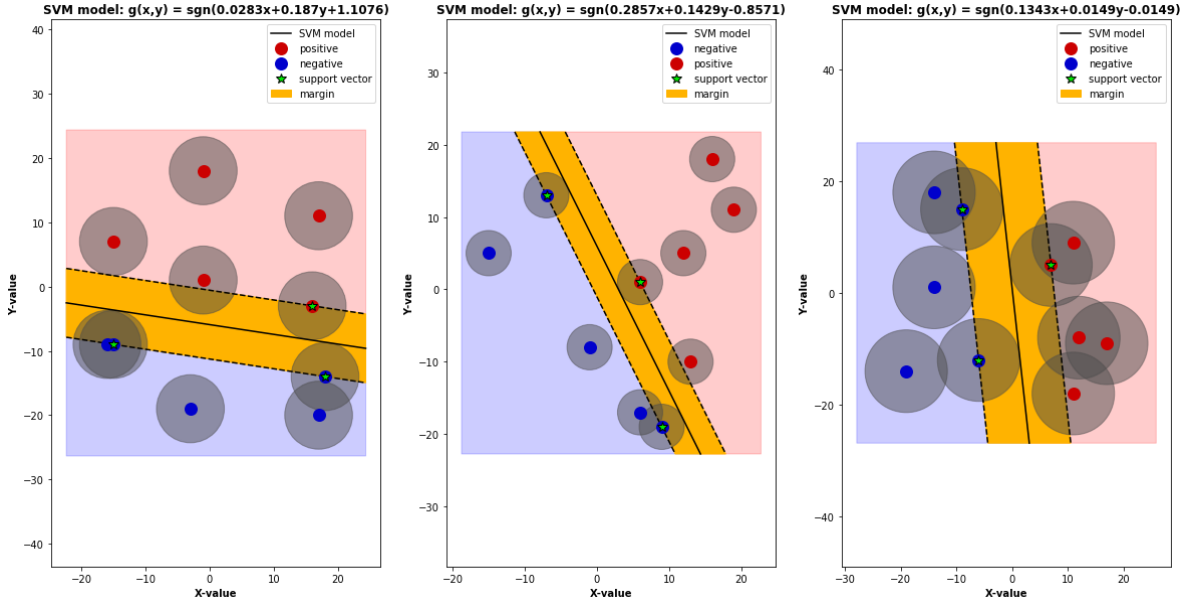
In [86]:

```
##############################################################################################
# *****
# *****
# ***** DEMONSTRATION II: Hard margin, linear kernel SVM with random data set.
# *****
# *****
##############################################################################################

# Step 1: Create random data set
value_selection_interval = [-50,50]
number_of_positive_samples = 10
number_of_negative_samples = 10
X_feature_dimensions = 2
(X,y) = createRandomDataSetLinear(X_feature_dimensions, number_of_positive_samples, nu

# Step 2: Train the SVM
# Set the tolerance for how close current solution needs to be to the previous one bef
error_tolerance = np.finfo(float).eps
# Set the number of training iterations
iterations = 10000
# Set other parameters, sigma for Gaussian kernel; lambda, gamma and q for polynomial
kwargs = {"sigma":1, "lambda":0, "gamma":1, "q":1, "printInfo":False}
# Create the SVM object with infinite penalty term (inf, hard-margin SVM) with the gen
SVM_model = SVM(X, y, "linear", iterations, error_tolerance, np.inf, kwargs)
# Begin training, GPAS-algorithm
SVM_model.train()
# Plot the resulting hyperplane and data
SVM_model.plot2D()
```


SVM model: g(x,y) = sgn(0.0895x+0.0145y-0.7066)
Y-value
X-value
SVM model
negative
positive
support vector
margin

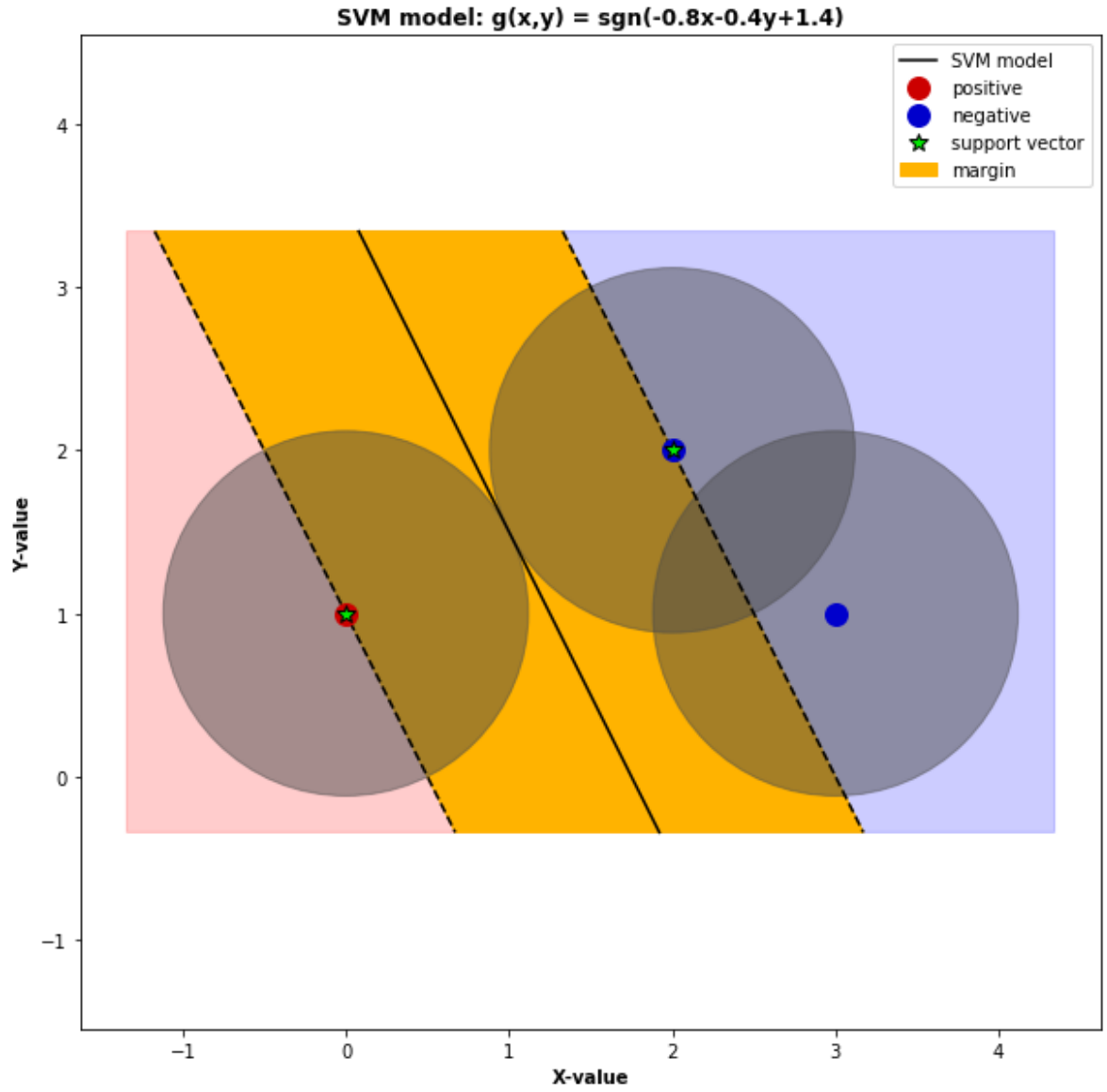
In [87]:

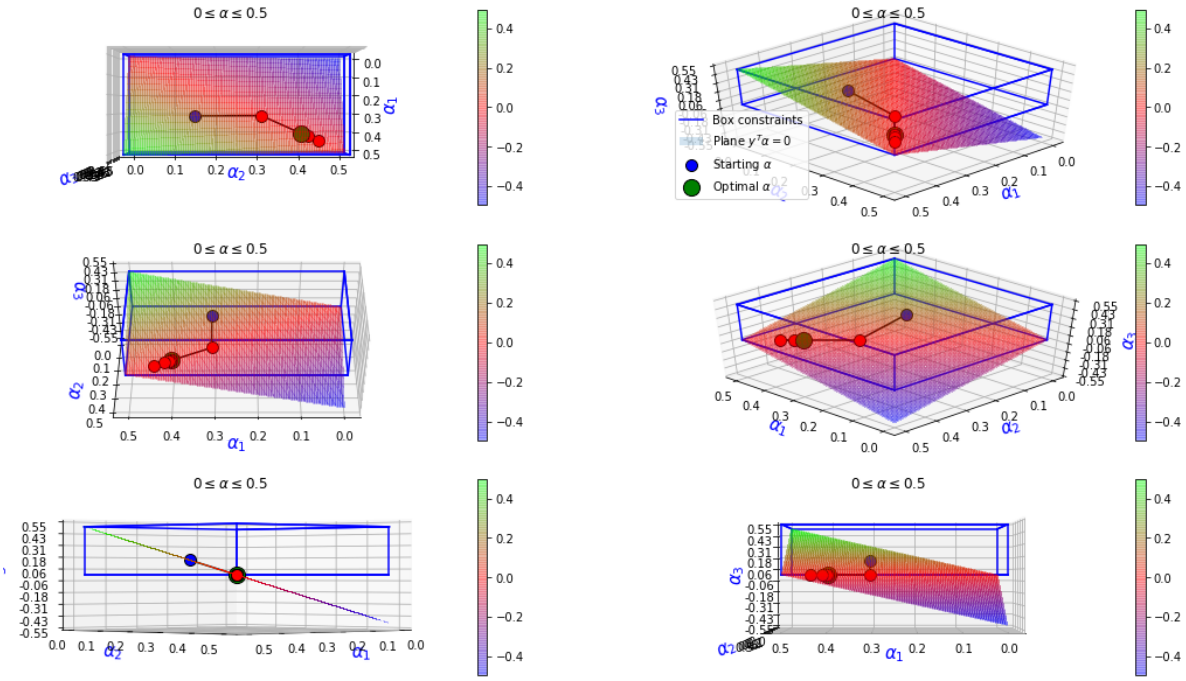
```
##############################################################################################
# *****
# *****
# ***** DEMONSTRATION III: Plot multiple linear hard margin SVMs with random data sets
# *****
# *****
##############################################################################################
plotMultipleSVMs(1,3)
```

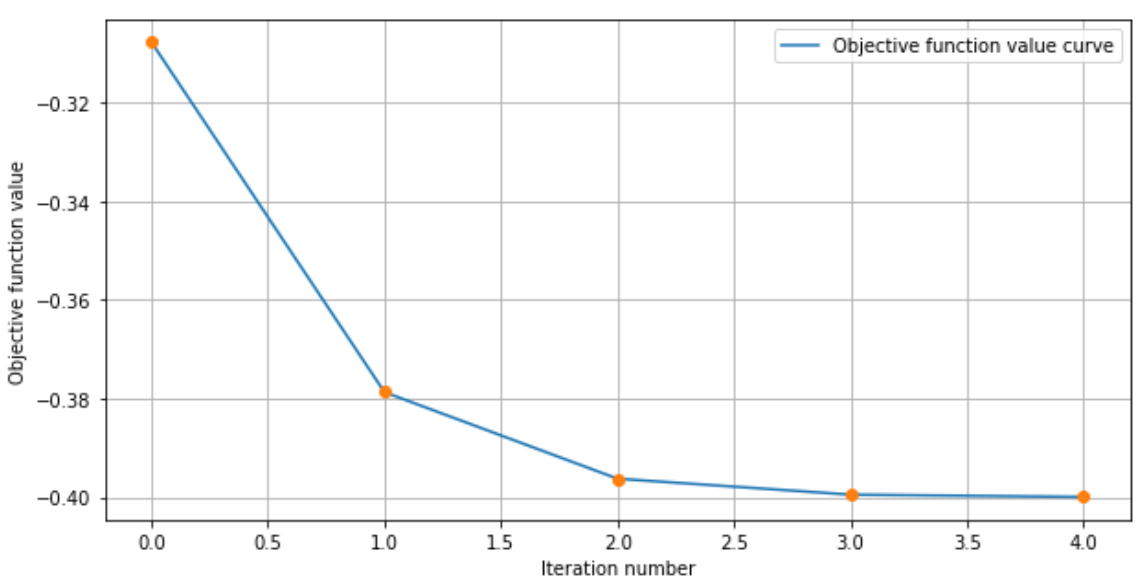


In [88]:

```
##########################################################################################
# *****
# *****
# ***** DEMONSTRATION IV: Plot the error curve and the development of the optimization
# in a simple soft-margin (C=0.5) SVM case.
# *****
# *****
##########################################################################################

# Step 1: Create custom data set (same as in the tutorial)
X= np.array([[0,1], [2,2], [3,1]])
y = np.array([[1, -1, -1]])
# Step 2: Train the SVM
# Set the tolerance for how close current solution needs to be to the previous one befo
error_tolerance = np.finfo(float).eps
# Set the number of training iterations
iterations = 100
# Set other parameters, sigma for Gaussian kernel; lambda, gamma and q for polynomial k
kwargs = {"sigma":1, "lambda":0, "gamma":1, "q":1, "printInfo":False}
# Create the SVM object with infinite penalty term (inf, hard-margin SVM) with the gene
SVM_model = SVM(X, y, "linear", iterations, error_tolerance, .5, kwargs)
# Begin training, GPAS-algorithm
SVM_model.train()
# Plot the resulting hyperplane and data
SVM_model.plot2D()
SVM_model.plotAlphaCurve()
SVM_model.plotErrorCurve()
```


SVM model: g(x,y) = sgn(-0.8x-0.4y+1.4)
SVM model
positive
negative
support vector
margin
Y-value
X-value

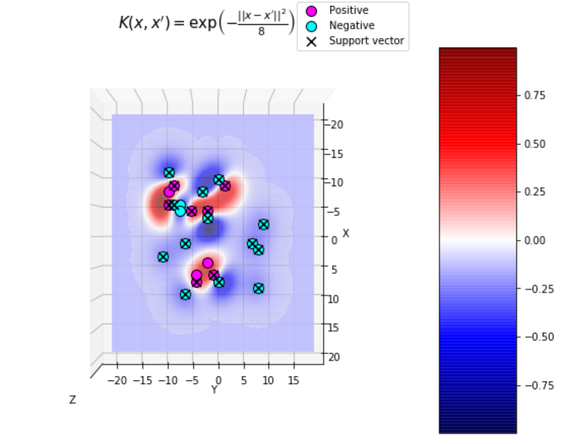

$0 \leq \alpha \leq 0.5$
Box constraints
Plane $y^T\alpha = 0$
Starting α
Optimal α

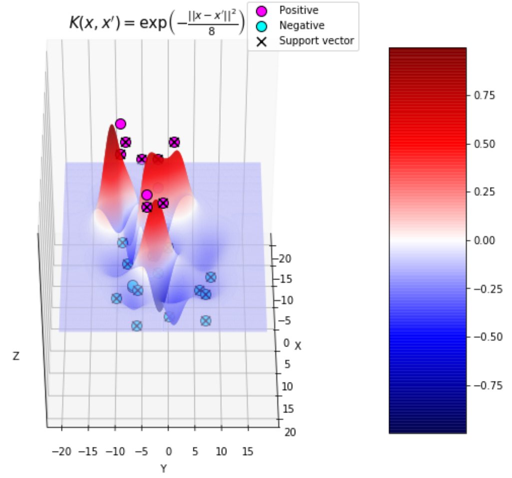

Objective function value curve
Objective function value
−0.32
−0.34
−0.36
−0.38
−0.40
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Iteration number

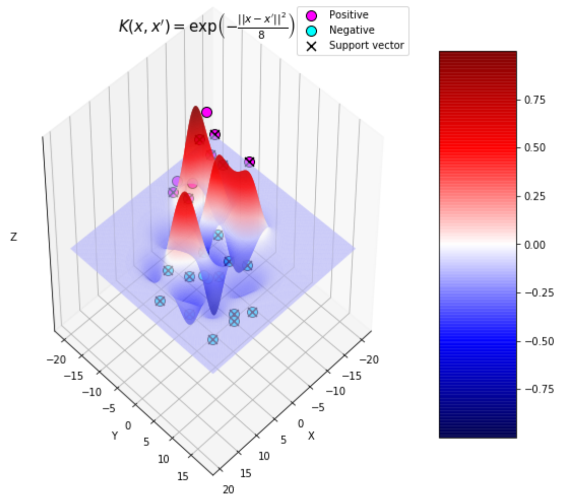
In [17]:

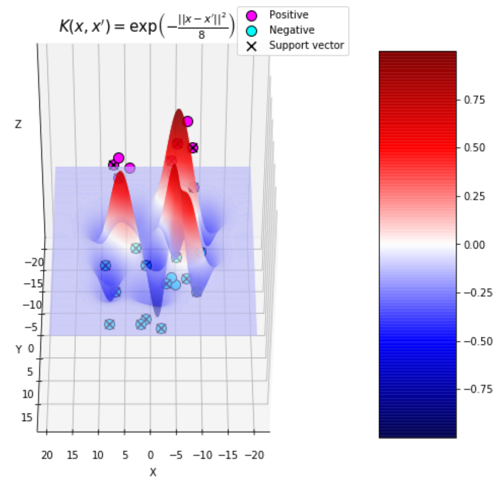
```
##########################################################################################
# *****
# *****
# ***** DEMONSTRATION V: Hard margin, Gaussian kernel SVM with random data set, narrow
# *****
# *****
##########################################################################################

# Step 1: Create custom data set (same as in the tutorial)
(X,y)=createRandomDataSetNL(2, 10, 15, [-10,10])
# Step 2: Train the SVM
# Set the tolerance for how close current solution needs to be to the previous one befo
error_tolerance = np.finfo(float).eps
# Set the number of training iterations
iterations = 20000
# Set other parameters, sigma for Gaussian kernel; lambda, gamma and q for polynomial k
kwargs = {"sigma":2, "lambda":1, "gamma":1, "q":2, "printInfo":True}
# Create the SVM object with infinite penalty term (inf, hard-margin SVM) with the gene
SVM_model = SVM(X, y, "Gaussian", iterations, error_tolerance, np.inf, kwargs)
# Begin training, GPAS-algorithm
SVM_model.train()
# Plot the resulting hyperplane and data
SVM_model.plot2DNK()
```

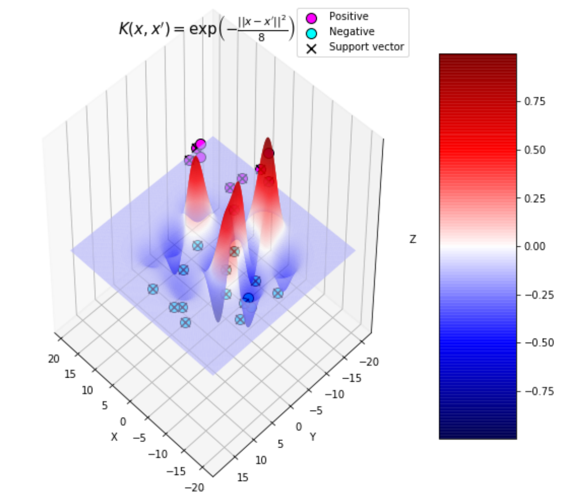
```
Using Gaussian kernel
Transforming alphas to hyperplane parameters alpha --> w,b
 number of support vectors: 20
```

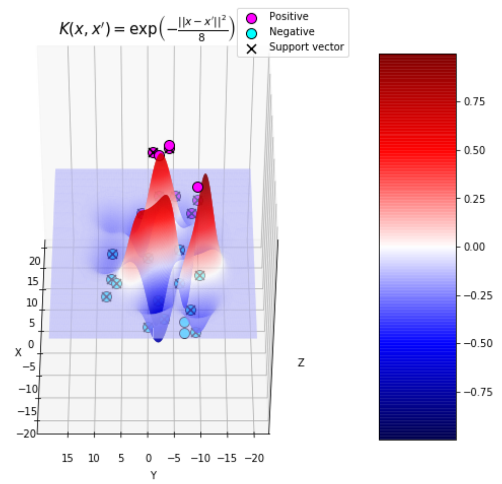

$K(x, x') = \exp\left(-\frac{||x-x'||^2}{8}\right)$
Positive
Negative
Support vector


$K(x, x') = \exp\left(-\frac{||x-x'||^2}{8}\right)$
Positive
Negative
Support vector


$K(x, x') = \exp\left(-\frac{||x-x'||^2}{8}\right)$
Positive
Negative
Support vector


$K(x, x') = \exp\left(-\frac{||x-x'||^2}{8}\right)$
Positive
Negative
Support vector

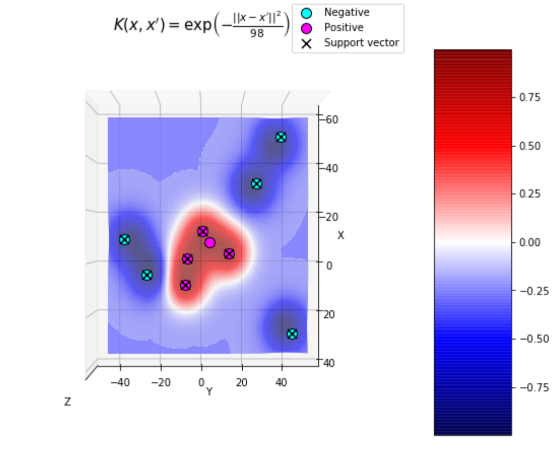

$K(x, x') = \exp\left(-\frac{||x-x'||^2}{8}\right)$
Positive
Negative
Support vector


$K(x, x') = \exp\left(-\frac{||x-x'||^2}{8}\right)$
Positive
Negative
Support vector

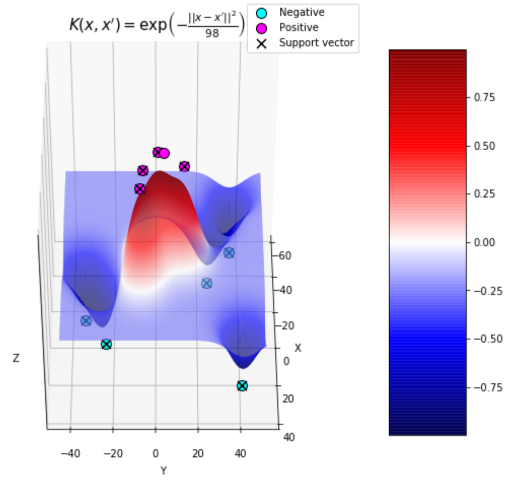
In [18]:

```
  ##############################################################################
# *****
# *****
# ***** DEMONSTRATION VI: Hard margin, Gaussian kernel SVM with radial data set, wide k
# *****
# *****
##############################################################################

# Step 1: Create custom data set (same as in the tutorial)
(X,y)=createRandomDataSetCircle(2, 5, 5, [-50,50], 15, 7)
# Step 2: Train the SVM
# Set the tolerance for how close current solution needs to be to the previous one befo
error_tolerance = np.finfo(float).eps
# Set the number of training iterations
iterations = 5000
# Set other parameters, sigma for Gaussian kernel; lambda, gamma and q for polynomial k
kwargs = {"sigma":7, "lambda":1, "gamma":1, "q":2, "printInfo":True}
# Create the SVM object with infinite penalty term (inf, hard-margin SVM) with the gene
SVM_model = SVM(X, y, "Gaussian", iterations, error_tolerance, np.inf, kwargs)
# Begin training, GPAS-algorithm
SVM_model.train()
# Plot the resulting hyperplane and data
SVM_model.plot2DNK()
```

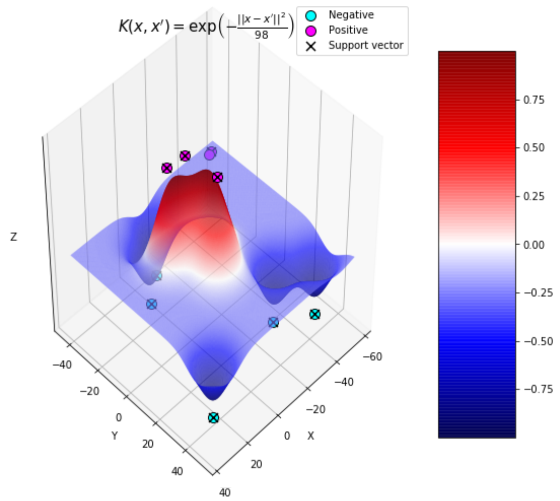
```
Using Gaussian kernel
Transforming alphas to hyperplane parameters alpha --> w,b
 number of support vectors: 9
```



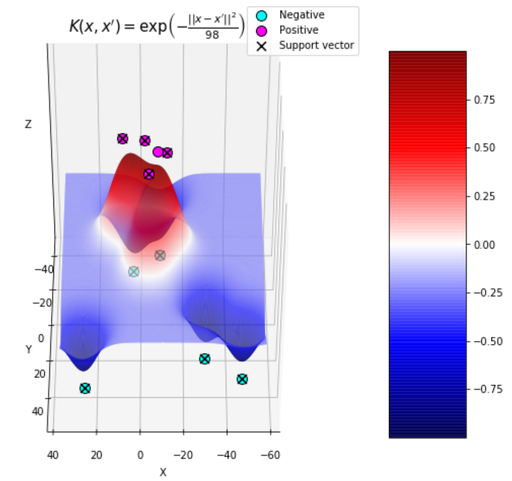
$K(x, x') = \exp\left(-\frac{||x - x'||^2}{98}\right)$
Negative
Positive
Support vector
X
Y
Z
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75



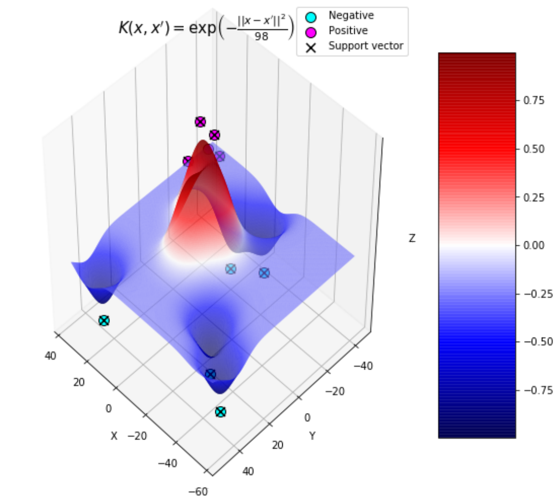
$K(x, x') = \exp\left(-\frac{||x - x'||^2}{98}\right)$
Negative
Positive
Support vector
X
Y
Z
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75



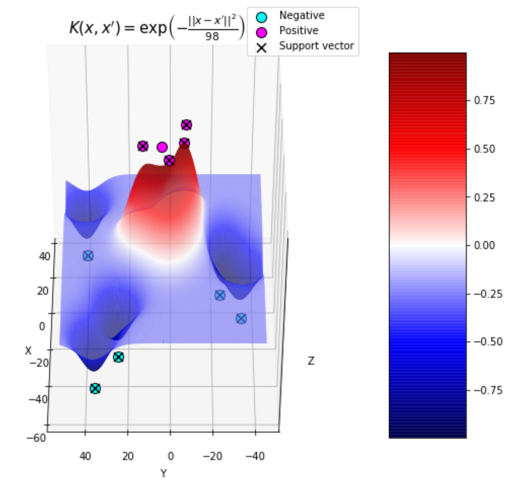
$K(x, x') = \exp\left(-\frac{||x - x'||^2}{98}\right)$
Negative
Positive
Support vector
X
Y
Z
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75



$K(x, x') = \exp\left(-\frac{||x - x'||^2}{98}\right)$
Negative
Positive
Support vector
X
Y
Z
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75



$K(x, x') = \exp\left(-\frac{||x - x'||^2}{98}\right)$
Negative
Positive
Support vector
X
Y
Z
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75

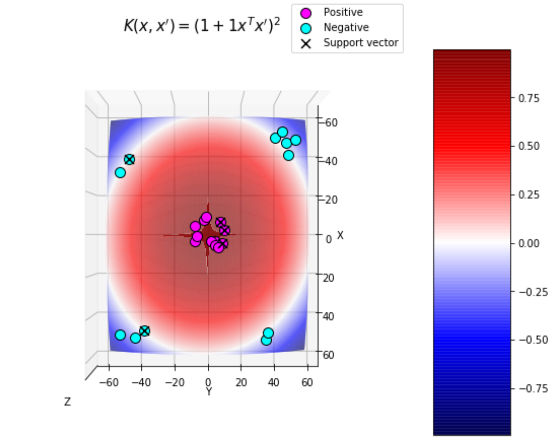


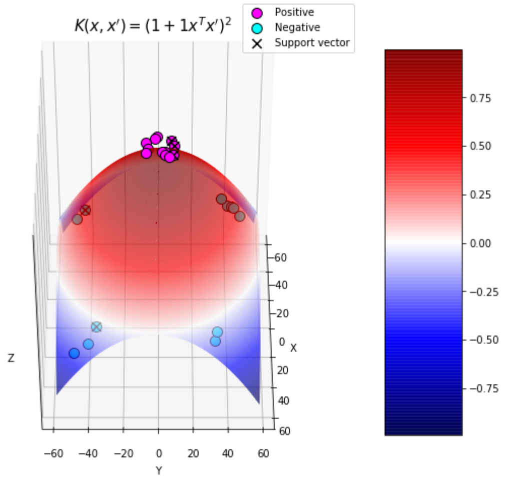
$K(x, x') = \exp\left(-\frac{||x - x'||^2}{98}\right)$
Negative
Positive
Support vector
X
Y
Z
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75

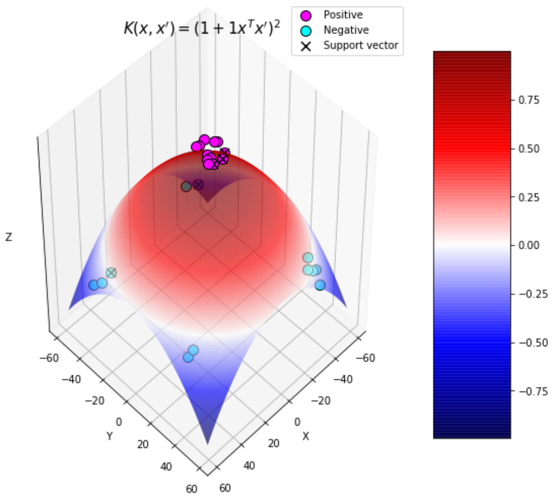
In [19]:

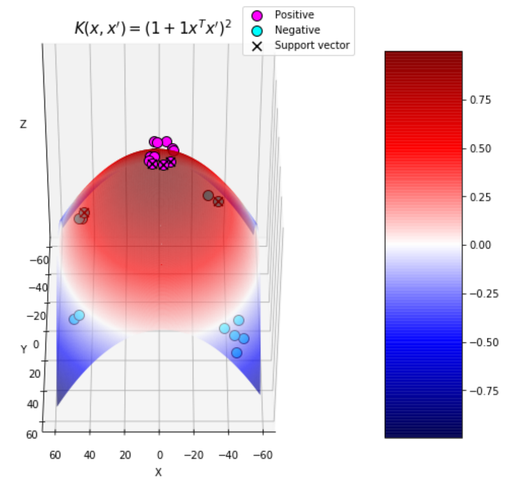
```
##########################################################################################
# *****
# *****
# ***** DEMONSTRATION VII: Hard margin, 2nd degree polynomial kernel SVM with radial da
# *****
# *****
##########################################################################################

# Step 1: Create custom data set (same as in the tutorial)
(X,y) = createRandomDataSetCircle(2, 12, 12, [-50,50], 10, 45)
# Step 2: Train the SVM
# Set the tolerance for how close current solution needs to be to the previous one befo
error_tolerance = np.finfo(float).eps
# Set the number of training iterations
iterations = 15000
# Set other parameters, sigma for Gaussian kernel; lambda, gamma and q for polynomial k
kwargs = {"sigma":2, "lambda":1, "gamma":1, "q":2, "printInfo":True}
# Create the SVM object with infinite penalty term (inf, hard-margin SVM) with the gene
SVM_model = SVM(X, y, "polynomial", iterations, error_tolerance, np.inf, kwargs)
# Begin training, GPAS-algorithm
SVM_model.train()
# Plot the resulting hyperplane and data
SVM_model.plot2DNK()
```

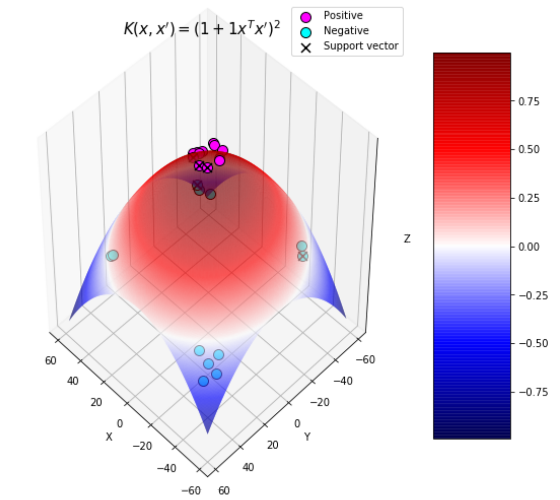
```
Using polynomial kernel
Transforming alphas to hyperplane parameters alpha --> w,b
 number of support vectors: 5
```

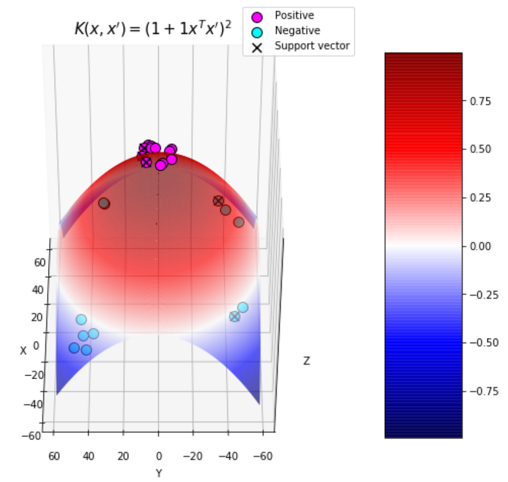

$K(x, x') = (1 + 1x^Tx')^2$
Positive
Negative
Support vector


$K(x, x') = (1 + 1x^Tx')^2$
Positive
Negative
Support vector


$K(x, x') = (1 + 1x^Tx')^2$
Positive
Negative
Support vector


$K(x, x') = (1 + 1x^Tx')^2$
Positive
Negative
Support vector


$K(x, x') = (1 + 1x^Tx')^2$
Positive
Negative
Support vector


$K(x, x') = (1 + 1x^Tx')^2$
Positive
Negative
Support vector

In [77]:

```
##########################################################################################
##########################################################################################
##########################################################################################
# *****
# *****
# ***** The relevant code regarding the tutorial starts from here. If you want to ski
# *****
# *****
##########################################################################################
##########################################################################################
##########################################################################################

# First we import the most relevant package
import numpy as np
# Second we need some visualization tools
import matplotlib.pyplot as plt
# Show the plots in the notebook
%matplotlib inline
# And the rest are for bookkeeping, ignoring non-relevant warnings, checking for erro
import sys
import warnings
import time
warnings.filterwarnings("ignore")
# The rest imports here are for visualization purposes
from mpl_toolkits import mplot3d
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
from mpl_toolkits.axes_grid1 import make_axes_locatable
from mpl_toolkits.mplot3d import Axes3D

#######################################
#
# DESCRIPTION:
# - Function used for generating a
# scattered nonlinear random data set
# with -1/+1 labels.
# -----------------------------------
# INPUT: input data dimension, number
# of positive and negative +1/-1 samples
# integer interval of values where data
# points are to be sampled. Integer val-
# ues are used to improve numerical
# accuracy.
# -----------------------------------
# OUTPUT: input data X and label
# output y.
#
#######################################
def createRandomDataSet(input_dimension, number_of_positive_samples, number_of_negati
    # Create data structures
    number_of_data_points = number_of_positive_samples + number_of_negative_samples
    X = np.zeros((number_of_data_points, input_dimension), dtype=np.float64)
    y = np.zeros((1, number_of_data_points), dtype=np.float64)
    positives_found = 0
    negatives_found = 0
    current_index = 0
    while positives_found < number_of_positive_samples:
        X[current_index,:] = np.random.randint(interval[0], interval[1], (1, input_di
        y[0,current_index] = 1
```

```
        positives_found += 1
        current_index += 1
    while negatives_found < number_of_negative_samples:
        X[current_index,:] = np.random.randint(interval[0], interval[1], (1, input_di
        y[0,current_index] = -1
        negatives_found += 1
        current_index += 1
    return(X, y)

########################################
#
# DESCRIPTION:
# - Function used for generating a
# random data set separated by a circle
# ----------------------------------
# INPUT: input data dimension, number
# of positive and negative +1/-1 samples
# integer interval of values where data
# points are to be sampled. Integer val-
# ues are used to improve numerical
# accuracy. Tolerance parameter is
# used to adjust how large distance
# between the two data clusters we
# want (i.e. margin)
# ----------------------------------
# OUTPUT: input data X and label
# output y.
#
########################################
def createRandomDataSetRadial(input_dimension, number_of_positive_samples, number_of_
    # Create data structures
    number_of_data_points = number_of_positive_samples + number_of_negative_samples
    X = np.zeros((number_of_data_points, input_dimension), dtype=np.float64)
    y = np.zeros((1, number_of_data_points), dtype=np.float64)
    positives_found = 0
    negatives_found = 0
    current_index = 0
    # Generate random data points until suitable points found
    while positives_found + negatives_found < number_of_data_points:
        input_data_point = np.random.randint(interval[0], interval[1], (1, input_dime
        if input_data_point[0,0]**2 + input_data_point[0,1]**2  <= radius**2 and posi
            X[current_index,:] = input_data_point
            y[0,current_index] = 1
            positives_found += 1
            current_index += 1
        elif input_data_point[0,0]**2 + input_data_point[0,1]**2  > (radius+tolerance
            X[current_index,:] = input_data_point
            y[0,current_index] = -1
            negatives_found += 1
            current_index += 1
    return(X, y)

########################################
#
# DESCRIPTION:
# - Function used for generating a
# random data set separated by a linear
# model with -1/+1 labels.
# ----------------------------------
# INPUT: input data dimension, number
# of positive and negative +1/-1 samples
```

```
# integer interval of values where data
# points are to be sampled. Integer val-
# ues are used to improve numerical
# accuracy.
# ----------------------------------
# OUTPUT: input data X and label
# output y.
#
########################################
def createRandomDataSetLinear(input_dimension, number_of_positive_samples, number_of_
    # Create data structures
    number_of_data_points = number_of_positive_samples + number_of_negative_samples
    X = np.zeros((number_of_data_points, input_dimension), dtype=np.float64)
    y = np.zeros((1, number_of_data_points), dtype=np.float64)
    # Create random line
    line_normal_vector = np.random.rand(input_dimension, 1)
    positives_found = 0
    negatives_found = 0
    current_index = 0
    # Generate random data points until suitable points found
    while positives_found + negatives_found < number_of_data_points:
        input_data_point = np.random.randint(interval[0], interval[1], (1, input_dime
        if input_data_point@line_normal_vector > 0 and positives_found < number_of_po
            X[current_index,:] = input_data_point
            y[0,current_index] = 1
            positives_found += 1
            current_index += 1
        elif input_data_point@line_normal_vector < 0 and negatives_found < number_of_
            X[current_index,:] = input_data_point
            y[0,current_index] = -1
            negatives_found += 1
            current_index += 1
    return(X, y)

########################################
#
# DESCRIPTION: Class for the SVM model
#
########################################
class SVM:

    def __init__(self, X, y, kernel, iterations, error_bound, C, kwargs):
        self.input_X = X
        self.output_y = y
        self.kernel = kernel
        self.maxIterations = iterations
        self.errorBound = error_bound
        self.penalty_term = C
        self.kwargs = kwargs
        self.alpha = None
        self.alphas = []
        self.errors = []
        self.w = None
        self.b = None

    def getKernel(self):
        return self.kernel

    def getPenaltyTerm(self):
        return self.penalty_term

```

```
    ########################################
    #
    # DESCRIPTION:
    # - Function used for solving the SVM
    # model alpha parameters. This function
    # implements the algorithm of section
    # 5.4
    # -------------------------------------
    # INPUT: input data X and labels y and
    # additional parameters of SVM object
    # -------------------------------------
    # OUTPUT: The optimal alpha vector,
    # weights w and bias b of solved
    # SVM model.
    #
    ########################################
    def checkActiveConstraints(self, alpha, d, M, g, I):
        # Check the active set constraints and update d accordingly
        active_set_indexes = []
        finished = False
        # We try to avoid really small numbers, because they cause numerical instabil
        alpha[np.where(np.abs(alpha) < np.finfo(float).eps*10)] = 0.0
        d[np.where(np.abs(d) < np.finfo(float).eps*10)] = 0.0
        while not finished:
            # Find out if any alpha indexes become active or not
            lower_active_set_bool = np.logical_and(alpha == 0, d < 0)
            upper_active_set_bool = np.logical_and(alpha == self.penalty_term, d > 0)
            active_set_indexes = np.where(lower_active_set_bool | upper_active_set_bo
            # If true, we have active constraints and we need to update M
            if len(active_set_indexes) > 0:
                for i in range(0, len(active_set_indexes)):
                    active_constraint = np.zeros((1, alpha.shape[0]), dtype=np.float6
                    active_constraint[0, active_set_indexes[i]] = 1.0
                    M = np.vstack([active_constraint, M])
            # M has been updated, check that some other semiactive constrains are not
            # Recalculate search direction d
            Mt = M.transpose()
            MMt_1 = np.linalg.inv(M@Mt) # @ stands for matrix multiplication
            P = I - Mt@MMt_1@M
            d = -P@g(alpha)
            # Avoid very small values
            d[np.where(np.abs(d) < np.finfo(float).eps*10)] = 0.0
            # Now we need another check. Since we have updated d, some constraints th
            # not active before might become active in the new updated d
            lower_active_set_bool = np.logical_and(alpha == 0, d < 0)
            upper_active_set_bool = np.logical_and(alpha == self.penalty_term, d > 0)
            active_set_indexes = np.where(lower_active_set_bool | upper_active_set_bo
            # Now we need to have a second check and do the process possibly again
            # depending if some constraints have become active in the new search dire
            if len(active_set_indexes) > 0:
                active_set_indexes = [] # Empty the list because loop starts over
            else:
                finished = True
        return(d, active_set_indexes)

    ########################################
    #
    # DESCRIPTION:
    # - Function used for solving the SVM
    # model alpha parameters. This function
    # implements the algorithm of section
```

```
    # 5.4
    # ------------------------------------
    # INPUT: input data X and labels y and
    # additional parameters of SVM object
    # ------------------------------------
    # OUTPUT: The optimal alpha vector,
    # weights w and bias b of solved
    # SVM model.
    #
    ########################################
    def train(self):
        data = self.input_X
        labels = self.output_y
        epsilon = self.errorBound
        maxIters = self.maxIterations
        kwargs = self.kwargs
        current_iteration = 1
        alpha = self.generateRandomAlpha(labels)
        prev_alpha = np.ones((data.shape[0], 1), dtype=np.float64)*np.inf # Keep trac
        alpha = np.zeros((data.shape[0], 1), dtype=np.float64)
        Q_D = self.getKernelMatrix()
        # Construct the objective function (i.e. the function to be minimized) and it
        f = lambda alpha : 0.5*alpha.transpose()@Q_D@alpha - np.sum(alpha)
        g = lambda alpha : Q_D@alpha - np.ones([alpha.shape[0],1], dtype=np.float64)
        # Next, we construct the initial projection matrix, only hyperplane constrain
        # see lemma in section 5.2
        M = labels
        Mt = M.transpose()
        MMt_1 = np.linalg.inv(M@Mt)
        I = np.identity(np.max(labels.shape), dtype=np.float64)
        P = I - Mt@MMt_1@M
        # Initial search direction
        d = -P@g(alpha)
        while np.any(np.abs(alpha-prev_alpha) >= epsilon) and current_iteration <= ma
            # Check for active constraints and return updated d if required
            (d, active_set_indexes) = self.checkActiveConstraints(alpha, d, M, g, I)
            # Find out which indexes are inactive
            non_active_indexes = [i for i in range(0, data.shape[0]) if i not in acti
            # Solve for the lambda values 'edge' values 0 <= a <= C
            lambda_0 = np.inf
            lambda_C = np.inf
            list_a0 = [l for l in -np.divide(alpha[non_active_indexes], d[non_active_
            list_aC = [l for l in np.divide(self.penalty_term-alpha[non_active_indexe
            if len(list_a0) > 0:
                lambda_0 = np.min(list_a0)
            if len(list_aC) > 0:
                lambda_C = np.min(list_aC)
            lambda_values = [lambda_0, lambda_C]
            lambda_values = [l for l in lambda_values if l > 0]
            # Solve for the line search optimizing lambda
            gPQ_DPg = g(alpha).transpose()@P@Q_D@P@g(alpha)
            gPg = g(alpha).transpose()@P@g(alpha)
            lambda_d = gPg / float(gPQ_DPg)
            lambda_d = lambda_d[0,0]
            lambda_values.append(lambda_d)
            opt_lambda = np.nanmax([0.0, np.min(lambda_values)])
            # Next make the update step
            prev_alpha = alpha[:]
            alpha = alpha + opt_lambda*d # Update the alpha vector with optimal step
            # Transform very small alpha values to zero (for numerical reasons)
            alpha[np.where(np.abs(alpha) < np.finfo(float).eps)] = 0.0
```

```
            # Check that alpha vector satisfies constraints
            if any(a < 0 for a in alpha) == True or any(a > self.penalty_term for a i
                sys.exit('Negative or penalty term violating entry in alpha vector!')
            if self.kwargs["printInfo"]:
                print("Current iteration: " + str(current_iteration) + "/" + str(maxI
                print("Objective function value change: " + str(f(prev_alpha)) + " --
            current_iteration += 1
            self.alphas.append(alpha)
            self.errors.append(f(alpha))
            if np.abs(f(alpha)-f(prev_alpha)) == 0: # Stop learning, no change
                break
        # Save for the optimal found alpha
        self.alpha = alpha
        # Save the SVM parameters w, b
        self.setSvmParameters(alpha, data, labels)

    ########################################
    #
    # DESCRIPTION:
    # - Function used for solving the SVM
    # model weight parameters W and bias
    # term. Direct application of the
    # equations in (23) and (24) in the
    # tutorial.
    # -------------------------------------
    # INPUT: The solved optimal alpha^*
    # values, input data X and labels y.
    # -------------------------------------
    # OUTPUT: The weights w and bias b of
    # solved SVM model.
    #
    ########################################
    def setSvmParameters(self, alpha, X, y):
        number_of_data_points = X.shape[0]
        w = np.zeros([1, X.shape[1]], dtype=np.float64)
        support_vector_alphas_inds = []
        for i in range(0, number_of_data_points):
            w += y[0,i]*alpha[i,0]*X[i]
            if alpha[i] > 0: # Collect support vector alphas
                support_vector_alphas_inds.append(i)
        if self.kwargs["printInfo"]:
            print("Transforming alphas to hyperplane parameters alpha --> w,b\n numbe
        support_vector_y = y[0,support_vector_alphas_inds[0]] # Assuming at least one
        support_vector_x = X[support_vector_alphas_inds[0], :] # Assuming at least on
        b = 1/float(support_vector_y)
        for i in range(0, number_of_data_points):
            b -= y[0,i]*alpha[i,0]*np.dot(X[i,:], support_vector_x)
        self.w = w[0]
        self.b = b

    ########################################
    #
    # DESCRIPTION:
    # - Function used for generating the
    # kernel function Q_D. Three options:
    # 'linear', 'Gaussian', 'polynomial'.
    # -------------------------------------
    # INPUT: numpy array of feature data,
    # numpy array of labels -1/+1,
    # additional parameters kwargs
    # -------------------------------------
```

```
    # OUTPUT: kernel matrix Q_D.
    #
    ######################################
    #def getKernelMatrix(self, data, labels, **kwargs):
    def getKernelMatrix(self):
        data = self.input_X
        labels = self.output_y
        kwargs = self.kwargs
        Q_D = np.zeros([np.size(labels), np.size(labels)], dtype=np.float64)
        kernel_function = None
        if kwargs["printInfo"]:
            print("Using " + self.kernel + " kernel")
        if self.kernel == "linear":
            kernel_function = lambda x, y : np.dot(x,y)
        elif self.kernel == "Gaussian":
            kernel_function = lambda x, y : np.exp(-np.dot(x-y,x-y)/(2*kwargs["sigma"
        elif self.kernel == "polynomial":
            kernel_function = lambda x, y : (kwargs["lambda"] + kwargs["gamma"]*np.do
        for i in range(0, np.size(labels)):
            for j in range(0, np.size(labels)):
                Q_D[i,j] = labels[0,i]*labels[0,j]*kernel_function(data[i,:], data[j,
        # Check for very small values, to improve numeric stability
        Q_D[np.where(np.abs(Q_D) < np.finfo(float).eps)] = 0
        return Q_D

    ######################################
    #
    # DESCRIPTION:
    # - Function used for generating a
    # random feasible alpha solution a,
    # that is yTa = 0, a >= 0.
    # ------------------------------------
    # INPUT: numpy array of labels -1/+1
    # ------------------------------------
    # OUTPUT: a feasible alpha solution of
    # numpy array type.
    # ------------------------------------
    # ASSUMPTIONS: There must exist at
    # least one +1 and -1 label.
    #
    ######################################
    def generateRandomAlpha(self, labels):
        suitable_alpha_found = False
        while not suitable_alpha_found:
            data_length = np.size(labels)
            # We use double floating point precision to get best precision as possibl
            alpha = np.zeros([data_length,1], dtype=np.float64)
            p_inds = np.where(labels > 0)[1]
            n_inds = np.where(labels < 0)[1]
            # Check that we have at least one +1/-1 pair
            if np.size(p_inds) == 0 or np.size(n_inds) == 0:
                sys.exit('There must be at least one +1 and -1 data point!')
            # Generate random constraint satisfying alpha values for negative cases
            if self.penalty_term < np.inf:
                alpha[n_inds,0] = np.random.rand(np.size(n_inds))*self.penalty_term
            else:
                alpha[n_inds,0] = np.random.rand(np.size(n_inds))
            negSum = np.sum(alpha)
            # The next step is just to make sure that penalty term constraint will be
            if negSum > self.penalty_term:
                alpha = np.true_divide(alpha, negSum/float(self.penalty_term))
```

```
                negSum = np.sum(alpha)
            # Next we generate the semirandom alpha values for positive labels
            pos_range = np.sort(np.random.rand(np.size(p_inds))*negSum)
            for i,val in enumerate(p_inds):
                if i == 0 and np.size(p_inds) == 1:
                    alpha[val] = negSum
                elif i == 0: # First entry
                    alpha[val] = pos_range[i]
                elif i == np.size(p_inds)-1: # Last entry
                    alpha[val] = negSum-pos_range[i-1]
                else:
                    alpha[val] = pos_range[i]-pos_range[i-1]
            # Final check, are constrains satisfied?
            if (np.size(np.where(alpha > self.penalty_term)) != 0) or (np.abs(np.sum(
                # Initial aplha does not satisfy constraints! Find another one
                suitable_alpha_found = False
            else:
                suitable_alpha_found = True
        return alpha

################################################################################
################################################################################
################################################################################
# *****
# *****
# ***** The relevant code regarding the tutorial ends here. The rest of the code in t
# *****
# *****
################################################################################
################################################################################
################################################################################

    ##########################################
    #
    # DESCRIPTION:
    # - Function for visualizing the SVM hyperplane
    # in linear kernel case.
    #
    ##########################################
    def plot2D(self, f=None, ax=None):
        data = self.input_X
        labels = self.output_y
        w = self.w
        b = self.b
        alpha = self.alpha
        wnorm = float(np.linalg.norm(w))
        margin = 1/wnorm # Margin of the SVM model
        # Next, we create the boundaries for the plot.
        minX = np.min(data[:,0])
        maxX = np.max(data[:,0])
        minY = np.min(data[:,1])
        maxY = np.max(data[:,1])
        if f is None and ax is None:
            f,ax = plt.subplots(1, figsize=(10,10))
        # Next, we draw the classification areas. We need to check the borders
        marginCoefficient = 1.2
        xborder = [minX-margin*marginCoefficient, maxX + margin*marginCoefficient]
        yborder = [minY-margin*marginCoefficient, maxY + margin*marginCoefficient]
        lineYvaluesAtXborder = [(-b-w[0]*xborder[0])/float(w[1]), (-b-w[0]*xborder[1]
        lineXvaluesAtYborder = [(-b-w[1]*yborder[0])/float(w[0]), (-b-w[1]*yborder[1]
        drawVals = self.getLineAndFillAreas(xborder, yborder, lineXvaluesAtYborder, l
```

```
        xvalues = drawVals[0]
        yvalues = drawVals[1]
        xFillArea1 = drawVals[2]
        yFillArea1 = drawVals[3]
        xFillArea2 = drawVals[4]
        yFillArea2 = drawVals[5]
        xPos, yPos = [], []
        xNeg, yNeg = [], []
        if xFillArea1[-1]*w[0] + yFillArea1[-1]*w[1] > -b:
            xPos = xFillArea1
            yPos = yFillArea1
            xNeg = xFillArea2
            yNeg = yFillArea2
        else:
            xPos = xFillArea2
            yPos = yFillArea2
            xNeg = xFillArea1
            yNeg = yFillArea1
        ax.plot(xvalues, yvalues, 'k-', label="SVM model")
        ba = margin*(w/wnorm)
        xval1, yval1 = [xvalues+ba[0]], [yvalues+ba[1]]
        xval2, yval2 = [xvalues-ba[0]], [yvalues-ba[1]]
        yv1 = self.getLineYvaluesAtXborder(xval1[0], yval1[0], xborder)
        xv1 = self.getLineXvaluesAtYborder(xval1[0], yval1[0], yborder)
        yv2 = self.getLineYvaluesAtXborder(xval2[0], yval2[0], xborder)
        xv2 = self.getLineXvaluesAtYborder(xval2[0], yval2[0], yborder)
        drawVals1 = self.getLineAndFillAreas(xborder, yborder, xv1, yv1)
        xvalues1 = drawVals1[0]
        yvalues1 = drawVals1[1]
        drawVals2 = self.getLineAndFillAreas(xborder, yborder, xv2, yv2)
        xvalues2 = drawVals2[0]
        yvalues2 = drawVals2[1]
        if w[1] == 0: # In this case the line is in 90 degree angle directly updwards
            xvalues1 = xval1[0]
            xvalues2 = xval2[0]
            yvalues1 = yval1[0]
            yvalues2 = yval2[0]
        ax.plot(xvalues1, yvalues1, 'k--')
        ax.plot(xvalues2, yvalues2, 'k--')
        ax.fill(xPos, yPos, color = [1, 0, 0], alpha=0.2)
        ax.fill(xNeg, yNeg, color = [0, 0, 1], alpha=0.2)
        marginVals = self.getMarginArea(xvalues1, yvalues1, xvalues2, yvalues2, xbord
        ax.fill(marginVals[0], marginVals[1], color = [1, .7, 0], alpha=1, label="mar
        markerSizeCircle = 10
        markerSizeCross = 10
        p1 = None;
        p2 = None;
        firstP = True
        firstN = True
        # Plot all the data points.
        for i in range(0, data.shape[0]):
            if labels[0,i] > 0:
                if firstP:
                    ax.plot(data[i,0], data[i,1], "o", color=[.8, 0, 0], markersize=m
                    firstP = False
                else:
                    ax.plot(data[i,0], data[i,1], "o", color=[.8, 0, 0], markersize=m
            else:
                if firstN:
                    ax.plot(data[i,0], data[i,1], "o", color=[0, 0, .8], markersize=m
                    firstN = False
```

```
                else:
                    ax.plot(data[i,0], data[i,1], "o", color=[0, 0, .8], markersize=m

            circle1 = plt.Circle((data[i,0], data[i,1]), margin, color=[.3,.3,.3], cl
            ax.add_patch(circle1)
        # Plot support vectors
        sv_inds = np.where(alpha > 0)[0]
        ax.plot(data[sv_inds,0], data[sv_inds,1], "*", color=[0,.9,0], markersize=mar
        ax.axis('equal')
        str1 = None
        str2 = None
        if w[1] < 0:
            str1 = str(np.round(w[1],4))
        else:
            str1 = "+" + str(np.round(w[1],4))
        if b < 0:
            str2 = str(np.round(b,4))
        else:
            str2 = "+" + str(np.round(b,4))
        ax.set_title("SVM model: g(x,y) = sgn(" + str(np.round(w[0],4)) + "x" + str1
        ax.set_xlabel("X-value", fontweight='bold')
        ax.set_ylabel("Y-value", fontweight='bold')
        ax.legend()
        # Show the plot if plotMaker is not used
        if f is None and ax is None:
            plt.show()

    ########################################
    #
    # DESCRIPTION:
    # - Function for calculating the relevant
    # coordinates for plotting the margin
    # in linear kernel case
    #
    ########################################
    def getMarginArea(self, xv1, yv1, xv2, yv2, xborder, yborder):
        marginAreaX = []
        marginAreaY = []
        if (yv1[0]==yv2[0] and yv1[1]==yv2[1]) or (xv1[0]==xv2[0] and xv1[1]==xv2[1])
            #print("Margin case 1 or 2")
            marginAreaX = [xv1[0], xv1[1], xv2[1], xv2[0]]
            marginAreaY = [yv1[0], yv1[1], yv2[1], yv2[0]]
        elif yv1[0]==yv2[0] and yv1[0] > yv1[1]: # Slope negative, case 3
            #print("Margin case 3")
            marginAreaX = [xv1[0], xv1[1], xborder[1], xv2[1], xv2[0]]
            marginAreaY = [yv1[0], yv1[1], yborder[0], yv2[1], yv2[0]]
        elif yv1[0]==yv2[0] and yv1[0] < yv1[1]: # positive, case 4
            #print("Margin case 4")
            marginAreaX = [xv1[0], xv1[1], xborder[1], xv2[1], xv2[0]]
            marginAreaY = [yv1[0], yv1[1], yborder[1], yv2[1], yv2[0]]
        elif yv1[1]==yv2[1] and yv1[0] < yv1[1]: # positive, case 5, left corner down
            #print("Margin case 5")
            marginAreaX = [xv1[0], xv1[1], xv2[1], xv2[0], xborder[0]]
            marginAreaY = [yv1[0], yv1[1], yv2[1], yv2[0], yborder[0]]
        elif yv1[1]==yv2[1] and yv1[0] > yv1[1]: # negative, case 6, left corner up
            #print("Margin case 6")
            marginAreaX = [xv1[0], xv1[1], xv2[1], xv2[0], xborder[0]]
            marginAreaY = [yv1[0], yv1[1], yv2[1], yv2[0], yborder[1]]
        elif xv1[1]==xv2[1] and yv1[0] > yv1[1]: # negative, case 7, left corner up
            #print("Margin case 7")
            marginAreaX = [xv1[0], xv1[1], xv2[1], xv2[0], xborder[0]]
```

```
            marginAreaY = [yv1[0], yv1[1], yv2[1], yv2[0], yborder[1]]
        elif xv1[1]==xv2[1] and yv1[0] < yv1[1]: # negative, case 8, left corner up
            #print("Margin case 8")
            marginAreaX = [xv1[0], xv1[1], xv2[1], xv2[0], xborder[0]]
            marginAreaY = [yv1[0], yv1[1], yv2[1], yv2[0], yborder[0]]
        elif xv1[0]==xv2[0] and yv1[0] < yv1[1]: # negative, case 8, left corner up
            #print("Margin case 9")
            marginAreaX = [xv1[0], xv1[1], xborder[1], xv2[1], xv2[0]]
            marginAreaY = [yv1[0], yv1[1], yborder[1], yv2[1], yv2[0]]
        elif xv1[0]==xv2[0] and yv1[0] > yv1[1]: # negative, case 8, left corner up
            #print("Margin case 10")
            marginAreaX = [xv1[0], xv1[1], xborder[1], xv2[1], xv2[0]]
            marginAreaY = [yv1[0], yv1[1], yborder[0], yv2[1], yv2[0]]
        elif yv1[0] < yv1[1]: #positive slope, case 8, left corner up
            #print("Margin case 11")
            marginAreaX = [xv1[0], xv1[1], xborder[1], xv2[1], xv2[0], xborder[0]]
            marginAreaY = [yv1[0], yv1[1], yborder[1], yv2[1], yv2[0], yborder[0]]
        elif yv1[0] > yv1[1]: #positive slope, case 8, left corner up
            #print("Margin case 12")
            marginAreaX = [xv1[0], xv1[1], xborder[1], xv2[1], xv2[0], xborder[0]]
            marginAreaY = [yv1[0], yv1[1], yborder[0], yv2[1], yv2[0], yborder[1]]
        return(marginAreaX, marginAreaY)

    ##########################################
    #
    # DESCRIPTION:
    # - Function for calculating hyperplane
    # x-values at plot y-border
    #
    ##########################################
    def getLineXvaluesAtYborder(self, xvals, yvals, yborder):
        slope = (yvals[1]-yvals[0]) / (xvals[1]-xvals[0])
        bias = yvals[0]-slope*xvals[0]
        x = lambda y: (y-bias)/float(slope)
        return [x(yborder[0]), x(yborder[1])]

    ##########################################
    #
    # DESCRIPTION:
    # - Function for calculating hyperplane
    # y-values at plot x-border
    #
    ##########################################
    def getLineYvaluesAtXborder(self, xvals, yvals, xborder):
        slope = (yvals[1]-yvals[0]) / (xvals[1]-xvals[0])
        bias = yvals[0]-slope*xvals[0]
        y = lambda x: slope*x + bias
        return [y(xborder[0]), y(xborder[1])]

    ##########################################
    #
    # DESCRIPTION:
    # - Function for calculating hyperplane
    # decision boundary areas
    #
    ##########################################
    def getLineAndFillAreas(self, xborder, yborder, lineXvaluesAtYborder, lineYvalues
        xFillArea1 = []
        yFillArea1 = []
        xFillArea2 = []
        yFillArea2 = []
```

```
        xvalues = []
        yvalues = []
        if np.min(lineYvaluesAtXborder) < yborder[0] and np.max(lineYvaluesAtXborder)
            minXind = np.where(lineXvaluesAtYborder==np.min(lineXvaluesAtYborder))[0]
            maxXind = [i for i in [0, 1] if i not in [minXind]][0]
            xvalues = [lineXvaluesAtYborder[minXind], lineXvaluesAtYborder[maxXind]]
            yvalues = [yborder[minXind], yborder[maxXind]]
            xFillArea1, yFillArea1 = xvalues[:], yvalues[:]
            xFillArea2, yFillArea2 = xvalues[:], yvalues[:]
            if yvalues[1] > yvalues[0]: # Positive slope
                #print("CASE 1")
                xFillArea1.extend([xborder[0], xborder[0]])
                xFillArea2.extend([xborder[1], xborder[1]])
                yFillArea1.extend([yborder[1], yborder[0]])
                yFillArea2.extend([yborder[1], yborder[0]])
            else: # Negative slope
                #print("CASE 2")
                xFillArea1.extend([xborder[0], xborder[0]])
                xFillArea2.extend([xborder[1], xborder[1]])
                yFillArea1.extend([yborder[0], yborder[1]])
                yFillArea2.extend([yborder[0], yborder[1]])
        elif np.min(lineYvaluesAtXborder) > yborder[0] and np.max(lineYvaluesAtXborde
            xvalues = xborder[:]
            yvalues = lineYvaluesAtXborder[:]
            xFillArea1, yFillArea1 = xvalues[:], yvalues[:]
            xFillArea2, yFillArea2 = xvalues[:], yvalues[:]
            if yvalues[1] > yvalues[0]: # Positive slope
                #print("CASE 3")
                xFillArea1.extend([xborder[1], xborder[0]])
                xFillArea2.extend([xborder[1], xborder[0]])
                yFillArea1.extend([yborder[1], yborder[1]])
                yFillArea2.extend([yborder[0], yborder[0]])
            else: # Negative slope
                #print("CASE 4")
                xFillArea1.extend([xborder[1], xborder[0]])
                xFillArea2.extend([xborder[1], xborder[0]])
                yFillArea1.extend([yborder[1], yborder[1]])
                yFillArea2.extend([yborder[0], yborder[0]])
        elif (lineYvaluesAtXborder[0] > yborder[0] and lineYvaluesAtXborder[0] < ybor
            xvalues = [xborder[0], lineXvaluesAtYborder[1]]
            yvalues = [lineYvaluesAtXborder[0], yborder[1]]
            xFillArea1, yFillArea1 = xvalues[:], yvalues[:]
            xFillArea2, yFillArea2 = xvalues[:], yvalues[:]
            #print("CASE 5")
            xFillArea1.extend([xborder[0]])
            xFillArea2.extend([xborder[1], xborder[1], xborder[0]])
            yFillArea1.extend([yborder[1]])
            yFillArea2.extend([yborder[1], yborder[0], yborder[0]])
        elif (lineYvaluesAtXborder[0] > yborder[0] and lineYvaluesAtXborder[0] < ybor
            xvalues = [xborder[0], lineXvaluesAtYborder[0]]
            yvalues = [lineYvaluesAtXborder[0], yborder[0]]
            xFillArea1, yFillArea1 = xvalues[:], yvalues[:]
            xFillArea2, yFillArea2 = xvalues[:], yvalues[:]
            #print("CASE 6")
            xFillArea1.extend([xborder[1], xborder[1], xborder[0]])
            xFillArea2.extend([xborder[0]])
            yFillArea1.extend([yborder[0], yborder[1], yborder[1]])
            yFillArea2.extend([yborder[0]])
        elif lineYvaluesAtXborder[0] < yborder[0] and lineYvaluesAtXborder[1] < ybord
            xvalues = [lineXvaluesAtYborder[0], xborder[1]]
            yvalues = [yborder[0], lineYvaluesAtXborder[1]]
```

```
            xFillArea1, yFillArea1 = xvalues[:], yvalues[:]
            xFillArea2, yFillArea2 = xvalues[:], yvalues[:]
            #print("CASE 7")
            xFillArea1.extend([xborder[1], xborder[0], xborder[0]])
            xFillArea2.extend([xborder[1]])
            yFillArea1.extend([yborder[1], yborder[1], yborder[0]])
            yFillArea2.extend([yborder[0]])
        elif lineYvaluesAtXborder[0] > yborder[1] and lineYvaluesAtXborder[1] < ybord
            xvalues = [lineXvaluesAtYborder[1], xborder[1]]
            yvalues = [yborder[1], lineYvaluesAtXborder[1]]
            xFillArea1, yFillArea1 = xvalues[:], yvalues[:]
            xFillArea2, yFillArea2 = xvalues[:], yvalues[:]
            #print("CASE 8")
            xFillArea1.extend([xborder[1]])
            xFillArea2.extend([xborder[1], xborder[0], xborder[0]])
            yFillArea1.extend([yborder[1]])
            yFillArea2.extend([yborder[0], yborder[0], yborder[1]])
        return(xvalues, yvalues, xFillArea1, yFillArea1, xFillArea2, yFillArea2)

    ##########################################
    #
    # DESCRIPTION:
    # - Returns the signal value of trained SVM
    # model for a given input data point.
    #
    ##########################################
    def evaluateFunction(self, input_point):
        alpha = self.alpha
        X = self.input_X
        y = self.output_y
        b = self.b
        supvec_alpha_inds = np.where(alpha > 0)[0]
        kernel_function = None
        kwargs = self.kwargs
        if self.kernel == "linear":
            kernel_function = lambda x, y : np.dot(x,y)
        elif self.kernel == "Gaussian":
            kernel_function = lambda x, y : np.exp(-np.dot(x-y,x-y)/(2*kwargs["sigma"
        elif self.kernel == "polynomial":
            kernel_function = lambda x, y : (kwargs["lambda"] + kwargs["gamma"]*np.do
        signal_value = 0
        for i in range(0, len(supvec_alpha_inds)):
            svi = supvec_alpha_inds[i]
            signal_value += y[0,svi]*alpha[svi]*kernel_function(X[svi,:], input_point
        return signal_value


    ##########################################
    #
    # DESCRIPTION:
    # - Function for visualizing the SVM hyperplane
    # in non-linear kernel case.
    #
    # Code not so beautiful, I will make this
    # more elegant in the future.
    #
    ##########################################
    def plot2DNK(self):
        data = self.input_X
        minX = np.min(data[:,0])
        maxX = np.max(data[:,0])
```

```
        minY = np.min(data[:,1])
        maxY = np.max(data[:,1])
        # Make the meshgrid that we calculate
        nx, ny = (500, 500)
        x = np.linspace(minX-10, maxX+10, nx)
        y = np.linspace(minY-10, maxY+10, ny)
        xv, yv = np.meshgrid(x, y)
        yv = np.flipud(yv)
        zv = np.zeros(xv.shape)
        resultPic = np.zeros((xv.shape[0], xv.shape[1], 3))
        for xi in range(0, xv.shape[1]):
            for yi in range(0, yv.shape[0]):
                zv[yi,xi] = self.evaluateFunction([xv[yi,xi], yv[yi,xi]])
        minZ = np.min(zv[:])
        int_length = np.max(zv[:])-minZ
        zv = (2*(zv-np.min(zv[:]))-int_length)/float(int_length)
        ele_azi = np.array([[90,0], [45, 0], [45, 45], [45,90], [45, 135], [45, 180]]
        fig = plt.figure(figsize=(20,30))
        for j in range(0, ele_azi.shape[0]):
            ax = fig.add_subplot(3,2, j+1, projection='3d')
            sv_inds = np.where(self.alpha > 0)[0]
            im = ax.plot_surface(xv, yv, zv, alpha=.9, cmap=cm.seismic,linewidth=0, a
            firstPos = True
            firstNeg = True
            for i in range(0, data.shape[0]):
                z= (2*(self.evaluateFunction([data[i,0], data[i,1]])-minZ)-int_length
                if self.output_y[0,i] > 0:
                    z += .3
                    if ele_azi[j,0] == 90: # Top view
                        z = 1
                    if firstPos:
                        ax.scatter3D(data[i,0], data[i,1], z, s=100,c=[[255/255.0, 0/
                        firstPos = False
                    else:
                        ax.scatter3D(data[i,0], data[i,1], z, s=100,c=[[255/255.0, 0/
                else:
                    z -= .3
                    if ele_azi[j,0] == 90: # Top view
                        z = 1
                    if firstNeg:
                        ax.scatter3D(data[i,0], data[i,1], z, s=100,c=[[0/255.0, 255/
                        firstNeg = False
                    else:
                        ax.scatter3D(data[i,0], data[i,1], z, s=100,c=[[0/255.0, 255/
            firstSV = True
            for i in range(0, len(sv_inds)):
                z= (2*(self.evaluateFunction([data[sv_inds[i],0], data[sv_inds[i],1]]
                #if z > 0:
                if self.output_y[0,sv_inds[i]] > 0:
                    z += 0.3
                else:
                    z -= 0.3
                if ele_azi[j,0] == 90: # Top view
                    z = 1
                if firstSV:
                    ax.scatter3D(data[sv_inds[i],0], data[sv_inds[i],1], z, s=80, mar
                    firstSV = False
                else:
                    ax.scatter3D(data[sv_inds[i],0], data[sv_inds[i],1], z, s=80, mar
            ax.view_init(elev=ele_azi[j,0], azim=ele_azi[j,1])
            ax.set_xlabel("X")
```

```
            ax.set_ylabel("Y")
            ax.set_zlabel("Z")
            ax.zaxis.set_major_locator(LinearLocator(10))
            ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))
            # Add a color bar which maps values to colors.
            fig.colorbar(im, shrink=1.5, aspect=5)
            ax.set_zticks([])
            if self.kernel == "polynomial":
                ax.set_title("$K(x,x') = \\left(" + str(self.kwargs["lambda"]) + " +
            elif self.kernel == "Gaussian":
                ax.set_title("$K(x,x') = \\exp\\left(-\\frac{||x-x'||^2}{" + str(2*(s
            ax.legend()

    ##########################################
    #
    # DESCRIPTION:
    # - Function for plotting the search path
    # of the alpha parameter in GPAS-method
    # from multiple different angles.
    # The constraint plane y^Ta = 0 is also shown.
    #
    # - Since the number of data points is
    # directly in connection to the number of
    # alpha parameters this function only works
    # with three data points.
    #
    ##########################################
    def plotAlphaCurve(self):
        normal = self.output_y
        nx, ny = (100, 100)
        C = self.penalty_term
        x = np.linspace(0, C, nx)
        y = np.linspace(0, C, ny)
        xv, yv = np.meshgrid(x, y)
        zv = (-normal[0,0] * xv - normal[0,1] * yv) * 1. /normal[0,2]
        # plot the surface
        ele_azi = np.array([[90,0], [45, 45], [45, 90], [45,135], [0,45], [0,90]])
        fig = plt.figure(figsize=(20,30))
        for j in range(0, ele_azi.shape[0]):
            ax = fig.add_subplot(3,2, j+1, projection='3d')
            im = ax.plot_surface(xv, yv, zv, alpha=0.2, cmap ="brg", rstride=1, cstri
            im._facecolors2d=im._facecolors3d
            im._edgecolors2d=im._edgecolors3d
            fig = plt.gcf()
            fig.set_size_inches(18.5, 10.5)
            ax.view_init(elev=40, azim=45)
            l1 = ax.set_xlabel("$\\alpha_1$", fontsize=15)
            l2 = ax.set_ylabel("$\\alpha_2$", fontsize=15)
            l3 = ax.set_zlabel("$\\alpha_3$", fontsize=15)
            l1.set_color("blue")
            l2.set_color("blue")
            l3.set_color("blue")
            ax.plot([C,C],[C,0],[0,0], c="blue", label="Box constraints")
            ax.plot([0,0],[C,0],[0,0], c="blue")
            ax.plot([C,C],[C,0],[C,C], c="blue")
            ax.plot([0,0],[C,0],[C,C], c="blue")
            ax.plot([0,C],[C,C],[0,0], c="blue")
            ax.plot([0,C],[C,C],[C,C], c="blue")
            ax.plot([C,C],[C,0],[C,C], c="blue")
            ax.plot([0,0],[C,0],[C,C], c="blue")
            ax.plot([0,C],[C,C],[0,0], c="blue")
```

```
            ax.plot([0,C],[C,C],[C,C], c="blue")
            ax.plot([0,C],[0,0],[0,0], c="blue")
            ax.plot([0,C],[0,0],[C,C], c="blue")
            ax.plot([C,C],[C,C],[0,C], c="blue")
            ax.plot([0,0],[C,C],[0,C], c="blue")
            ax.plot([0,0],[0,0],[0,C], c="blue")
            ax.plot([C,C],[0,0],[0,C], c="blue")
            ax.view_init(elev=ele_azi[j,0], azim=ele_azi[j,1])
            alphas = self.alphas
            c = None
            s = 100
            label=None
            first = True
            for i in range(0, 10):
                alpha = alphas[i]
                #print(alpha)
                if i == 0:
                    c = "blue"
                    label = "Starting $\\alpha$"
                elif i == 9:
                    c = "green"
                    s = 200
                    label = "Optimal $\\alpha$"
                else:
                    c = "red"
                    if first:
                        label = None
                        first = False
                    else:
                        label = None
                ax.scatter(alpha[0], alpha[1], alpha[2], s=s, marker="o", c=c, edgeco
                if i > 0:
                    ax.plot([alpha[0][0], old_alpha[0][0]], [alpha[1][0], old_alpha[1
                old_alpha = alpha[:]
            fig.colorbar(im)
            ax.zaxis.set_major_locator(LinearLocator(10))
            ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))
            ax.set_title("$0 \leq \\alpha \leq " + str(C) + "$")
            if j == 1:
                ax.legend()

    ##########################################
    #
    # DESCRIPTION:
    # - Function for plotting the error curve
    # visualizing how the GPAS-method minimizes
    # the objective function in eq. (31).
    #
    ##########################################
    def plotErrorCurve(self):
        errors = self.errors
        endInd = 0
        for i in range(1, len(errors)):
            diffVal = np.abs(errors[i-1]-errors[i])
            if diffVal < 10**-6:
                break
            endInd += 1
        errors = errors[0:endInd]
        f,ax = plt.subplots(1, figsize=(10,5))
        ax.plot(range(endInd), np.reshape(errors, (len(errors),)), label="Objective f
        ax.plot(range(endInd), np.reshape(errors, (len(errors),)), "o")
```

```
        ax.grid("on")
        ax.set_ylabel("Objective function value")
        ax.set_xlabel("Iteration number")
        ax.legend()

########################################
#
# DESCRIPTION:
# - Function for plotting multiple random
# linear SVM plots
#
########################################
def plotMultipleSVMs(rows, cols):
    f, ax = plt.subplots(rows, cols, figsize=(20,10))
    ind = 0
    value_selection_interval = [-20,20]
    number_of_positive_samples = 5
    number_of_negative_samples = 5
    X_feature_dimensions = 2
    error_tolerance = np.finfo(float).eps
    kwargs = {"sigma":1, "lambda":0, "gamma":1, "q":1, "printInfo":False}
    for r in range(0, rows):
        for c in range(0, cols):
            (X,y) = createRandomDataSetLinear(X_feature_dimensions, number_of_positiv
            SVM_model = SVM(X, y, "linear", iterations, error_tolerance, np.inf, kwar
            SVM_model.train()
            if rows == 1 and cols == 1:
                SVM_model.plot2D(f, ax)
            elif rows == 1:
                SVM_model.plot2D(f, ax[c])
            elif cols == 1:
                SVM_model.plot2D(f, ax[r])
            else:
                SVM_model.plot2D(f, ax[r,c])
            ind += 1
    plt.show()
```

In [2]:

```
import matplotlib.pyplot as plt
%matplotlib inline
#############################################################
#
# - Used to plot the example figures in introduction
#
#############################################################
def drawClassifierLines(showRadius):
    # Four data points from two different classes.
    x = [0, 2, 2, 3]
    y = [0, 2, 0, 0]
    # Three classifier lines.
    lines = [[-1, 4, -2, 2.2], [-1, 3.4, -2, 3], [-1, 4, -2, 3]]
    # Three subplots.
    f, axarr = plt.subplots(1, 3, figsize=(20,5))
    # Set plot limits.
    xlim = [-2, 4]
    ylim = [-2, 3]
    # Plot all data points.
    for i in range(0, len(axarr)):
        axarr[i].plot(x[0:2], y[0:2], "o", color=[0, 0, .8], markersize=15, markeredgew
        axarr[i].plot(x[2:4], y[2:4], "o", color=[.8, 0, 0], markersize=15, markeredgew
        axarr[i].set_xlim(xlim)
        axarr[i].set_ylim(ylim)
        axarr[i].plot(lines[i][0:2], lines[i][2:4], 'k-')
        axarr[i].set_xticks([])
        axarr[i].set_yticks([])
    # Draw uncertainty circles.
    if showRadius:
        rad = [0.4, 0.4, 1/np.sqrt(2)]
        for j in range(0, len(axarr)):
            for i in range(0, len(x)):
                circle1 = plt.Circle((x[i], y[i]), rad[j], color=[.3,.3,.3], clip_on=Fa
                axarr[j].add_patch(circle1)
    # Plot the hard coded decision areas.
    axarr[0].fill([-1, -2, -2, 4, 4, -1], [-2, -2, 3, 3, 2.2, -2], color = [0, 0, 1], a
    axarr[0].fill([-1, 4, 4, -1], [-2, 2.2, -2, -2], color = [1, 0, 0], alpha=0.2)
    axarr[1].fill([-1, -2, -2, 3.4, -1], [-2, -2, 3, 3, -2], color = [0, 0, 1], alpha=0
    axarr[1].fill([-1, 3.4, 4, 4, -1], [-2, 3, 3, -2, -2], color = [1, 0, 0], alpha=0.1
    axarr[2].fill([-1, -2, -2, 4, -1], [-2, -2, 3, 3, -2], color = [0, 0, 1], alpha=0.1
    axarr[2].fill([-1, 4, 4, -1], [-2, 3, -2, -2], color = [1, 0, 0], alpha=0.2)
    # Show plot
    plt.show()

#############################################################
#
# - Used to plot the example figures in introduction
#
#############################################################
def drawNonSeparableCases():
    # Make the custom data, first plot two data clusters with two outliers
    neg_x = [-1, 0, -.5, 0.5, 2.5]
    neg_y = [1, 1.5, 0, 2, -1]
    pos_x = [1, 2, 3, 1.2, -.8]
    pos_y = [-1, -.3, 0, -.2, 2]
    f, axarr = plt.subplots(1, 2, figsize=(20,5))
    # Set plot limits.
    xlim = [-2, 4]
    ylim = [-2, 3]
```

```
    line = [-2, 4, -2, 3]
    # Plot all data points.
    axarr[0].plot(neg_x, neg_y, "o", color=[0, 0, .8], markersize=15, markeredgewidth=
    axarr[0].plot(pos_x, pos_y, "o", color=[.8, 0, 0], markersize=15, markeredgewidth=
    axarr[0].set_xlim(xlim)
    axarr[0].set_ylim(ylim)
    axarr[0].plot(line[0:2], line[2:4], 'k-')
    # Plot the hard coded decision areas.
    axarr[0].fill([-2, -2, 4], [3, -2, 3], color = [0, 0, 1], alpha=0.2)
    axarr[0].fill([-2, 4, 4, -2], [-2, -2, 3, -2], color = [1, 0, 0], alpha=0.2)
    axarr[0].set_xticks([])
    axarr[0].set_yticks([])
    # Second, draw an example of radial nature. That is, data is separated by a circle
    axarr[1].plot([1], [0.6], "o", markersize=175, markerfacecolor=[255/255.0, 204/255
         markeredgecolor='black', markeredgewidth=3)
    axarr[1].plot([1], [0.6], "o", markersize=175, markerfacecolor="None",
         markeredgecolor='black', markeredgewidth=3)
    axarr[1].fill([-2, -2, 4, 4], [-2, 3, 3, -2], color = [0, 0, 1], alpha=0.2)
    neg_x = [-1, -.3, -.9, 0, 2.5, 3, 3.2, 1, 1.8, 0.2]
    neg_y = [1, 1.5, 0, 2.2, -1, 1, 2, -1.5, 2.3, -1]
    pos_x = [1, 0.5, 0.6, 1.5, 1.2, 1]
    pos_y = [1, 0, 1.2, 0, 1.4, -0.3]
    axarr[1].plot(neg_x, neg_y, "o", color=[0, 0, .8], markersize=15, markeredgewidth=
    axarr[1].plot(pos_x, pos_y, "o", color=[.8, 0, 0], markersize=15, markeredgewidth=
    axarr[1].set_xlim(xlim)
    axarr[1].set_ylim(ylim)
    axarr[1].set_xticks([])
    axarr[1].set_yticks([])
    # Show plots
    plt.show()

```

7. References

Y.S. Abu-Mostafa, M. Magdon-Ismail, H.-T Lin. Learning From Data, California Institute of Technology, 2012.

S. Boyd, L. Vandenberghe. Convex Optimization, Cambridge University Press, 2009.

D.G. Luenberger, Y. Ye. Linear and Nonlinear programming, Springer, 3rd edition, 2008.

V.N. Vapnik. Statistical Learning Theory, Jon Wiley & Sons Inc., 1998.