Support Vector Machines

Seminar Data Mining

Leon Zamel
Department of Informatics
Technical University of Munich
Email: leon.zamel@tum.de

***Abstract*—In this paper we will cover the internal workings and mathematical aspects behind the Support Vector Machine (SVM). We will see how constructing a hyperplane allows us to classify data points depending on which side they lie on and how to construct an optimal hyperplane which generalizes well by maximizing the margin. To do so, we will introduce an optimization problem and will solve it by using Lagrange multipliers to formulate the dual problem. We will then briefly look at how to expand this simple model to also work with non-linearly separable data via slack variables and the kernel method. Lastly, we will be comparing the one-against-the-rest and one-against-one multi-class methods and will see that one-against-one is popular for real world usage, since its training time is lower and both methods yield similar results.**

***Keywords*—Data Mining, SVM, multi-class, slack variables, kernel method, Lagrange multipliers**

I. Introduction

Machine learning has become a very important topic in recent years with the rise of Big Data and the Internet of Things. With more information to analyze than ever before, it is necessary for machines to take care of some of this work for us. One interesting task to solve is the classification of data. Classification means assigning labels to things. Deciding if a given credit card transaction is likely to be fraudulent or deciding which digit is present in a given image are just two examples of the many classification problems, we may face in the real world which can be solved with Support Vector Machines (SVMs) [1], [2]. The SVM uses training data for which we already know the correct class and builds a model from this to make decisions on its own. By treating the pre-labeled learning samples as points in space, it will construct a so-called hyperplane, which acts as a boundary to separate these points. It can then make predictions on new unseen data, based on where this new data point lies with respect to the boundary.

Throughout this paper we will see how the SVM works for classification. Step by step we will build up a visual understanding of how to optimally categorize data points and will translate these concepts to mathematical formulas. Starting with the modeling of our data and the hyperplane, we will then find out how to find the best hyperplane for the data. We will formulate this as an optimization problem, which we will solve using Lagrange multipliers and quadratic programming. We will then generalize the model to work with non-linearly separable data via the kernel method and will introduce slack variables to deal with statistical outliers. Lastly, we look at different ways to use SVMs for multi-class classification.
Throughout this paper, we will mainly work along two pieces of literature which are also the basis for most of the formulas and offer additional information for the interested reader [2], [3].

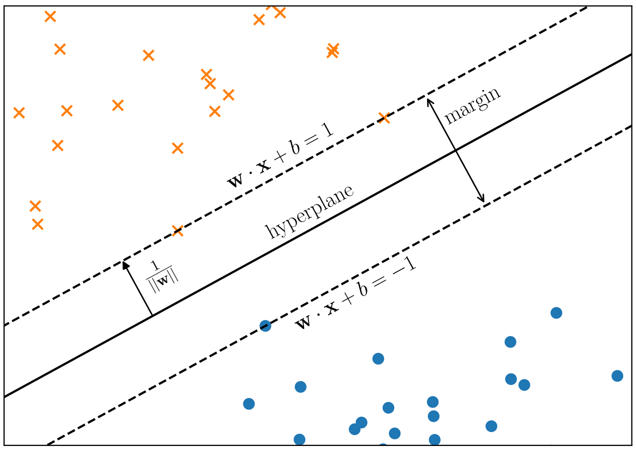

$\mathbf{w}\cdot\mathbf{x}+b=1$
margin
hyperplane
$\frac{1}{\|\mathbf{w}\|}$
$\mathbf{w}\cdot\mathbf{x}+b=-1$

Fig. 1: A hyperplane separating two classes of data. We want to maximize the distance between the dashed lines so our model generalizes well.

II. Two-class SVMs

As support vector machines lie in the area of supervised machine learning, we must first have a labeled dataset from which the SVM can construct a model to use later on for classification of new data. In its simplest form, an SVM will only discern between two classes, -1 and $+1$ will make for easy notation later. We can formulate this as the following:

$$(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), ..., (\boldsymbol{x}_n, y_n) \in \mathbb{R}^p \times \{-1, 1\} \tag{1}$$

Where $\boldsymbol{x}_i$ is a p-dimensional vector representing a data point with p features and y_i being the corresponding class. We now look to create a classifier which builds a model from this data to predict the class of new unseen data points. That is, given an unseen $\boldsymbol{x}$ it should predict a y. In the simplest form of SVMs we create one *hyperplane*, i.e., an $n-1$ dimensional object in n dimensional space, to separate these data points. When we want to predict the class of a new point, we can plot it in this space and predict which class it belongs to depending on which side of the plane it lies on. We now define our hyperplane as the points $\boldsymbol{x}$ satisfying

$$\boldsymbol{w} \cdot \boldsymbol{x} + b = 0 \tag{2}$$

that is the dot product of $\boldsymbol{w}$ and $\boldsymbol{x}$ plus a bias b, with $\boldsymbol{w}$ being the normal vector of the hyperplane. The goal is to find a $\boldsymbol{w}$ such that the hyperplane correctly separates our data points. This means that the formula

$$sgn(\boldsymbol{w} \cdot \boldsymbol{x}_i + b) = y_i \tag{3}$$

where $sgn()$ gives us the sign of the number, holds true for all data points $i \in 1, 2, ..., n$. Because we chose -1 and 1 for the labels, we get this very concise equation. We'll shortly see, that if a hyperplane satisfying these conditions exists, there exist infinitely many by slightly moving or rotating an existing plane. It is now our goal to find the 'best' one.

A. *Finding the best hyperplane*

To clearly find the best fitting hyperplane, we must first define what 'best' means in this context. For one, the hyperplane should label all our known data correctly. Also, as we want to predict the class of unseen data, we would like the model to generalize well. In intuitive terms this means that new data points which are closer to the already labeled points of class 1 than -1 should also be labeled as 1 and vice versa. We therefore look for a hyperplane which separates the data points and maximizes the distance to the closest points from both classes. This distance is also referred to as the *margin*. For now, no points will lie in this space, so we will get a *hard-margin* SVM. As a two-dimensional example, we look at Figure 1.

We see that the hyperplane correctly separates the two classes at the top and at the bottom. The two dashed lines are parallel to the hyperplane and show the margin, i.e., the distance of the hyperplane to the closest data points. The goal now is to maximize this margin which will also maximize the hyperplane's distance to the data points. We can think of the two outer boundaries as being hyperplanes themselves. We specify them via the equations:

$$\boldsymbol{w} \cdot \boldsymbol{x} + b = 1 \tag{4}$$

$$\boldsymbol{w} \cdot \boldsymbol{x} + b = -1 \tag{5}$$

The reason why we want to maximize this distance can be seen in Figure 2. Here we see the optimal hyperplane in green. The two parallel dashed green lines show the margin between the hyperplane and the closest points. In contrast to the optimal hyperplane, we also see a sub optimal hyperplane in red. The reason why this line is worse, is that it won't generalize well. Imagine we want to predict the class of a new data point, indicated by the purple star. We clearly see that it is quite close to the blue class and would probably like it to be classified as such. The optimal line classifies it correctly, the other one does not.

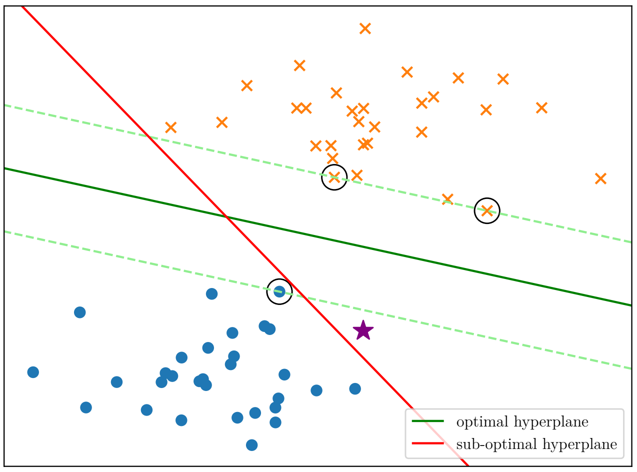


Fig. 2: A simple instance of separating two groups of data with a straight line. The optimal hyperplane maximizes the distance to both groups, while the sub-optimal hyperplane doesn't. A new data point (indicated by the purple star) will be classified differently based on which hyperplane is chosen.

To maximize the margin, we first need to calculate the distance between the two outer hyperplanes. For this, we can simply choose points $\boldsymbol{x}_+, \boldsymbol{x}_-$ from each plane respectively and project the difference onto the unit normal vector of the hyperplane to get the margin m:

$$\begin{aligned} m &= \frac{\boldsymbol{w}}{\|\boldsymbol{w}\|} \cdot (\boldsymbol{x}_+ - \boldsymbol{x}_-) \\ &= \frac{\boldsymbol{w} \cdot \boldsymbol{x}_+ - \boldsymbol{w} \cdot \boldsymbol{x}_-}{\|\boldsymbol{w}\|} \\ &= \frac{(1-b) - (-1-b)}{\|\boldsymbol{w}\|} \quad \text{(subst. 4 and 5)} \\ &= \frac{2}{\|\boldsymbol{w}\|} \end{aligned} \tag{6}$$

We may now use this simpler equation to maximize the margin.

B. *Maximizing the margin*

First, we see that maximizing the margin is purely done by minimizing $\|\boldsymbol{w}\|$ with respect to the condition that 3 still holds. We can rewrite this condition to

$$y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + b) - 1 \geq 0 \quad \text{for } i = 1, ..., n \tag{7}$$

resulting in the optimization formula

$$\begin{aligned} \min_{w,b} \quad & \frac{1}{2}\|\boldsymbol{w}\|^2 \\ \text{s.t.} \quad & y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + b) - 1 \geq 0 \quad \text{for } i = 1, ..., n \end{aligned} \tag{8}$$

We have added the factor $\frac{1}{2}$ and squared $\|\boldsymbol{w}\|$ here. This does not change where we find the minimum, however it simplifies the formula later on.

This is a constrained optimization problem for which we will use Lagrange multipliers to find a solution. First we

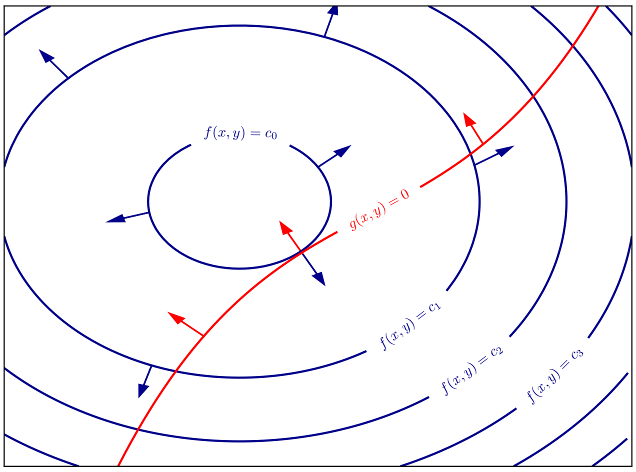


Fig. 3: An example showing the idea of Lagrange multipliers. We would like to minimize the function f with the condition that $g(x, y) = 0$ holds. Shown are some height levels for f in blue and the points where the constraint is fulfilled in red. The arrows show the gradients. The minimal value will be found where the gradients of f and g are parallel.

will understand how they work for a problem with equality constraints, i.e., satisfying a function $g(\boldsymbol{x}) = 0$, before moving on to inequality constraints of the form $g(\boldsymbol{x}) \leq 0$[1].

The idea of Lagrange multipliers is the following: imagine we have a function $f(x, y)$, which we would like to minimize given a constraint $g(x, y) = 0$, i.e., our minimum must fulfill this condition to be valid. We can imagine that we walk along the line where the condition for g holds. Then we will reach the minimum of f under that constraint where the gradients of f and g are parallel. If this wasn't the case, because the gradient always points in the direction with the steepest incline, we could walk further along one direction on g to decrease f further. In other words, we look for points where the gradients are multiples of each other:

$$\nabla_{x,y} f = -\lambda \nabla_{x,y} g \tag{9}$$

where λ is a so-called Lagrange multiplier. It is needed here so that the two vectors can have different lengths and may show in opposite or equal directions, while still keeping them parallel. This concept is shown in Figure 3. The red line shows where the constraint is met. The blue ovals are the contours for the function f at various levels. In three dimensions, f would look like a cone with its lowest point at the middle. The arrows belonging to the functions show the gradients. There is one point shown where the gradients show in exactly opposite directions. This is the minimum of f under the given constraint. It should also be noted that this method will give us necessary conditions for extrema. It works the same way for maximization problems and will thus generally find the minimum and maximum of a function, even if we only want one of these. This is not a problem, however, as we can just compare the values for f at those points and take the smaller one.

[1]Our constraints are currently of the form $g(\boldsymbol{x}) \geq 0$, so we will just multiply them with -1 later on

A more compact way of expressing our wish to find parallel gradients and points which satisfy the constraints is with the Lagrange function:

$$\mathcal{L}(x, y, \lambda) = f(x, y) + \lambda g(x, y) \tag{10}$$

We solve for the gradient of this function being equal to zero

$$\nabla_{x,y,\lambda} \mathcal{L}(x, y, \lambda) = \mathbf{0} \tag{11}$$

If we take the partial derivatives, we see that this is equal to setting the gradients of f and g equal, while also solving for $g(x, y) = 0$.

Inequality constraints work similarly, we can imagine two cases for where the minimum of f lies:

- The minimum is at the border of where the inequality is true. In this case, the constraint will be an equality constraint, so $g(x, y) = 0$ will hold. Then we know how to find the minimum via the parallel gradients. Also, $\lambda \geq 0$ must hold. The gradients will point in opposite directions.
- The minimum is somewhere where $g(x, y) < 0$. This means the point will be a local minimum of f, so $\nabla f = \mathbf{0}$. The gradients are equal $\iff \lambda = 0$.

From the two cases above we can see that $g(x, y) * \lambda = 0$ and $\lambda \geq 0$ must always be true.

The topic of Lagrange multipliers can be quite complex, but we now have an intuitive understanding as to why this method works. More information can be found in other literature [4]. We now generalize this concept to work in an arbitrary number of dimensions and use an arbitrary number of constraint functions. With $f(\boldsymbol{w}) = \|\boldsymbol{w}\|^2 = \boldsymbol{w} \cdot \boldsymbol{w}$, we formulate the Lagrange function for our optimization problem:

$$\mathcal{L}(\boldsymbol{w}, b, \boldsymbol{\Lambda}) = \frac{1}{2}\boldsymbol{w} \cdot \boldsymbol{w} - \sum_{i=1}^{n} \alpha_i [y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + b) - 1] \tag{12}$$

Where $\boldsymbol{\Lambda} = (\alpha_1, ..., \alpha_n)^T$ is a vector corresponding to the Lagrange multipliers – one for each constraint in 7. The points where the gradient is zero will be at saddle points of the function. We can see this in the above function via a different intuition of Lagrange functions. If we first maximize the above function with respect to $\boldsymbol{\Lambda} \geq \mathbf{0}$ we get

$$\max_{\boldsymbol{\Lambda}} \quad \mathcal{L}(\boldsymbol{w}, b, \boldsymbol{\Lambda}) = \begin{cases} f(\boldsymbol{w}), & \text{if all constraints are satisfied} \\ +\infty, & \text{otherwise} \end{cases} \tag{13}$$

This is because if the constraints aren't met, we can choose an arbitrarily large value for α. If they are met, then the entire sum will be zero, as we have concluded after examining both possible cases for where the minimum can lie.
If we then minimize this function with respect to $\boldsymbol{w}$ and b we

get the minimum of f with all our constraints satisfied. Which means we now want to solve:

$$\begin{aligned} &\min_{w,b} \quad \max_{\mathbf{\Lambda}} \quad \mathcal{L}(\boldsymbol{w}, b, \mathbf{\Lambda}) \\ &\text{subject to} \quad \mathbf{\Lambda} \geq \mathbf{0} \iff \alpha_i \geq 0, \quad i = 1, ..., n \end{aligned} \tag{14}$$

To obtain a minimum we will check where the partial derivatives of $\mathcal{L}$ for $\boldsymbol{w}$ and b respectively are zero:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = \boldsymbol{w} - \sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i = \mathbf{0} \tag{15}$$

$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_{i=1}^{n} \alpha_i y_i = 0 \tag{16}$$

From 15 it follows that

$$\boldsymbol{w} = \sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i \tag{17}$$

This last equation is extremely interesting, as it shows us where the SVM gets its name from. Our training data $x_1, ..., x_n$ is used as a linear combination to form the plane $\boldsymbol{w}$, i.e., they act as *support vectors* for the plane. An α_i will be non-zero exactly if it is a support vector. Then the minimum of the optimization lies directly on the boundary of that ith constraint function, which means the constraint function will have a value of zero at that point. If we recall the visuals of separating points and maximizing the margin, these training points are exactly those which will be closest to the hyperplane. Most vectors won't contribute though, they will have an $\alpha_i = 0$

We now substitute back into 12 to obtain:

$$\mathcal{L}(b, \mathbf{\Lambda}) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \boldsymbol{x}_i \cdot \boldsymbol{x}_j - b \sum_{i=1}^{n} \alpha_i y_i \tag{18}$$

Lastly, using 16 we see that the last term must be zero. It won't contribute to the final result, so we instead include it as another condition to our maximization problem. We arrive at:

$$\begin{aligned} \max_{\mathbf{\Lambda}} \quad & \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \boldsymbol{x}_i \cdot \boldsymbol{x}_j \\ \text{s.t.} \quad & \mathbf{\Lambda} \geq \mathbf{0} \\ & \sum_{i=1}^{n} \alpha_i y_i = 0 \end{aligned} \tag{19}$$

as the final optimization problem. The interesting thing is, that using the Lagrange multipliers we have arrived at a very different problem which will give us the solution to the original problem. This is known as the duality principle: our optimization problem can be viewed in different ways, and some are easier to solve than others. We also implicitly swapped the order of minimization and maximization in the former steps. Usually, the dual problem only gives us a lower bound to our minimization problem, but in this case, the Karush-Kuhn-Tucker conditions hold, which means we get *strong duality*: solving this *dual problem* gives us the same result as solving the former *primal problem*. The advantage of this form is that after rewriting it using matrices [2], it can then be solved using quadratic programming, for which there exist efficient algorithms.

We have now arrived at a point where we can solve the problem of maximizing the margin for our hyperplane. We should note that we have assumed that such a hyperplane *can* be found, i.e., our data is linearly separable. However, this is not always the case. We will now look at ways to deal with those cases.

III. Non-linearly separable data

Our system right now only works if we can linearly separate our data points. In two dimensions this means having a straight line separating the data perfectly; in three dimensions it would be a plane separating our data. In the real world we can easily think of two reasons this might not hold. One is that our data has outliers. This means there are data points which *look* like they should belong to one class, but actually don't. The other reason might be that there *is* a clear distinction between the two classes, yet the boundary can't be represented by a hyperplane. A two-dimensional example would be a circle representing one class being inside of another circle representing the other class. While both problems could be solved with just one of the following two methods, we see that each solution has a clear pattern of 'mixed' data that it will excel on.

A. Soft-margin SVM

To tackle the problem of statistical outliers, we realize that we might not actually *want* our SVM to classify this data correctly. After all, if the SVM sees such a combination of features in the future, we probably would like it to assume that everything is as normal, and the data is not a statistical outlier. This introduces the notion of *slack variables*. These allow the SVM to make some errors. Points can now lie within the margin (hence the name soft-margin SVM) and possibly even on the wrong side of the hyperplane. We therefore modify our constraints defined in 7 to:

$$\begin{aligned} y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + b) - 1 \geq -\xi_i \quad & \text{for } i = 1, ..., n \\ \xi_i \geq 0 \quad & \text{for } i = 1, ..., n \end{aligned} \tag{20}$$

One can see that we could just choose large values for ξ and our constraints will always be fulfilled. That's why we need to find a balance between acceptable errors and accuracy, we include our slack variables in our minimization problem 8:

$$\begin{aligned} \min_{w,b,\xi} \quad & \frac{1}{2}\|\boldsymbol{w}\|^2 + C \sum_{i=1}^{n} \xi_i \\ \text{s.t.} \quad & y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + b) - 1 \geq -\xi_i \quad && \text{for } i = 1, ..., n \\ & \xi_i \geq 0 && \text{for } i = 1, ..., n \end{aligned} \tag{21}$$

We have added another factor C here. This can be used later on to choose how the errors should be weighted. For example,

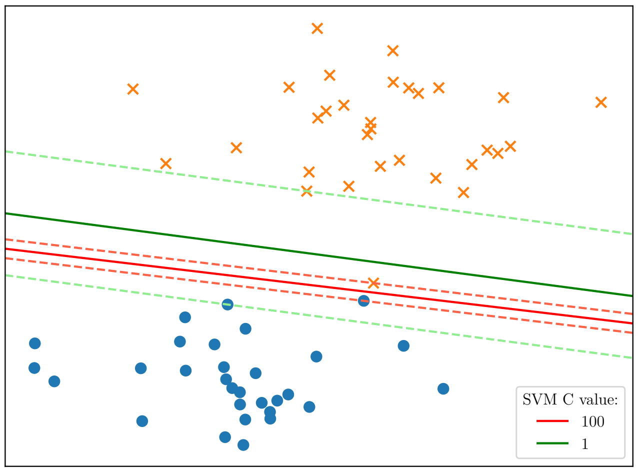


Fig. 4: Two different C values yield very different results for the hyperplane. The model resulting in the red hyperplane heavily punishes the use of slack variables, while the model with the green hyperplane balances these two values. The green hyperplane will generalize better, as it ignores statistical outliers.

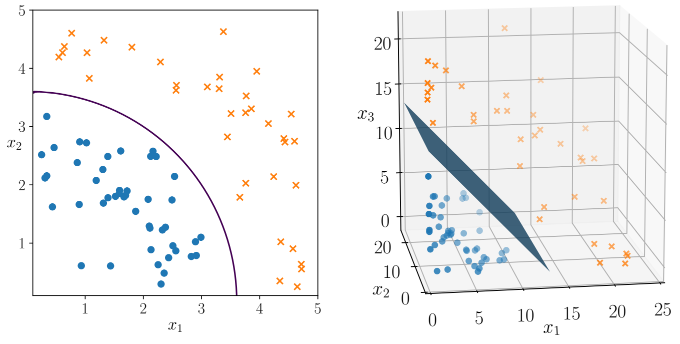
we can set it to a very large value to receive a hard-margin classifier again. We see how slack variables and the choice of C can affect the resulting hyperplane in Figure 4. Technically, this example is linearly-separable, however, it shows that slack variables are also very important for noisy data. The red hyperplane is the result of choosing a high value for C, 100. The result is, that correctly classifying every single data point is very important. Therefore, the hyperplane is squeezed just between the two points of the different classes. The green hyperplane results from a model with a more modest C value of 1. As we can see, it actually wrongly classifies one orange x, as it lies on the same side of the line as the blue dots. But arguably, the green boundary still seems better. It correctly treats the orange x as an outlier in exchange for a line with a more even distance from the groups.

The control we get through this is very valuable, as it allows us to tweak the SVM and push its decision boundary in the right direction based on our knowledge. Also, the resulting optimization problem 19 hardly changes at all. The only difference is, that the condition $\mathbf{\Lambda} \geq \mathbf{0}$ changes to $\mathbf{0} \leq \mathbf{\Lambda} \leq \mathbf{1}C$.
We now have a way to deal with statistical outliers and will now turn to the second type of non-linear problems.

B. *The Kernel method*

Maybe the data we are dealing with inherently doesn't have a linear separation. In two dimensions we could imagine a curved line separating our data. As an example, look at Figure 5a.

As we can see, in the two-dimensional space the lower-left and upper-right samples form two different classes which we can separate with a bent line, but it is not possible to separate these with a straight line. Our slack variables will also not be the right tool, they will allow us to find a straight line,



(a) Two classes of data separated by a rounded boundary. A straight line could not separate the two classes without error.

(b) The data points are transformed to a three dimensional space in which we can separate the data with a plane.

Fig. 5: Some data can only be separated once transformed to a higher dimension.

but that boundary won't be good at capturing the underlying structure of our data. Instead, we will use the *kernel method*. The main idea is to transform the data into a higher dimension and then find a hyperplane in that space. It can be shown that every dataset may be linearly separated if transformed into a high enough dimension. We therefore look for a non-linear projection $\Phi : \mathbb{R}^p \to \mathbb{R}^f, f > p$ which takes a data point $\boldsymbol{x}$ from the *input space* to a new, so called *feature space*. We then find a hyperplane in the feature space.
A simple transformation for our example is $\Phi : \mathbb{R}^2 \to \mathbb{R}^3, (x_1, x_2) \mapsto (x_1^2, \sqrt{2}x_1x_2, x_2^2)$ which has the effect of straightening out points which lie on an arc around the origin. We see the resulting transformation in Figure 5b.

Here we can clearly see that there is a plane separating our two classes of data. The plane we use can be transformed back to input space as well and forms the round outline we saw in Figure 5a where it 'cuts' the x_1, x_2 plane.
Regarding the optimization problem, this simply translates to replacing the occurrences of $\boldsymbol{x}_i \cdot \boldsymbol{x}_j$ with $\Phi(\boldsymbol{x}_i) \cdot \Phi(\boldsymbol{x}_j)$ in 19. This will work, but it isn't optimal. In the example we found a simple solution to our problem. However, the process of first transforming our space and then calculating the distance in a potentially very high dimensional space can get very complex and slow. Instead of converting our points to a new space and calculating the distance there, it sometimes is possible to skip those steps and calculate the distance in feature space directly. That is, we must find a function K such that:

$$K(\boldsymbol{x_i}, \boldsymbol{x_j}) = \Phi(\boldsymbol{x}_i) \cdot \Phi(\boldsymbol{x}_j) \quad \text{for } i, j = 1, ..., n \tag{22}$$

For our example, there exists such a function, namely:

$$\begin{aligned} K(\boldsymbol{a}, \boldsymbol{b}) &= (\boldsymbol{a} \cdot \boldsymbol{b})^2 \\ &= (a_1b_1 + a_2b_2)^2 \\ &= a_1^2b_1^2 + 2a_1b_1a_2b_2 + a_2^2b_2^2 \\ &= (a_1^2, \sqrt{2}a_1a_2, a_2^2)^T \cdot (b_1^2, \sqrt{2}b_1b_2, b_2^2)^T \\ &= \Phi(\boldsymbol{a}) \cdot \Phi(\boldsymbol{b}) \end{aligned} \tag{23}$$

We have renamed our vectors to $\boldsymbol{a}$ and $\boldsymbol{b}$ here for clearer notation. If we can find such a function for the transformation, this will have a very large impact on the runtime of the algorithm and is indispensable for real life datasets.
We can take this idea even further, by not even knowing which theoretical transformation our kernel does. There are kernels which map to infinitely dimensional space, yet they are still easy to deal with for computers. It can be shown that we can use any symmetric and positive definite function as a kernel, the mathematical reasoning for this is not covered here, but can be found in other literature [5].

C. Bringing it back together

With both of these modifications we turn back to our optimization problem 19. Incorporating both changes gives us our final optimization problem:

$$\begin{aligned} \max_{\boldsymbol{\Lambda}} \quad & \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j K(\boldsymbol{x_i}, \boldsymbol{x_j}) \\ \text{s.t.} \quad & \mathbf{0} \leq \boldsymbol{\Lambda} \leq \mathbf{1}C \\ & \sum_{i=1}^{n} \alpha_i y_i = 0 \end{aligned} \tag{24}$$

As we can see, we hardly had to change our formula 19 at all, yet the new possibilities are immensely important for solving real life categorization problems. As a last question, we might wonder what the best kernel for our data is and which value we should choose for C. The answer to this question often depends on the data itself and goes beyond the scope of this paper. But much research has been done on this topic and the choice for these so-called *hyperparameters* can be seen as a minimization problem itself [6].

IV. Multi-class SVMs

We now turn our focus to multi-class SVMs. As the name implies, instead of just having two classes we can have an arbitrary number of classes to distinguish. The two most well-known methods will be covered here: One-against-the-rest and One-against-one. The basis for both of these still is the two class SVM. There have been efforts to create models which take into account all classes at once, however due to the much more complex optimization problem it does not seem suitable in its current form, since training times are mostly higher and the accuracy isn't necessarily better [7].

A. One-against-the-rest

Assuming now that we have n classes for our data, we will construct n different classifiers. The i-th classifier will be a regular two-class classifier which separates the class i from all $n-1$ other classes via a hyperplane, hence the name. It can happen that multiple classifiers predict the new data point to be in its corresponding class. Either we accept this, or we use the winner-takes-all approach, in which we usually measure the distance to the hyperplanes and choose the one with the highest distance.

B. One-against-one

In this method we have $\frac{n(n-1)}{2}$ different two-class SVMs. Each one represents the distinction between only two of the n classes. To determine the class, the usual approach is to use max-win voting. When we predict on a new data point, we let every classifier predict which of the two classes it thinks the point should be part of. We then increase the vote for that class by one. The class with the highest vote wins.

C. Comparison of multi-class methods

From empirical comparisons [7], [8] the common consensus is that both methods yield similar results. However, even though we need more classifiers in the second method, due to the lower sample count the training time is usually lower. This is also noted as the reason for why the popular SVM library LIBSVM uses one-against-one [9]. Practically speaking, it is difficult to say which method is best, as each one will behave differently based also on which kernel and C value is used, for example. For now, a good rule of thumb seems to be to just go with one-against-one until a clearly superior method is found.

V. Summary and Outlook

In this paper we have seen how Support Vector Machines work. From the simple idea of constructing a hyperplane with maximum distance to the closest datapoints, over extensions to allow this idea to also work with non-linearly separable data, to multi-class SVMs which can be utilized for a multitude of day-to-day problems. We can now understand, why one strong point of SVMs is that they are very efficient. After training a model, which means finding the weights for the support vectors, we can simply store these vectors which will usually only be a very small fraction of the original training set, resulting in a small file size. This also allows them to be used in embedded systems, like a raspberry pi for example [10].
SVMs are also suitable for regression, which opens up many more possible applications of this technology [11]. These expansions to the base idea of SVMs certainly exist because the concept is easy to visualize on a basic level. They offer an intuitive way to think about them, also giving opportunity for further research. Because of these reasons, SVMs deservedly get a spot in many data scientist's toolboxes.

References

[1] R. Gupta, A. Rastogi, M. Chandel, and R. Ahemad, "A machine learning approach for detection of fraud based on svm," *International Journal of Scientific Engineering and Technology*, vol. 1, p. 194, 07 2012.

[2] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, Sep 1995. [Online]. Available: https://doi.org/10.1023/A:1022627411411

[3] K. Alexandre, *Support Vector Machines Succinctly*, 2017. [Online]. Available: https://www.syncfusion.com/ebooks/support_vector_machines_succinctly

[4] D. KALMAN, "Leveling with lagrange: An alternate view of constrained optimization," *Mathematics Magazine*, vol. 82, no. 3, pp. 186–196, 2009. [Online]. Available: http://www.jstor.org/stable/27765899

[5] T. Hofmann, B. Schölkopf, and A. J. Smola, "Kernel methods in machine learning," *Ann. Statist.*, vol. 36, no. 3, pp. 1171–1220, 06 2008. [Online]. Available: https://doi.org/10.1214/009053607000000677

[6] R. Stecking and K. B. Schebesch, “Comparing and selecting svm-kernels for credit scoring,” in *From Data and Information Analysis to Knowledge Engineering*, M. Spiliopoulou, R. Kruse, C. Borgelt, A. Nürnberger, and W. Gaul, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 542–549.

[7] Chih-Wei Hsu and Chih-Jen Lin, “A comparison of methods for multi-class support vector machines,” *IEEE Transactions on Neural Networks*, vol. 13, no. 2, pp. 415–425, March 2002.

[8] K.-B. Duan and S. S. Keerthi, “Which is the best multiclass svm method? an empirical study,” in *Multiple Classifier Systems*, N. C. Oza, R. Polikar, J. Kittler, and F. Roli, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 278–285.

[9] C.-C. Chang and C.-J. Lin, “Libsvm: A library for support vector machines,” *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 3, May 2011. [Online]. Available: https://doi.org/10.1145/1961189.1961199

[10] G. E. Sakr, M. Mokbel, A. Darwich, M. N. Khneisser, and A. Hadi, “Comparing deep learning and support vector machines for autonomous waste sorting,” in *2016 IEEE International Multidisciplinary Conference on Engineering Technology (IMCET)*, 2016, pp. 207–212.

[11] H. T. Nguyen, S. S. H. Ling, and H.-k. Lam, *Computational Intelligence And Its Applications: Evolutionary Computation, Fuzzy Logic, Neural Network And Support Vector Machine Techniques*. Singapore, SINGAPORE: World Scientific Publishing Company, 2012.