Cairo University
Faculty of Computers and Artificial Intelligence
Machine Learning Course

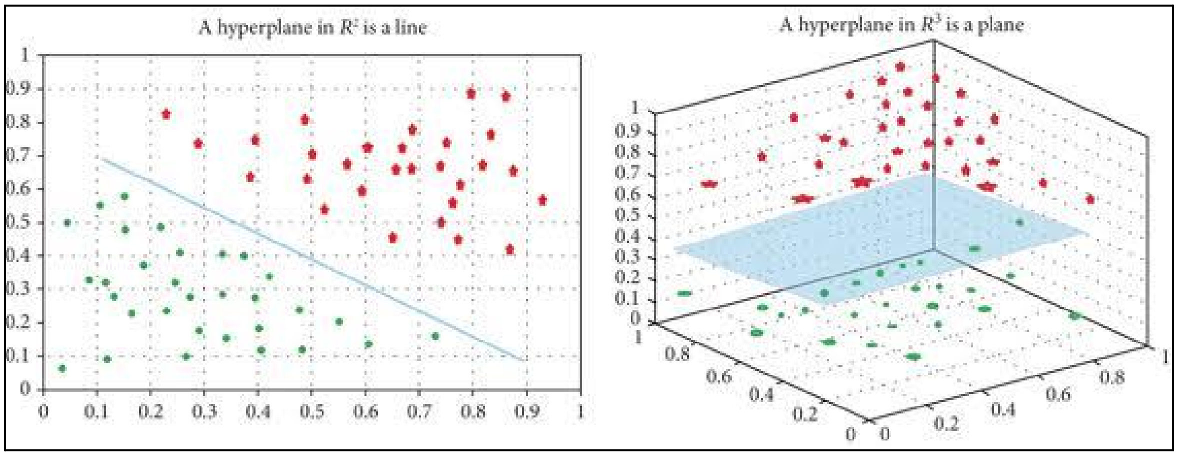
Lab 7 – Support Vector Machine

Lab Outline:

- ➢ Summary of support vector machines
- ➢ SVM using gradient descent
- ➢ The kernel trick
- ➢ SVM example in Python
- ➢ Practice

Summary of Support Vector Machines:

Support vector machine (SVM) is a ***supervised*** machine learning algorithm that is typically used for ***classification*** tasks (but can also be used for ***regression***). An SVM constructs a hyperplane in a multidimensional space to separate different classes. A good separation is achieved by the hyperplane that has the largest distance to the nearest training data points of any class because the larger the margin, the lower the generalization error of the classifier.

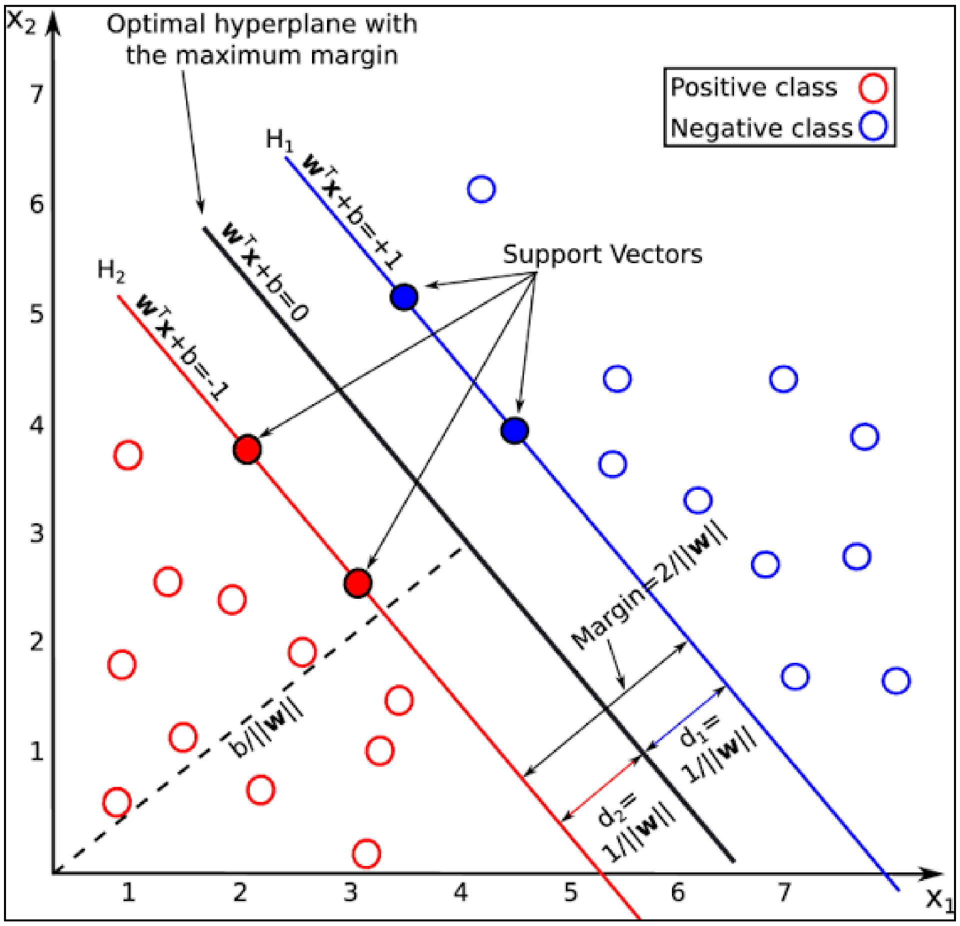


Assuming we have two features and one target with two classes (-1 and 1), we write the equation of the hyperplane we want to find as follows:

$\boldsymbol{w^T x + b = 0}$

We select two parallel hyperplanes that can separate the two classes, with the distance between them ***"margin"*** being as large as possible (the maximum-margin hyperplane is the one that lies halfway between them).

Geometrically, the distance between these two hyperplanes is (**2 / ||*w*||**), so we want to maximize this margin.



However, we also want to prevent data points from falling into the margin (being incorrectly classified), so we add the following constraints:

$w^Tx + b >= 1$ **(if y = 1)**

$w^Tx + b <= -1$ **(if y = -1)**

which can be rewritten as:

$y\ (w^Tx + b) >= 1$

This enforces a ***hard margin*** that doesn't allow any points to be misclassified. Our problem becomes an ***optimization problem*** in which we want to **minimize the inverse of the margin** subject to the "points must be correctly classified" constraint.

This is called the primal formulation of the SVM:

$$\begin{array}{ll} \underset{\mathbf{w},\, b}{\text{minimize}} & \|\mathbf{w}\|_2^2 \\ \text{subject to} & y_i(\mathbf{w}^\top \mathbf{x}_i + b) \geq 1 \quad \forall i \in \{1, \ldots, n\} \end{array}$$

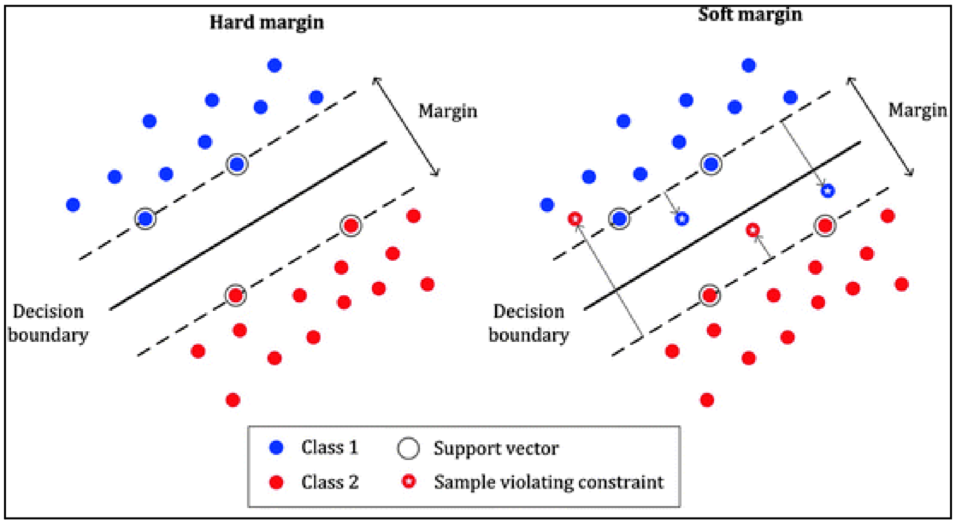
Unfortunately, problems aren't always perfectly separable with a hyperplane. So, we allow some samples to be misclassified ***(soft margin)*** by relaxing the constraint and adding a penalty term (for incorrectly classified points) in the objective function.

$$\begin{array}{ll} \underset{\mathbf{w},\, b,\, \zeta}{\text{minimize}} & \|\mathbf{w}\|_2^2 + C\sum_{i=1}^{n} \zeta_i \\ \text{subject to} & y_i(\mathbf{w}^\top \mathbf{x}_i + b) \geq 1 - \zeta_i, \quad \zeta_i \geq 0 \quad \forall i \in \{1, \ldots, n\} \end{array}$$

Since we want the penalty term to be zero for points that were correctly classified and to be proportional to the distance from the margin for incorrectly classified points, we can write the objective function as:

$$\lambda\|\mathbf{w}\|^2 + \left[\frac{1}{n}\sum_{i=1}^{n} \max\left(0, 1 - y_i(\mathbf{w}^\mathsf{T}\mathbf{x}_i + b)\right)\right]$$

The hard margin vs the soft margin is shown in the image below.



To solve the optimization problem of SVMs, we have several options. One method is to get the dual form, i.e. use Lagrange multipliers.

Another way to solve the problem is to use gradient descent.

SVM Using Gradient Descent:

Recall the function we are trying to minimize:

$$\boxed{\lambda\|\mathbf{w}\|^2 + \left[\frac{1}{n}\sum_{i=1}^{n}\max\left(0, 1 - y_i(\mathbf{w}^\top\mathbf{x}_i + b)\right)\right]}$$

Since this function is convex in ***w***, we can use gradient descent. Recall that gradient descent updates the parameters as follows:

$$w_{t+1} = w_t - \alpha\, \partial J(w)/\partial w_t$$

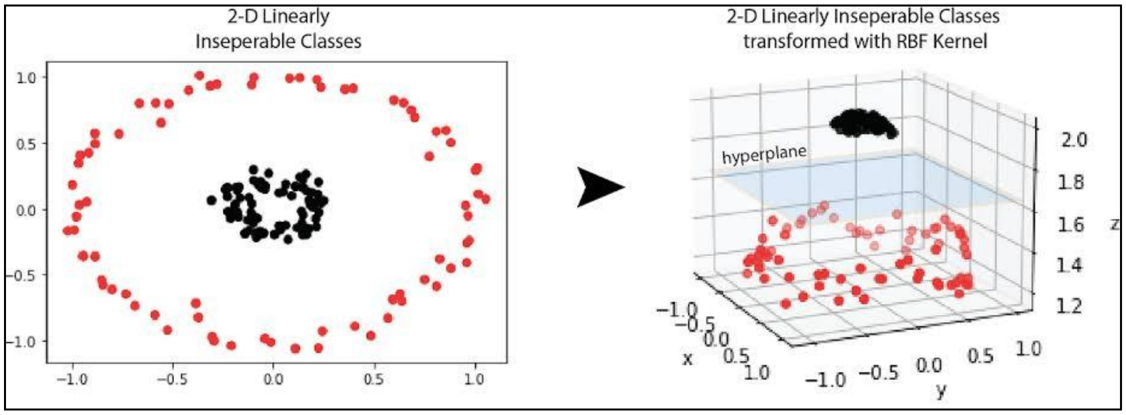
We will apply ***stochastic gradient descent*** where in every single iteration, we loop over training examples and update each parameter according to the partial derivative of the cost function at that example.

However, the hinge loss (penalty term) is not differentiable! That is why we need to use the ***subgradient method*** in which we compute the gradient for each case of the “max” function. Now, the partial derivative is straightforward and the parameters can be updated as follows:

$$\boxed{\begin{array}{llll} \mathbf{w}_{t+1} & \leftarrow & \mathbf{w}_t - \eta(\lambda\mathbf{w}_t - y_i\mathbf{x}_i) & \text{if } y_i f(\mathbf{x}_i) < 1 \\ & \leftarrow & \mathbf{w}_t - \eta\lambda\mathbf{w}_t & \text{otherwise} \end{array}}$$

The Kernel Trick:

Some problems can’t be solved using linear hyperplane (when data is entirely not linearly separable). In such a situation, SVM uses a kernel trick to transform the input space to a higher dimensional space in which data is linearly separable.



In the primal classifier, we can simply map ***x*** to ***Φ(x)*** where data is separable. However, when there are more and more dimensions, computations within that space become more and more expensive. This can be avoided by applying the dual classifier then using the kernel trick.

Primal version of classifier:

$$f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$$

Dual version of classifier:

$$f(\mathbf{x}) = \sum_i^N \alpha_i y_i (\mathbf{x}_i{}^\top \mathbf{x}) + b$$

With the dual classifier, this is where the kernel trick comes in.

$$\begin{aligned} f(\mathbf{x}) &= \sum_i^N \alpha_i y_i \mathbf{x}_i{}^\top \mathbf{x} + b \\ \rightarrow f(\mathbf{x}) &= \sum_i^N \alpha_i y_i \Phi(\mathbf{x}_i)^\top \Phi(\mathbf{x}) + b \end{aligned}$$

Since ***Φ(x)*** only occurs in pairs $\Phi(x_j)^T\Phi(x_i)$, we can define the kernel as:

$k(x_j, x_i) = \Phi(x_j)^T\Phi(x_i)$

With this, the function can be applied without explicitly computing ***Φ(x)***. For example:

$$\Phi : \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \rightarrow \begin{pmatrix} x_1^2 \\ x_2^2 \\ \sqrt{2}x_1x_2 \end{pmatrix} \qquad \mathbb{R}^2 \rightarrow \mathbb{R}^3$$

$$\begin{aligned} \Phi(\mathbf{x})^\top \Phi(\mathbf{z}) &= \left(x_1^2, x_2^2, \sqrt{2}x_1x_2\right) \begin{pmatrix} z_1^2 \\ z_2^2 \\ \sqrt{2}z_1z_2 \end{pmatrix} \\ &= x_1^2z_1^2 + x_2^2z_2^2 + 2x_1x_2z_1z_2 \\ &= (x_1z_1 + x_2z_2)^2 \\ &= (\mathbf{x}^\top \mathbf{z})^2 \end{aligned}$$

So, the kernel trick allows us to operate in the original feature space without computing the coordinates of the data in a higher dimensional space.

Sample Kernels:

Linear	$K(x, x_i) = x^{T} \cdot x_i$
Polynomial	$K(x, x_i) = [\gamma * (x^{T} \cdot x_i) + coef]^d$
RBF	$K(x, x_i) = \exp(-\gamma * \lVert x - x_i \rVert^2)$
Sigmoid	$K(x, x_i) = \tanh(\gamma(x^{T} \cdot x_i) + coef)$

SVM Example in Python:

Recall the steps of building machine learning models from the previous labs:

1. **Analyze** the dataset
2. Perform data **preprocessing** *(including **splitting** into train and test sets)*
3. **Fit** the model to the training data
4. **Evaluate** the fitted model on the test data
5. Generate **predictions** for new data

Note: Support Vector Machine algorithms are not scale invariant, so it is highly recommended to scale the data.

Sklearn offers SVC, NuSVC and LinearSVC which are capable of performing binary and multi-class classification on a dataset. LinearSVC is faster for the case of a linear kernel but it lacks some of the attributes of SVC and NuSVC.

We will use SVC on the “acceptance_data” as follows:

```
# Step 3
from sklearn.svm import SVC

model = SVC() # kernel='linear' or 'rbf' or …
model.fit(X_train, y_train)
```

There are also more hyperparameters that we can set in sklearn's SVC. After the model is fitted, we can print information about the support vectors.

```
# get the number of support vectors for each class
print("Number of support vectors:", model.n_support_)

# get the support vectors
print(model.support_vectors_)

# get the indices of support vectors
print(model.support_)
```

```
Number of support vectors: [14 15]
[[75.01365839 30.60326323]
 [61.83020602 50.25610789]
 [93.1143888  38.80067034]
 [67.94685548 46.67857411]
 [74.78925296 41.57341523]
 [33.91550011 98.86943574]
 [52.34800399 60.76950526]
 [82.36875376 40.61825516]
 [82.22666158 42.71987854]
 [32.57720017 95.59854761]
 [52.10797973 63.12762377]
 [42.07545454 78.844786  ]
 [67.37202755 42.83843832]
 [39.53833914 76.03681085]
 [80.19018075 44.82162893]
 [66.74671857 60.99139403]
 [53.97105215 89.20735014]
 [42.26170081 87.10385094]
 [84.43281996 43.53339331]
 [97.77159928 86.72782233]
 [40.45755098 97.53518549]
 [69.07014406 52.74046973]
 [75.02474557 46.55401354]
 [76.97878373 47.57596365]
 [62.27101367 69.95445795]
 [55.34001756 64.93193801]
 [89.84580671 45.35828361]
 [90.54671411 43.39060181]
 [58.84095622 75.85844831]]
[10 13 24 32 33 38 39 43 50 55 57 72 74 77  2  4  5  7  8 11 16 19 22 25
 27 54 58 65 71]
```

```
print(model.dual_coef_)
print(model.intercept_)
```

```
[[-0.59290189 -1.          -1.          -1.          -1.          -1.
  -1.          -1.          -1.          -1.          -1.          -1.
  -1.          -0.98234082  1.           1.           1.           1.
   1.           0.58183162  1.           1.           1.           1.
   0.04045617  1.           0.95295492  1.           1.         ]]
[-0.08858529]
```

Here, `model.dual_coef_` gets the dual coefficients of the support vector in the decision function multiplied by their targets.

Practice:

Implement SVM using subgradient descent:

- ➢ Create the class **"SVM_GD"** with the necessary attributes.
- ➢ Implement the **"fit", "predict" and "get_cost"** methods.