Introduction to Support Vector Machines with Python

**Abstract**

With computing power increasing, more and more intelligent algorithms have been used to classify data.  One of the most popularized Machine Learning algorithms is the Support Vector Machine (SVM). This technique finds an optimal solution space that segregates classes based on their features in a dataset. SVM has been utilized for solving various pattern recognition and classification problems in many fields. Yet, the use of this technique in psychology is still unpopularized. One possible reason is the misconception that it is too complex to understand and/or implement. This tutorial aims at demystifying this technique by elaborating more on what is an SVM, how it functions intuitively and how to implement it in Python. Furthermore, it dives deeper into the mathematics behind the SVM. This tutorial is divided in following fashion: an intuitive introduction to the SVM, a guide on how to implement it in python, a detailed explanation of the mathematics behind SVM and a short discussion.

# Intro

Thanks to technological advancements in both hardware and software as well as the democratization of computing resources, Machine Learning (ML) has become the *de facto* approach for data analysis. Both in research and in industry, the power of ML has been leveraged successfully in a plethora of domains and the problem of supervised classification is considered practically solved.

Before the more recent advent of neural networks and Deep Learning methods more generally, Support Vector Machines (SVM) most often provided peak performance when it came to supervised classification tasks. While nowadays they might not always provide state-of-the-art performance, for many applications they remain highly-performant and they remain much easier to design and implement by non-specialized analysts.

The emergence of SVM goes back to the late seventies (Vapnik, 1979), but was popularized only during the late nineties due to data storage and computational power limitations. It progressively became widely accepted and utilized for solving various pattern recognition and classification problems in many fields (Burges, 1998; Suykens & Vandewalle,1999; Lutz, et.al, 2010). These include text categorization and face recognition (Joachims, 1998; Guo, Li, & Chan, 2000), condition monitoring, diagnostics, and medical imaging, (Mourao-Miranda, Bokde, Born, et.al, 2005; Widodo & Yang, 2007), gene and protein selection and analysis (Guyon, Weston, Barnhill, & Vapnik, 2002) as well as the continued application to neural networks and artificial intelligence (Asfaram, Ghaedi, Azqhandi, & 2016).

In Psychology however the use of the SVM particularly and machine learning algorithms more generally,remains scarce and is rarely (if at all) taught at the graduate level or in advanced undergraduate statistical courses.Oftentimes, this results from an over-estimation of the mathematical complexity behind the algorithm, the lack of out-of-the-box statistical software providing SVM functionality or the lack of resources and training to implement pre-built libraries in analysis pipelines through programming languages. Thus, the purpose of this tutorial is to provide an introductory to the basic concepts behind SVM and to help beginners with no prior knowledge to implement this intelligent algorithm in Python.

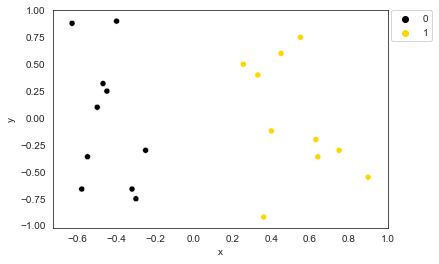
# Background

Statistical applications in psychology most often fall within the domain of descriptive and inferential statistics. Traditionally, once individual-level data is gathered and cleaned, it goes through a series of statistical models and tests such as ANOVAs, T-tests, ANCOVAs, to determine if there are significant differences between groups of participants and whether those patterns would hold beyond the specific dataset at hand.

However, these methods fall short when it comes to classification, i.e. determining if an individual data point not seen before pertains to one of the sampled groups based on its specific characteristic, or features... At this point, what can be used is a classifier, a method that allows researchers to determine to which known groups an individual belongs too. One such technique is the SVM. Simply worded, an SVM takes a database of known, pre-classified data, learns to distinguish between the known classes, and uses this knowledge to classify new data.

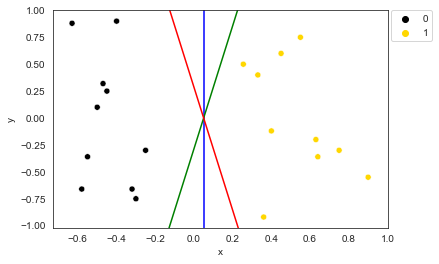
This is accomplished through what is called a supervised learning algorithm. This learning method relies on data that has been classified *a priori.* It attempts to organize itself in a way that best respects the restrictions of the known data, i.e., the best *fit*. For example, the SVM can only classify an individual as depressed or not after it has learned from the classified dataset. This set of known data is called the training set. In this set, each participant is represented by an input feature vector and its corresponding target vector. In layman's terms, an individual is represented by the values of each measurement (also called features) used to assess him, (e.i., stress, anxiety and emotional scores) and his associated group (e.i., depressed or not). The magic behind the SVM is that it learns to optimally represent the training set of input feature vectors and their corresponding class labels and uses this representation to classify new data. In other words, it looks to determine to which class does the new individual’s feature resemble the most.

Consider this simple 2D classification task in figure 1, where the training dataset is represented by 20 input feature vectors that pertain to one of the two classes, black or yellow. The x and y axis represent two arbitrary features (e.i., stress and anxiety score). Simply put, the goal of the SVM is to create a line that would separate the two classes as best as possible. In a classification task, this separating line is referred to as the decision boundary. To facilitate understanding, it is important to note that for a 2D task, this bound refers to a line, for a 3D task, a plane and for an *n*D task, a hyperplane. Thus, generally speaking, a consensus is to referee to the decision boundary (or any lines) as a hyperplane.



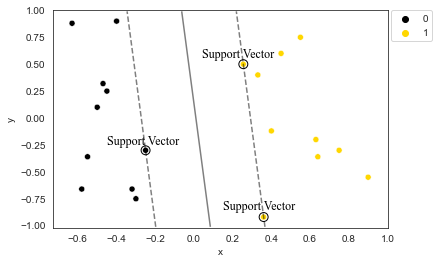
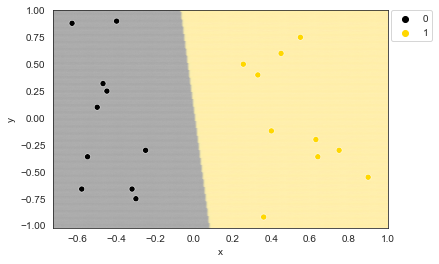
1. Simple 2D linear classification task.

As noted, for such a task there are various possible decision boundaries that can separate the training set in the two classes,  as seen in figure 2. The question becomes which one does it best?



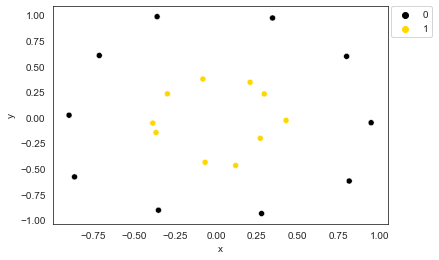
1. Various possible decision boundaries to classify both classes.

To solve this, an SVM defines the best classification boundary as the one that leaves the most distance between the two classes. To do so, it creates a hyperplane for each class. Each hyperplane is found using points that are closest to the opposite class, also known as support vectors. The goal of the SVM is to find the widest margin or gap between the two classes. From there, the decision boundary can be determined as the middle point of the two class hyperplanes (figure 3a). Once the decision boundary is found, any new instance can be classified by looking at its location on the feature space, i.e. on which side of the decision boundary it is located (figure 3b). In other words, the SVM learns to perform a classification task by finding the input feature vectors that are closest to the opposing class and using them to establish the optimal decision boundary.

a) b)

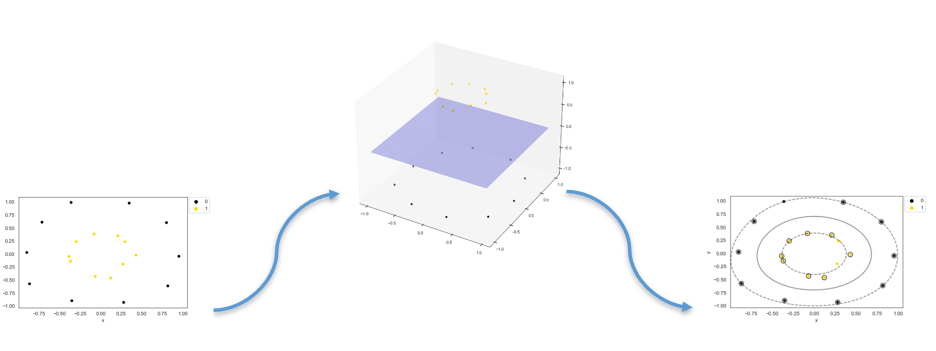
1. A) SVM’s decision boundary (straight line) definition from class hyperplanes (dotted line) and support vectors (circled dots). B) SVM decision zones, in the black zone everything will be classified as 0 while in the yellow zone as 1.

So far, the data illustrated has been linearly separable, which means a single straight decision boundary is sufficient to classify the dataset. However, this might not be the case for real world data. Often, a problem can be non-linearly separable. In such a case, the solution might require a curved decision boundary or even multiple decision boundaries. Figure 4 illustrates a simple non-linearly separable task with 2 features (x and y) and 2 classes (yellow and black). As it can be noted, no single straight hyperplane can be used to adequately separate the two classes.



1. Circular dataset

Fortunately, the SVM can learn to classify non-linear tasks by using the kernel trick. Essentially, this method permits to project the input feature vectors into an infinitely higher dimensionality and use this augmented feature space to find a decision boundary while following the same general procedure of using support vectors. Figure 5 visually illustrated this process.



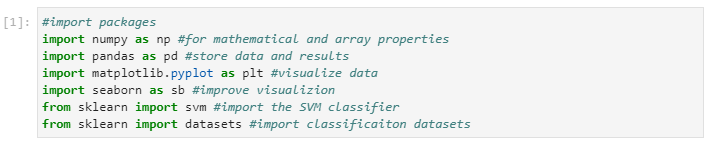
1. SVM’s decision boundary (straight line) definition from class hyperplanes (dotted line) and support vectors (circled dots).

To better understand this process, an in-depth explanation and mathematical breakdown of the SVM as well as the kernel trick is available towards the end of this tutorial. What follows this introduction will be the implementation of the SVM through python by using one of the most popular libraries called SciKit-Learn (<https://scikit-learn.org/stable/>). The goal will be to demonstrate how a SVM can be used in Python to classify non-linear datasets and lower the learning curve for novel users.

**A guide to SVM in python**

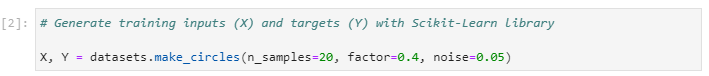
To implement the SVM in Python, we will use *Scikit-Learn*. This handy library comes installed with Anaconda and contains many pre-built classifiers and machine learning algorithms (see Rolon-Mérette, D., Ross, M., Rolon-Mérette, T., & Church, K for a simple installation tutorial). This section will consist of a step-by-step implementation of an SVM in Python using an arbitrary 2D non-linear dataset, the same as illustrated in figure 4, and a more complex dataset with many features (multi-dimensional).

To begin, we import the necessary libraries and packages, as shown in figure 12. *Numpy* is useful for various computing and mathematical operations. *Pandas* allows for inputs and results to be stored in organized table-like datasets. *Seaborn* and *Matplolib* are great visualization libraries. The *SVM* package contains the SVM classifier, and the *datasets* package contains a vast amount of different classification and non-classification datasets. Both are found within the *Scikit-Learn* (*sklearn)* Library.



1. Import libraries and packages.

Then, we import our first dataset using the *make\_circles* function from *Scikit-Learn*. This creates two circles (one within the other), each comprising its own class. The different parameters control the shape and size of the dataset. This dataset will be stored in the *X* (input feature vectors) and *Y* (target vectors). This will serve as the training set that will be presented to the SVM for it to find the optimal decision boundary. The two class labels are 0 or 1.



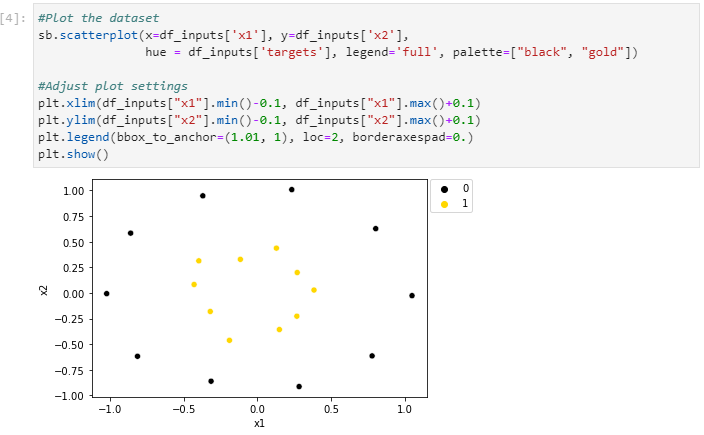
1. Import training inputs

Optionally, to better view the dataset, we can store the inputs and targets into a *Pandas Dataframe*. This assembles the dataset into a table-like manner and eases many processes like visualization, accessing and storing. This is first done by creating a pandas dataframe for the inputs (*df\_X*) and targets (*df\_Y*) separately and then joining them together in *df\_inputs*. We can then view a sample of the dataset using the function *head( )*.



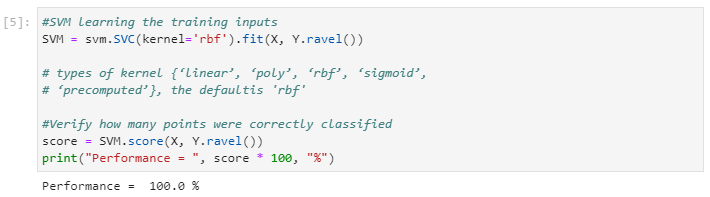
1. Store dataset into *Pandas Dataframe*

Time to visualize the data. We can use both the *Seaborn* and *Matplotlib* library to plot our training inputs and color them according to their class label. Again, this step is optional.



1. Visualize a dataset using *Matplotlib* and *Seaborn*.

We can now train the SVM for classification. Since it is already imported via *Scikit-Learn*, we can just call it with *svm.SVC* and its function *fit* to train the SVM on the inputs (X) and targets (Y). We can store this SVM in a variable, called *SVM*, to easily access it later. There are many parameters to the *svm* package, one interesting is *kernel* which defines which kernel to use.  For this tutorial, we will use the *rbf* (Radial Basis Function) kernel, but there are several others. For a more in-depth explanation for all the SVM parameters, please visit the Scikit-Learn website (<https://scikit-learn.org/stable/modules/svm.html>).

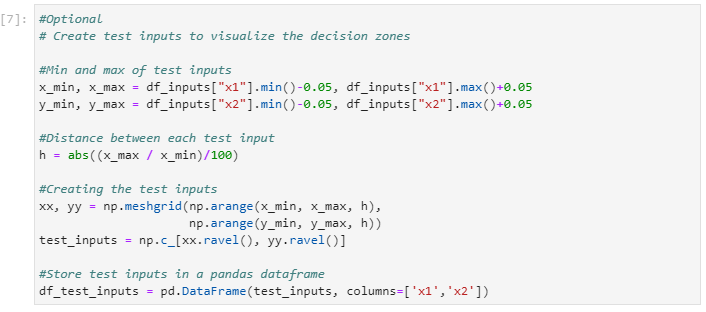


1. SVM learning of training inputs.

After training the SVM, we can test whether the dataset has been correctly classified by using the *score* function. This will yield the SVM’s performance on the training inputs. As shown, with a performance of 100%, the SVM has correctly classified all inputs. Furthermore, we can also infer what class it would predict from a new input. For example, say the SVM received the input (1, 1), which would visualize to be in the top right corner in figure 9. Clearly, this input belongs to the “0” (or black) class. To evaluate if the SVM can determine this, we can use the *predict* function on this new input. As shown, the SVM’s “prediction” is that the class of the input is 0.

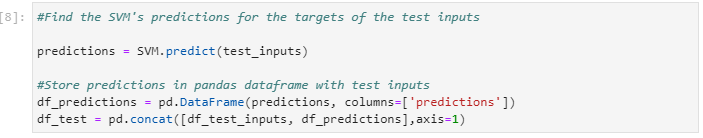


While predicting a new output is useful, it would be even more interesting to observe what is the decision boundary of the SVM. To do this, test inputs can be used to evaluate what the SVM would predict as their class labels. These test inputs can also be used to determine the decision boundary of the SVM. Thus, we create these test inputs using *meshgrid*, a package included in *Numpy* that will evenly space out the test inputs between preset values (the lowest and highest *x* and *y* coordinates found in the training inputs). The distance between each input, or step size, is given by the parameter *h*. A low value of *h* will yield a higher number of test inputs and a smoother and more precise decision boundary. A high value of *h* will yield a lower number of test inputs and a rougher and less precise decision boundary. To ensure that the *h* is small enough, we can set it at about 100 times smaller than the width of our dataset distribution. The test inputs are then stored in a *Pandas Dataframe.*



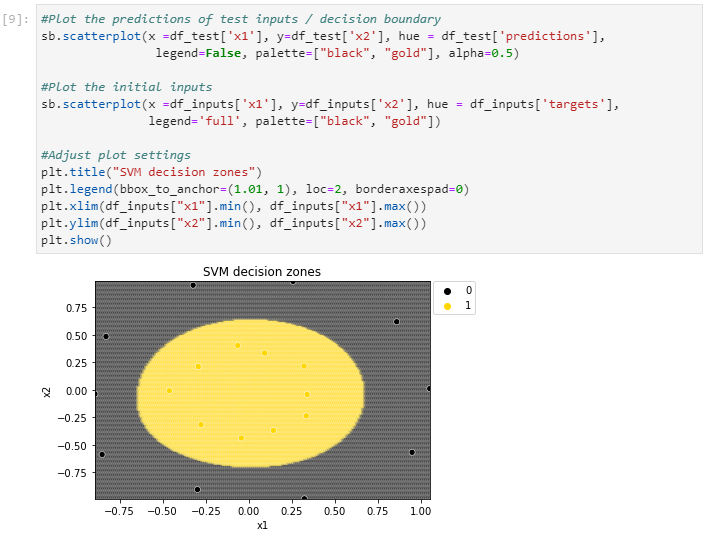
1. Creating test inputs.

Subsequently, the *predict* function can be used on the test inputs to determine the decision boundary. First, the results, or predicted labels, are stored in a variable, *predicted\_labels*, and it is then subsequently stored in a *Pandas Dataframe* along with the test inputs.



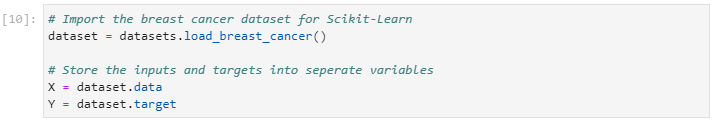
1. SVM predictions on test inputs.

Finally, we can plot the decision boundary using *Seaborn* and *Matplotlib* to visualise the decision zones that the SVM has learned to represent for each class.

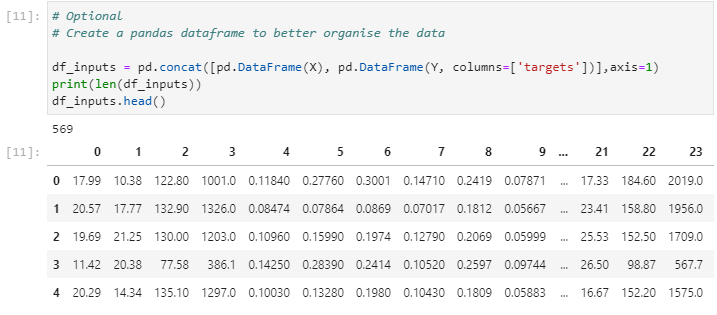


1. Decision Boundary on test inputs.

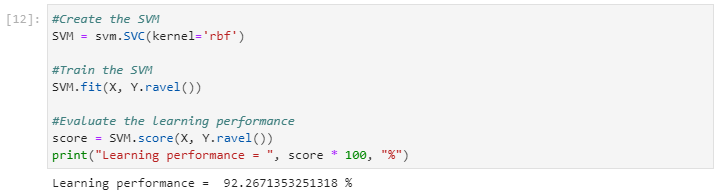
As it can be noted, implementation of the SVM in Python requires just a few lines of code. This makes using the SVM in python very simple and gives access to its classification power by the click of a button. This is not only applicable to 2D datasets, but to bigger, more “realistic”, datasets as well. In the *Scikit-learn* library, there are many such datasets provided to test, practice or evaluate a model. One of such is the breast cancer dataset, which contains 589 inputs with 30 features for a malign or benign tumour. This data set can be imported using the same“*datasets*” package already used for the previous task. We can store the inputs in a variable X and the targets in a variable Y.



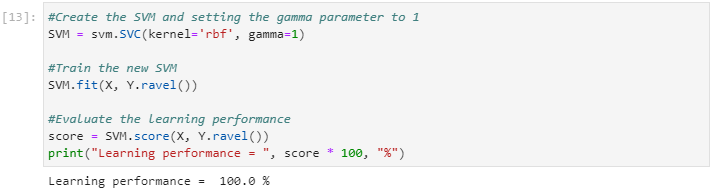
Once again, to get an organised view of the dataset, we can use a *Pandas* dataframe. As previous, this step is optional. We can *print* the length of the dataset to see that there are 569 inputs with 30 features. We can use the *.head()* function to see the first five inputs.



To train and evaluate an SVM on the new dataset, the same steps as in figure X can be done. First, we create the SVM (in this case with an “rbf” kernel) and then train it on the dataset with the *fit* function. Subsequently, with the *score* function we can observe that the SVM has correctly classified around 92% of the dataset.



Although a classification performance of 92% might seem decent, it would be desirable to achieve 100% performance. Thankfully, the SVM package form *Scikit-Learn* comes with many parameters that can be changed and tuned to a specific task. Once such a parameter is *gamma* which can be manually set to any value. Essentially, the *gamma* parameter influences whether “closer” or “further” points are considered important by the SVM. Although such parameters fall outside the scope of this tutorial, for a more in-depth explanation, please see <https://scikit-learn.org/stable/auto_examples/svm/plot_rbf_parameters.html#:~:text=Intuitively%2C%20the%20gamma%20parameter%20defines,the%20model%20as%20support%20vectors>. Thus, by setting the gamma parameter to 1, we can achieve 100% classification performance on the breast cancer dataset.



**The math behind SVM**

**Notation**

We first define an individual data point or measurement by the row vector x:

x=x1 xj xF

Each element xj represents the magnitude of feature j=1,2,…,F with F being the total number of features. The xvectorcan be thought as a data point in an F-dimensional space F⊆CF and most commonly F⊆RF. We call F the feature space. The training set is composed of *N* individual measurements, which we can index by the letter i =1,2,…,N and arrange them in table form such that the training set can be written as:

X=x1 xi xN =x11 x1j x1F xi1 xij xiF xN1 xNj xNF

In our example, a vector xi refers to an individual dot (either blue or red) with its two associated features (*x* and *y*, F = 2).In the context of a binary classifier, we define the class label for data point xias yi=-1, 1. The label of a data point is an arbitrary representation of the class, i.e. 1 (blue) or -1 (red). The training dataset of N measurements is therefore defined as the pairs:

x1, y1,…,xi, yi, …,xN, yN

We can concisely define the training set and individual classes as:

Xy=X y =x1 y1 xi yi xN yN

**Hyperplane Geometry**

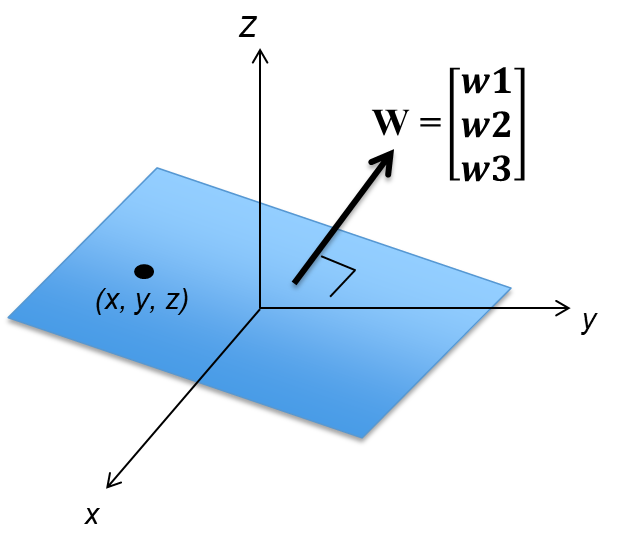
SVM performs its operations based on class and boundary hyperplanes. It is necessary then to first define them in mathematical terms. A hyperplane can be fully defined by a vector perpendicular to its surface, what is called its normal vector wand an offset scalar value *c* in the general equation for a plane:

w=w1 wj wF

⟺w∙x=c

Where represents the dot-product between the two vectors. For simplicity, the dot product notation will be dropped and considered implicitly.

⟺wx=c



1. Definition of a hyperplane by its normal vector *w*

We can see how such definition of a plane is sufficient by taking the example of a hyperplane in 2D space, a straight line:

wx=c

w1 w2 x y =c

w1x+w2y=c

y=-w1w2x+cw2

y=mx+n

Before finding the decision boundary, each class must be represented by a hyperplane. For each of the two classes of the training set data, we define the hyperplanes -, +, which respectively correspond to class yi=-1 and yi=1:

-: w-T x-=c-

+: w+T x+=c+

For the SVM to work, a few restrictions must be imposed. The first is that the hyperplanes are to be parallel:

-+

w-w+

We further assume the normal vectors to be equal and pointing in the direction of the positive class:

w-=w+=w

We also impose the following restriction on the offset constants:

cy=yi-b

Where *b* is a shared constant across hyperplanes. Given these restrictions, the hyperplanes can be concisely defined as:

y: wx+b=y

From these two hyperplanes, the decision boundary 0     is simply:

0: wx+b=0

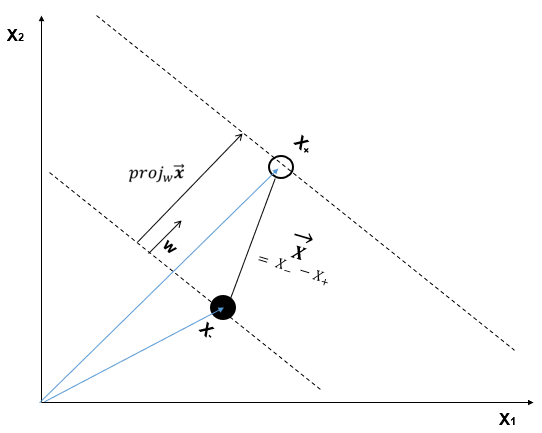
Since wpoints in the direction of the positive class, any point above0 is part of the positive class and any point under 0 is part of the negative class. Therefore, the classification criteria for any new point xi becomes:



The specific value of wandb however needs to be found for each specific data set. The process of finding their specific values is called parameter optimization. The next part of this tutorial explains how to find the specific wandb that would allow to classify the data with the classification equation.

**Optimization – Training the SVM**

SVM tries to classify new data by defining the decision boundary that leaves the most space between classes. In this case, our first optimization condition is that the distance between y or margin must be the greatest possible. To find this distance, we can use something called a projection (figure 5). Essentially, the distance between the closest data points from each class (x, these two points are on - and +) will be projected onto vector *W*. Sine this vector is perpendicular to the hyperplanes, the projection will result in the distance between the - and +.



1. Visualisation of the projection to find the distance between two planes

Once again, we can find the distance between the two planes by selecting two arbitrary points x-and x+from - and + respectively. We define the vector pointing from one of these arbitrary points to the other x=x--x+. The distancebetween the two planes can be found from the projection to the normal vector:

projwx=xw‖w‖w‖w‖

=xw‖w‖2w

⟺D1, 2=‖projwx‖

=‖xw‖w‖2w‖

=xw‖w‖2‖w‖

=xw‖w‖

D1, 2=x--x+w‖w‖

From the definitions of the planes, we relate the arbitrary points to the normal vector and offset constant:

w xy+b=y

⟺wT xy=y-b

⟺w‖w‖2w xy=w‖w‖2y-b

⟺xy=w‖w‖2y-b

Therefore:

D1, 2=x--x+w‖w‖

=w‖w‖2-1-b-w‖w‖21-bw‖w‖

=-1-b-1-b‖w‖

=-1-b-1+b‖w‖

D1, 2=2‖w‖

Maximizing the distance between the hyperplanes is therefore equivalent to minimizing ‖w‖

 For mathematical convenience, this is equivalent to minimizing 12‖w‖2or‖w‖2. F    or consistency with the literature, we will use 12‖w‖2.

Therefore, the optimization condition given by the maximization of the distance is equivalent to:

arg12‖w‖2 L

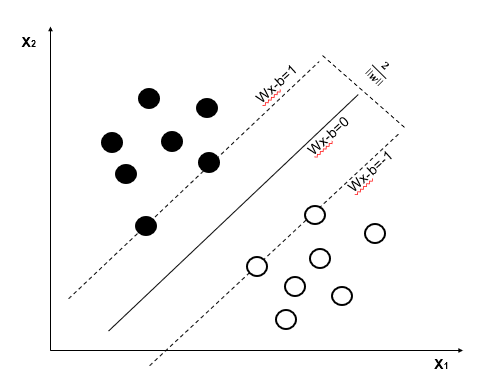
This equation essentially means that across all possible values of ***w*** and ***b***, we want to minimize the magnitude of w. This optimization equation consists of a quadratic, or convex, problem. This is a good thing because there will only be one optimal solution, so the SVM will not get stuck in a local minimum and will always find the global minimum (correct solution).

Maximizing the distance between classes is not sufficient by itself to appropriately define the decision boundary. Indeed, if applied on its own, the optimization of ***w*** would just set two hyperplanes at an infinite distance from each other and the data would simply fall in between the two planes. One can also notice that the optimization criteria does not consider the value *b* defining the hyperplanes. Moreover, an additional problem with the unique optimization condition is that the minimum would be obviously found at w=0 which would make no geometrical or practical sense in our problem.

The additional restrictions necessary to find wand b depend on the distribution of the specific training set. For all practical purposes, the distribution and respective optimization problem fall under one of the following three categories: (1) linearly-separable data, (2) almost linearly-separable data and (3) non-linearly separable data.

**Linearly Separable Data – Hard Margins**

Hard margins are applied when the data is linearly separable, which means that the classes are distributed in such a way that the decision boundary can be drawn between the two classes and all points of each class are found in their respective side of the decision boundary.



1. Hard-margin classification by the SVM.

Mathematically, the condition that all points must be on the right side of the decision boundary is defined as:

yi= -1 ⟺wxi+b≤-1

yi= 1 ⟺w xi+b1

Or more concisely:

yiwxi+b1 ∀ i ∈1,…,i,…,N

For the hard-margins categorisation, the class hyperplane lies on the data points closer to the decision boundary. These points are called support vectors (hence the name Support Vector Machine). Fun fact, you only need a few support vectors to classify an entire dataset.

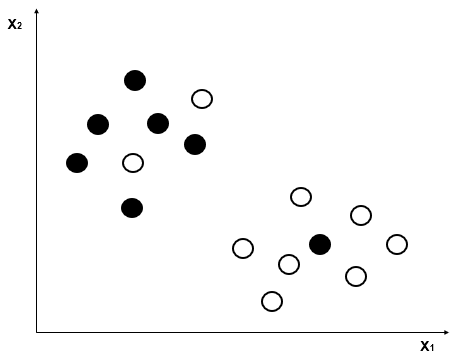
The full optimization problem for linearly separable data therefore becomes:

arg12‖w‖2 ,  s.t.  yiwxi+b1  ∀ i ∈1,…,i,…,N

Each type of SVM (linear, near-linear, non-linear) is essentially an optimization problem. Depending on the SVM type, the optimization problem changes slightly. The next section of this tutorial will dive into building a linear decision boundary with a less strict classification rule, where an arbitrary amount of error will be accepted.

**Linearly Separable Data – Soft margins**

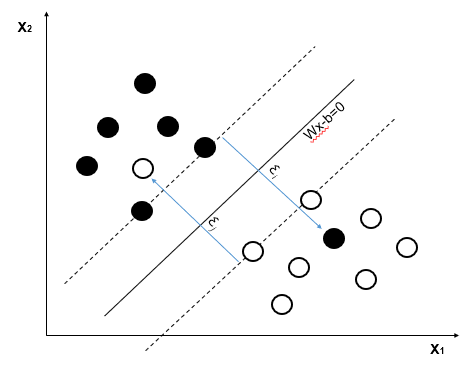
The assumption behind the previous linear classifier is that the two classes are clearly divided within the feature space. But it is often more common to find overlap between the two classes as seen in figure 7



1. Near-linear classification problem.

While in this type of situations a linear divider could not be used, SVMs can be effective if we introduce a slack variable . This variable represents the importance of the error of a data point or how deep the data point is in the other class’ territory, as seen in figure 8. We can calculate the total errors as the sum of the individual errors and add them to the optimization problem. :

arg12‖w‖2 +ii,  s.t.  yiwxi+b1-i  ∀ i ∈1,N

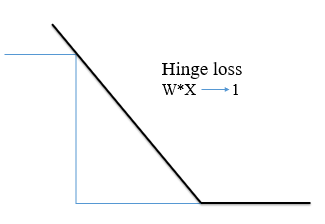


1. Soft-margin classification.

The parameter determines the importance we attribute to the errors. This is chosen by the data analyst for each specific situation at hand, depending on how lenient one can be about accepting errors. If we want to minimize them, a large value of will get us to the hard-margin situation (where no errors are allowed to happen). On the other hand, a low value will be more lenient and at the extreme values of 𝜆, it will stop considering the data altogether.

If data point *i* were to fall on the right side of its class’ support vector, we would expect the slack variable to be 0, independently on how far in its class’ territory it is. Also, the furthest away the data point is inside another class’ territory, the higher we would want i to be. We therefore define i as the “hinge loss” function:

i=max0, 1-yiwxi+b



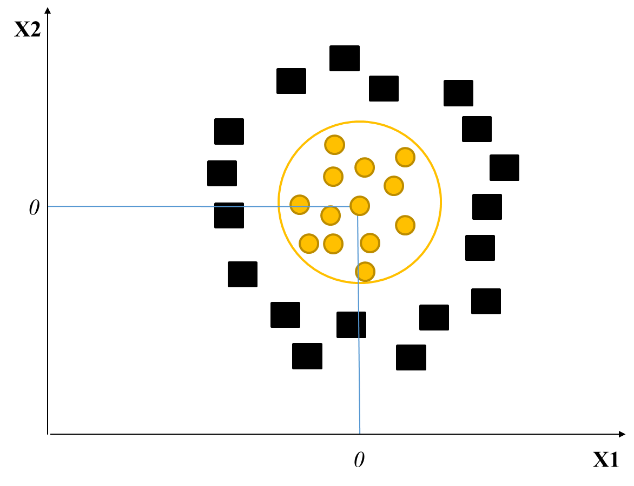
1. Hinge loss function representation.

We can then rewrite the optimization problem as:

arg12‖w‖2+imax0, 1-yiwxi+b

**Non-linear classifier**

The previous definitions assume that the two classes can be separated linearly (or allow small errors). Rarely data is distributed in such a neatly organised fashion. As seen in figure 10, most datasets are arranged in complicated ways.

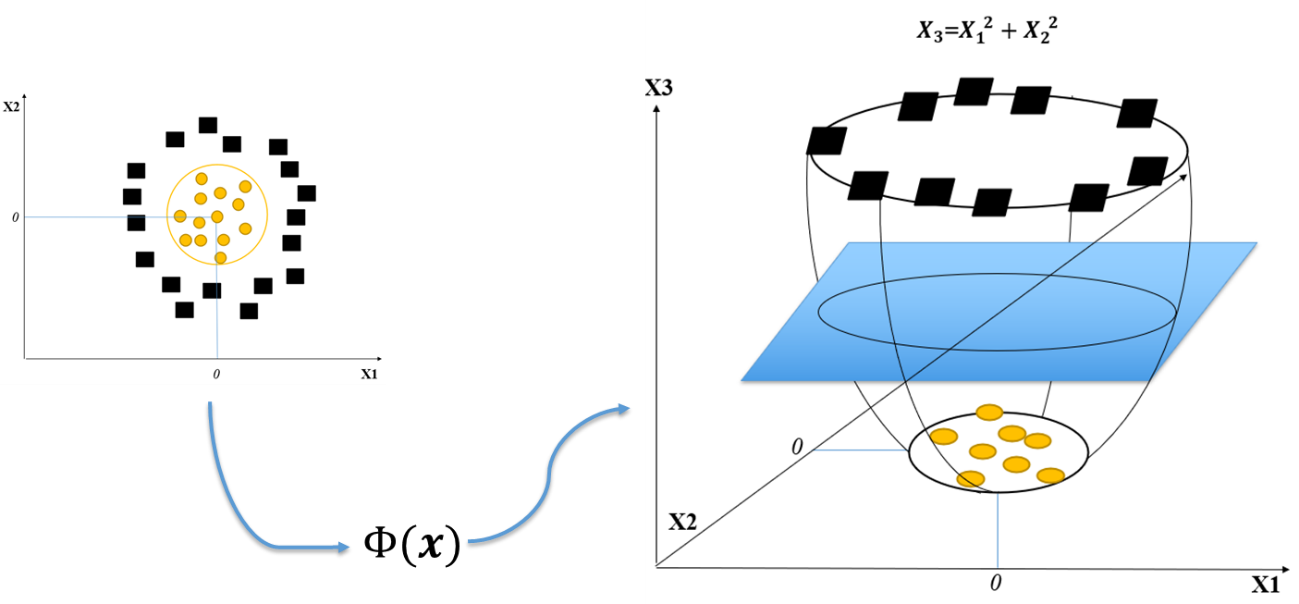


1. Non-linearly separable problems.

While SVM could not be applied to classify this type of data, it is possible to apply non-linear operations on the dataset to increase their dimensionality until they become linearly separable. A dataset might be non-linearly separable in its input space (X1 and X2, 2D in this example), but can be mapped (with the use of a non-linear transformation) into a higher dimensionality called the feature space (Z1 and Z1). In this feature space, the dataset will be linearly separable, and a decision boundary can be implemented. Once the decision bound is found, it is mapped back into the input space.

In sum, this transformation, denoted by x, increases the dimensionality of the measurements to allow the SVM to determine an appropriate decision boundary. Figure 11 provides a visualisation of this process with the following transformation:

xi'=x=xi jxij2



1. Transformation on the non-linearly separable distributions to increase dimensionality and allow classification.

It is important to note that all operations of SVM optimization and classification depend on the dot product between two vectors. This is significant because there are mathematical functions that allow these dot products to be done in higher dimensions, without ever going into these higher dimensionalities.

If we were to transform all data points into a higher dimensional feature space, perform the operations and transform them back to its original input space, this would quickly become computationally prohibitive and would limit the SVM to only small datasets. Furthermore, some transformations even require increasing dimensionality to infinity.  If that situation would arise, it would be impossible to compute or store such an amount of data.

We can however avoid such problem with the use of Kernels and its functions as shortcuts:

Kx, y=xy

Such a shortcut allows us to perform computations on the higher dimensional space without having to transform the data first. We can illustrate these operations by assuming a simple cubic transformation of 2D space:

x=x1 x2 =x13 x12x2 x1x22 x23

Optimization and classification require dot products as:

xy=x13 x12x2 x1x22 x23 y13 y12y2 y1y22 y23

=x13y13+x12x2y12y2+x1x22y1y22+x23y23

=x1y1+x2y23

We can replace the long computations of the cubic transformation by a polynomial kernel:

xy=Kx, y=xy3

=x1y1+x2y23

Even with such a simple example, we see how we can save on computations. Have we used the direct transformation method, we would have to perform 16 multiplications and 3 sums. On the other hand, the Kernel trick requires us to do only 4 multiplications and 1 sum. Commonly used Kernels are the polynomial Kernel, the Radial Basis Function or Gaussian Kernel and the Hyperbolic tangent (or sigmoid) Kernel. Respectively:

Kx, y=xy+cd

Kx, y=exp-‖x-y‖22

Kx, y=tanhkxy+c with k>0, c<0

The specific kernel to use depends on numerous factors, such as the arrangement of data, the type of dataset or the preference of the analyst.

**Discussion**

In this tutorial, the SVM was presented as a supervised classifier capable of classifying linearly and nonlinearly separable data. Furthermore, a simple python implementation was shown through the Scikit-learn library. What followed was a detail explanation of the mathematical mechanism behind the SVM, and how it relies on building class hyperplanes by finding support vectors while respecting two conditions: hyperplanes class having the widest distance/margin between them and data points falling on the margin (the support vectors) respecting the restriction of the b variable.

Three types of SVM were presented: the hard margin and the soft margin classifier, that are used for linearly separable data, and the non-linear SVM that uses the kernel trick. In all cases, once trained, a SVM can classify new data points with high accuracy. This being said, there are some limitations to this machine learning technique. A downside to this approach is that in some cases it requires tunable parameters (the  C, Gamma and the kernel type). To correctly find the right values for both of these parameters and to avoid overfitting, it seems like for now trial and error are the best options. Furthermore, another down side of the SVM is that it does not deal well with data sets that have the number of features greater than the number of samples.

That being said, the SVM is still seen in general as the most performant machine learning algorithm for classification tasks out there. Additionally, SVM is not only exclusive to classification, but it can also perform regression analysis as well (Smola & Scholkopf, 2004). An important aspect of this method is that it analyses datasets without worrying about any assumptions (homogeneity, normality, etc.). Traditional statistics have always been bound by these assumptions, and while they have proven to be good tools in statistical analysis in the past, interesting alternatives are now emerging in the world of Machine Learning.

This tutorial aimed to introduce the SVM in a simple and intuitive way while also demonstrating the sophistication behind it. The Python implementation was of simple nature to illustrate how feasible it can be to use the SVM in research. Although not mentioned, there are more particular versions of the SVM out there worth exploring. Scikit-learn is a great and simple library where it is possible to find  many of these versions as well as other machine learning algorithms, such as artificial neural networks, boltzmann machines, etc.. Furthermore, they elaborate the functioning of each method. Thus it is highly recommended as the next step from this tutorial to leap around in the documentation of Scikit-learn . Lastly, as with all machine learning methods, there exist various tutorial videos that can be used to solidify or even further your understanding. This tutorial has laid the foundation for any new users to continue their exploration and learning process of the SVM. Hopefully, from now on there can be more research using the power of SVM in psychology.

**References**

Asfaram, A., Ghaedi, M., Azqhandi, M. A., Goudarzi, A., & Dastkhoon, M. (2016). Statistical experimental design, least squares-support vector machine (LS-SVM) and artificial neural network (ANN) methods for modelling the facilitated adsorption of methylene blue dye. *RSC Advances*, *6*(46), 40502-40516.

Burges, C. J. (1998). A tutorial on support vector machines for pattern recognition. *Data mining and knowledge discovery*, *2*(2), 121-167.

Guo, G., Li, S. Z., & Chan, K. (2000). Face recognition by support vector machines. In *Automatic Face and Gesture Recognition, 2000. Proceedings. Fourth IEEE International Conference on* (pp. 196-201). IEEE.

Guyon, I., Weston, J., Barnhill, S., & Vapnik, V. (2002). Gene selection for cancer classification using support vector machines. *Machine learning*, *46*(1-3), 389-422.

Luts, J., Ojeda, F., Van de Plas, R., De Moor, B., Van Huffel, S., & Suykens, J. A. (2010). A tutorial on support vector machine-based methods for classification problems in chemometrics. *Analytica Chimica Acta*, *665*(2), 129-145.

Mourao-Miranda, J., Bokde, A. L., Born, C., Hampel, H., & Stetter, M. (2005). Classifying brain states and determining the discriminating activation patterns: support vector machine on functional MRI data. *NeuroImage*, *28*(4), 980-995.

Rolon-Mérette, D., Ross, M., Rolon-Mérette, T., & Church, K (2021). Introduction to Anaconda and Python: Installation and setup.

Smola, A. J., & Scholkopf, B. (2004). A tutorial on support vector regression. *Statistics and computing, 14*, 199-222.

Wang, Z., & Xue, X. (2014). Multi-class support vector machine. *Support Vector Machines Applications (Springer)*, 23-48.

Widodo, A., & Yang, B. S. (2007). Support vector machine in machine condition monitoring and fault diagnosis. *Mechanical systems and signal processing*, *21*(6), 2560-2574.

Vapnik, V. *Estimation of Dependences Based on Empirical Data [in Russian]*. Nauka, Moscow, 1979. (English

translation: Springer Verlag, New York, 1982).