Machine Learning

Lecture 11

Support Vector Machines (SVM)

Support Vector Machine

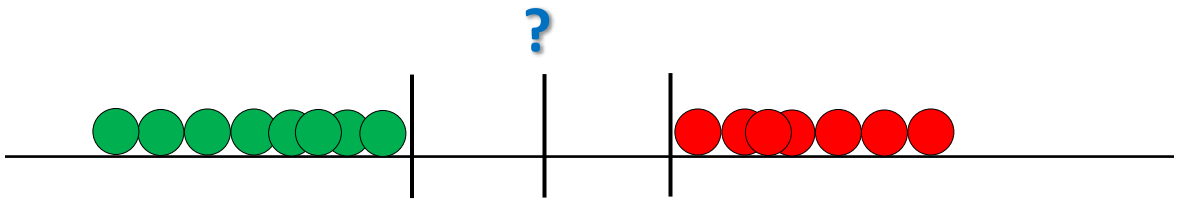
- SVM learning
  - SVM is a learning algorithm that identifies a hyperplane separating the classes.
  - A little history:
    - Introduced by Boser, Guyon & Vapnik in 1992
    - Became popular because of their success in handwritten digit recognition. SVM is now an important and active field of Machine Learning research
    - SVM performs classification by constructing an N-dimensional hyperplane that optimally separates the data into two categories.

Characteristics of SVM

- Learning method: Supervised learning
- Types of SVM: Linear and nonlinear
- Handling of mixed data types: Yes (via encoding)
- Requires linearly separable data? No
- Classification ability: Binary (extension to MC)
- Computationally demanding? Yes (training)
- Ability for online classification: Yes
- Interpretability: Low (mostly a black-box algorithm)
- When to consider: image recognition (handwriting, face) and classification, bioinformatics, advertisement display, text classification, image-based gender detection

Intuition for SVM

- Suppose we have one-dimensional data (a feature and a binary target):

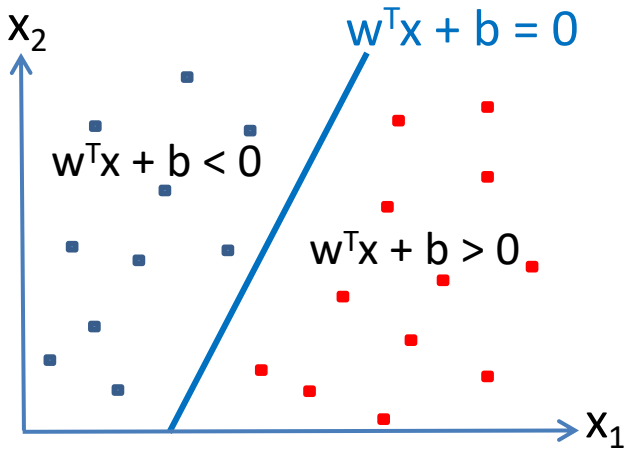


- Where would you put the decision boundary that separates the two classes?
- What is the most optimum threshold here?
- We can identify the instances on the edge of each group and use the **midpoint** between them as a threshold.
- The shortest distance between the threshold and the 1st instances is called the **margin**. The midpoint provides the largest margin possible.
- This way, we get a **maximal margin classifier**.

Separating hyperplane

- A **perceptron** for a linearly separable data set:

Decision: sgn($\mathbf{w}^T\mathbf{x}$)

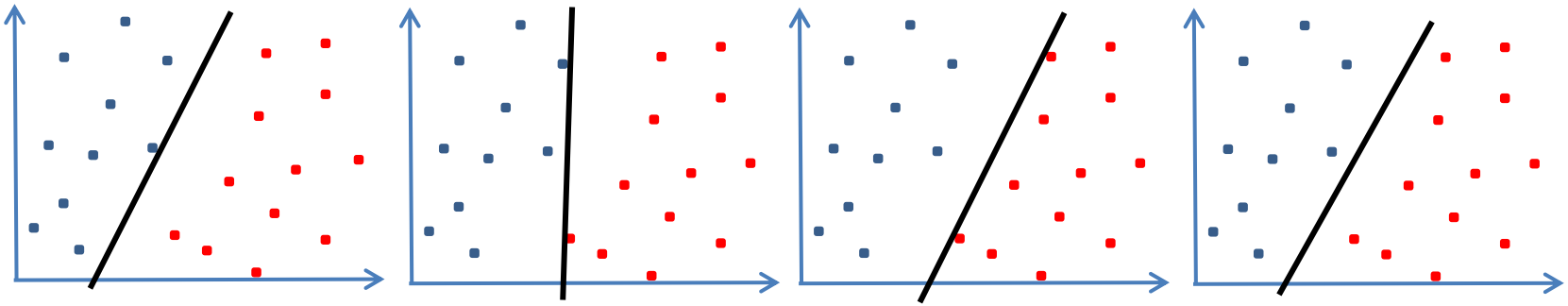


$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$\mathbf{w}^T\mathbf{x} + b = w_1 x_1 + w_2 x_2 + b = 0$$

- Is this hyperplane unique?
- Is it the optimal one?

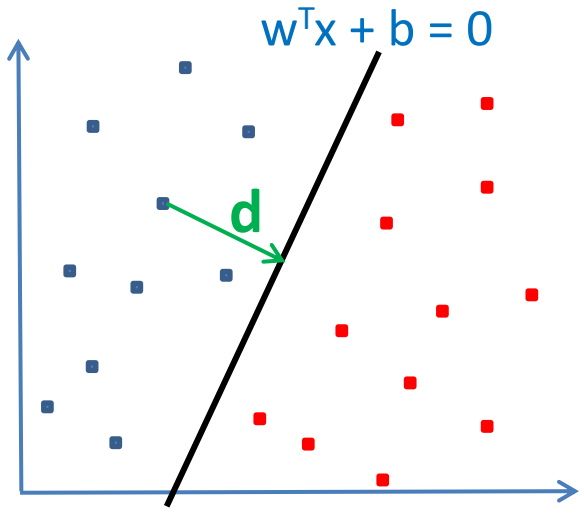
Which of the linear separators is the best (optimal)?



Need to define an optimization problem!

Separating hyperplane – cont'd

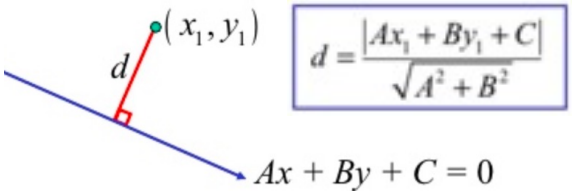
Basic definitions:



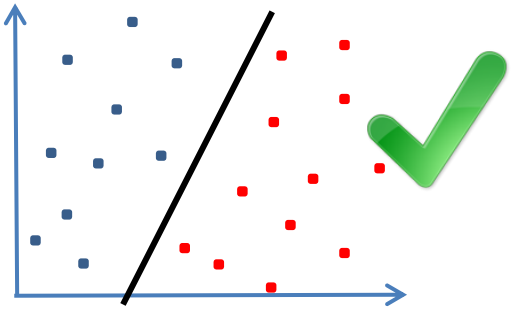
Margin of a point: **d**

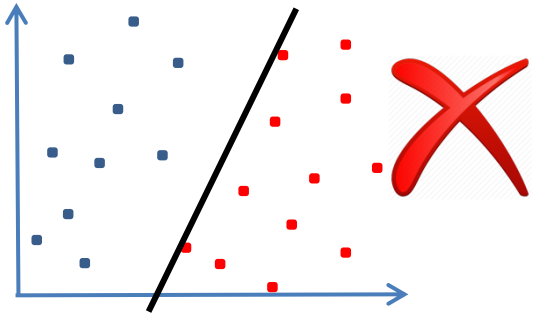
This is the perpendicular distance from the data point to the separator which defines the degree to which this data point is correctly classified.

$$d = \frac{\mathbf{w}^T \mathbf{x} + b}{\|\mathbf{w}\|}$$

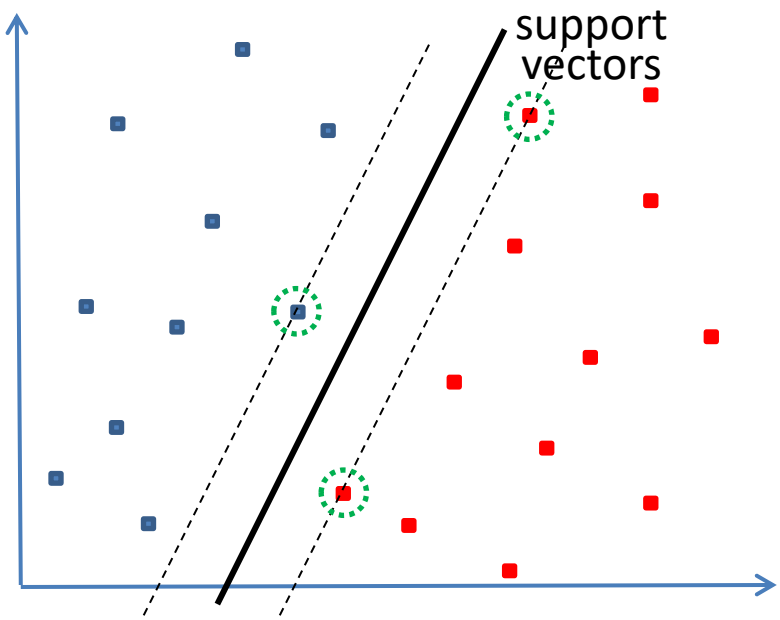


The goal here is to maximize the margin widening the distance between the data points and the separator.





SVM picks the best separating hyperplane based on the maximum margin (**Max. Margin Classification**)



Maximum-margin separator is determined by the closest data points called the "support vectors" as identified by the symbol ◌.

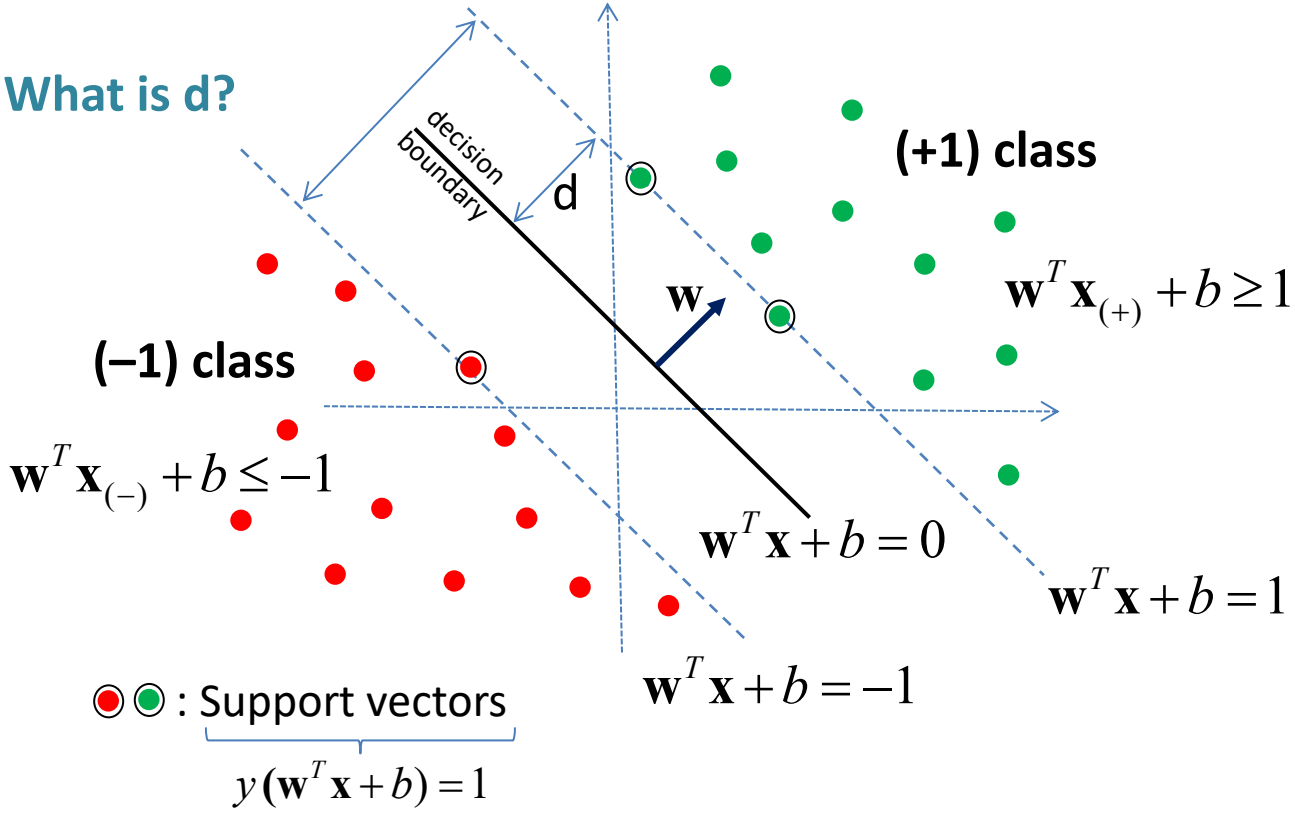
Define the margin as the width that the boundary could be increased by before hitting a data point.

A margin as fat as possible is also in line with intuition (more room for correct classification)

Notes:

• In a SVM learning algorithm, among all training data, only the support vectors are important in determining the separator. Data points beyond the separating lines (or hyperplanes in dimensions greater than 2) have no effect.

- Optimization problem for linearly separable data



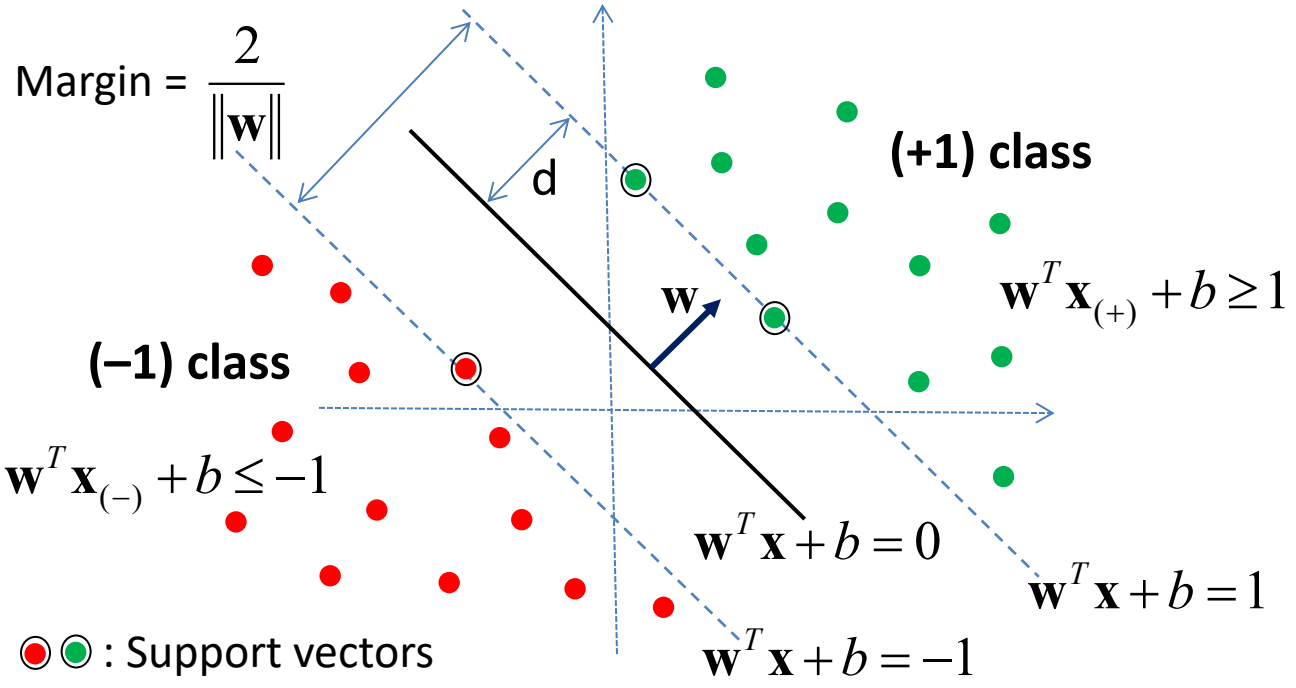
If $\mathbf{x}_{dc}$ is on the decision boundary, then: $\mathbf{w}^T\mathbf{x}_{dc}+b=0$

So, $\mathbf{w}^T\left(\mathbf{x}_{dc}+d\frac{\mathbf{w}}{\|\mathbf{w}\|}\right)+b=1 \Rightarrow \mathbf{w}^T\mathbf{x}_{dc}+d\frac{\mathbf{w}^T\mathbf{w}}{\|\mathbf{w}\|}+b=1 \Rightarrow d=\frac{1}{\|\mathbf{w}\|}$

$\mathbf{w}^T\mathbf{x}_{dc}+b=0$

SVM basics – cont'd

- Optimization problem for linearly separable data



$$d = \frac{1}{\|\mathbf{w}\|}$$

Maximize $\frac{1}{\|\mathbf{w}\|}$ subject to $\mathbf{w}^T\mathbf{x}_{(+)} + b \geq 1$ for points in class (+) and

$\mathbf{w}^T\mathbf{x}_{(-)} + b \leq -1$ for points in class (–)

Maximizing $\frac{1}{\|\mathbf{w}\|} \Rightarrow$ Minimizing $\|\mathbf{w}\|$ or $\frac{1}{2}\|\mathbf{w}\|^2$ (mathematical convenience)

subject to $\begin{array}{ll} \mathbf{w}^T\mathbf{x}_{i,(+)} + b \geq 1 & when \quad y_i = +1 \\ \mathbf{w}^T\mathbf{x}_{i,(-)} + b \leq -1 & when \quad y_i = -1 \end{array}$ for all points *i=1,2,.....,N*

SVM basics – cont'd

$\frac{1}{2}\|\mathbf{w}\|^2$ minimized such that $y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1$ for all instances *i=1,…,N*

- This is an optimization problem with a constraint which could be solved by the Lagrange formulation (primal form).
- Lagrange formulation in the primal form:

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

$$= \frac{1}{2}\mathbf{w}^T\mathbf{w} - \sum_{i=1}^{N} \lambda_i y_i(\mathbf{w}^T\mathbf{x}_i + b) + \sum_{i=1}^{N} \lambda_i$$

$$\frac{\partial L_P}{\partial b} = 0 \Rightarrow \sum_i \lambda_i y_i = \mathbf{0}$$

$$\frac{\partial L_P}{\partial w} = 0 \Rightarrow \mathbf{w} = \sum_i \lambda_i y_i \mathbf{x}_i$$

The only instances that contribute to the definition of the margin itself are the support vectors. Since w is part of the definition of the margin, **for any data point that is not a support vector, corresponding λ_i should be zero**.

- By plugging these back into the equation for L_P ,we get the dual form (Kuhn-Tucker theorem):

$$L_D = \sum_i \lambda_i - \frac{1}{2}\sum_i \sum_j \lambda_i \lambda_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j$$ which is free of the terms w and b.

SVM basics – cont'd

- As opposed to minimizing over w and b subject to constraints involving λ's as in the previous phase, we'll try now to maximize over λ subject to w and b.
- Find λ_i
  such that: $L_D = \sum_i \lambda_i - \frac{1}{2}\sum_i \sum_j \lambda_i \lambda_j y_i y_j (\mathbf{x}_i^T \mathbf{x}_j)$ (simple quadratic form for λ)
  is maximized subject to $\sum_i \lambda_i y_i = \mathbf{0}$ and $\lambda_i \geq 0$ for all *i*
- This is still a constrained optimization problem and it turns out that this maximization depends only on the **inner products** of pairs of training points which will later come handy in the solution of non-linearly separable problems.
  Maximizing L_D above is the same as minimizing this expression over λ:

$$Min_{\lambda \geq 0} \Rightarrow \frac{1}{2}\sum_i \sum_j \lambda_i \lambda_j y_i y_j (\mathbf{x}_i^T \mathbf{x}_j) - \sum_i \lambda_i$$

SVM basics – cont'd

The solution (quadratic programming):

$$Min_{\lambda} \Rightarrow \frac{1}{2}\boldsymbol{\lambda}^{T}\begin{bmatrix} y_1y_1\mathbf{x}_1{}^{T}\mathbf{x}_1 & y_1y_2\mathbf{x}_1{}^{T}\mathbf{x}_2 & \dots & y_1y_N\mathbf{x}_1{}^{T}\mathbf{x}_N \\ y_2y_1\mathbf{x}_2{}^{T}\mathbf{x}_1 & y_2y_2\mathbf{x}_2{}^{T}\mathbf{x}_2 & \dots & y_2y_N\mathbf{x}_2{}^{T}\mathbf{x}_N \\ \dots & \dots & \dots & \dots \\ y_Ny_1\mathbf{x}_N{}^{T}\mathbf{x}_1 & y_Ny_2\mathbf{x}_N{}^{T}\mathbf{x}_2 & \dots & y_Ny_N\mathbf{x}_N{}^{T}\mathbf{x}_N \end{bmatrix}\boldsymbol{\lambda} + (-1)^{T}\boldsymbol{\lambda}$$

subject to $\mathbf{y}^{T}\boldsymbol{\lambda} = 0$ and $0 \le \boldsymbol{\lambda} \le \infty$

Numerical solution of this problem yields ➔ λ_i

(each non-zero λ_i indicates that point ***i*** is a support vector)

We can calculate w and b now:

$$\mathbf{w} = \sum_i \lambda_i y_i \mathbf{x}_i \quad \text{and} \quad b = y_n \mathbf{w}_n{}^{T}\mathbf{x}_n \quad \text{such that} \quad \lambda_n \neq 0$$

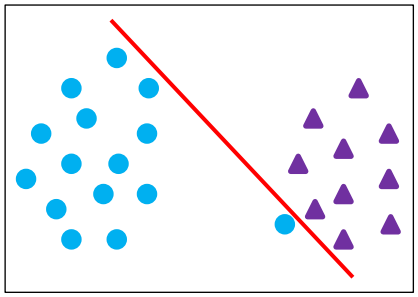
So, the decision rule will turn out to be:

$$\operatorname{sgn}\left(\sum_i \lambda_i y_i (\mathbf{x}_i{}^{T}\mathbf{u}) + b\right) = \begin{cases} + & class(+) \\ - & class(-) \end{cases}$$

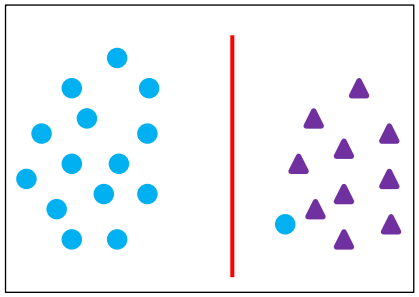
where $\boldsymbol{x_i}$ is the support vector and ***u*** is the (unknown) test point

Soft margin classification

- Which one is better?



Data linearly separable by only a very narrow margin



Large margin separation is better in case of minor misclassifications (watch for outliers)

- Tolerance for misclassifications
  - Hard margin classification requires that all data points are classified correctly (which is what we've seen before)
  - Soft margin classification allow misclassification on difficult/noisy data points
- Is there a tradeoff between the margin size and the number of misclassifications that could be allowed in the training data?

Soft-margin SVM

- Consider the case that your data isn't perfectly linearly separable, and you want to allow some data points of one class to appear on the other side of the boundary (misclassified) for better generalization.
- If the data isn't linearly separable, we introduce a penalty:

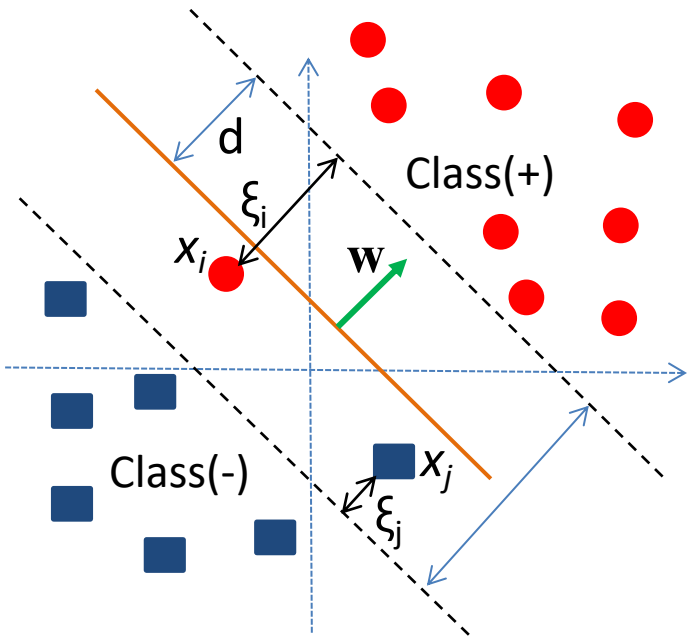
$$\text{argmin}_{w,b} = \frac{1}{2}\|w\|^2 + C \text{ x } (\#\text{ of mistakes})$$

A hyperparameter that decides the trade-off between maximizing the margin and minimizing the mistakes

- **Objective**: Maximum margin with the smallest number of mistakes possible.
- When C is small, classification mistakes are given less importance and focus is more on maximizing the margin, whereas when C is large, the focus is more on avoiding misclassification at the expense of keeping the margin small
- How do we penalize mistakes?

Soft-margin SVM

- As not all mistakes are equally bad, we shall use a margin called "**slack variables**" to penalize mistakes.
- We introduce, what we call the **slack variables** – an $\xi_i \geq 0$ for each x_i:



Our optimization problem becomes:

$$\min_{\mathbf{w},b,\xi} = \frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} + \mathbf{C}\sum_{i=1}^{N}\xi_i$$

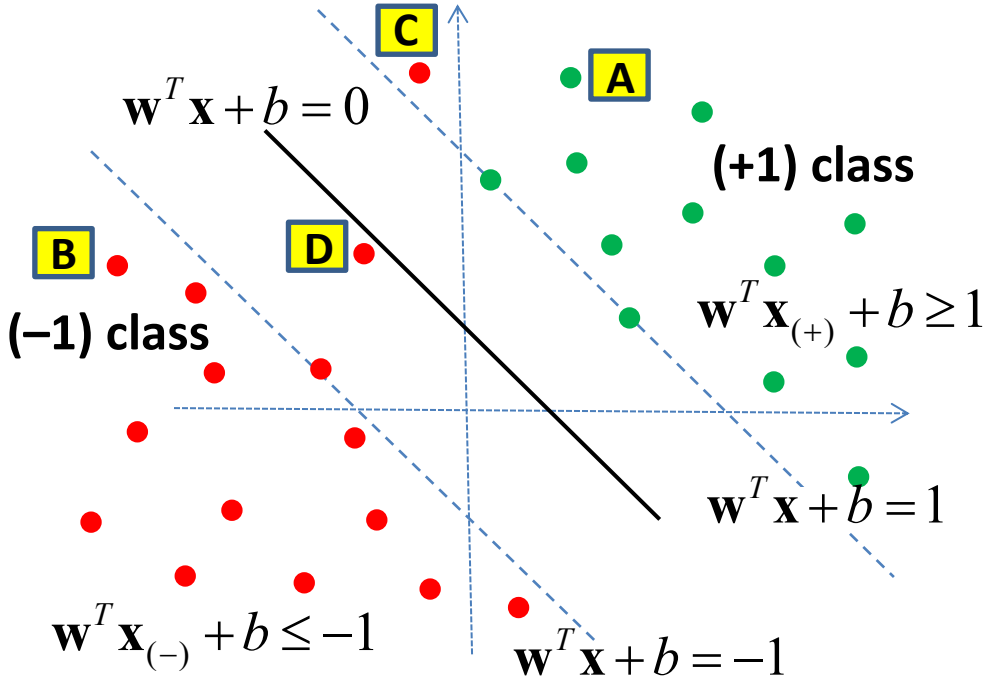
C represents how much of the error is tolerable. If C is small, the penalty for misclassified points is low so a decision boundary with a large margin is chosen at the expense of a greater number of misclassifications.

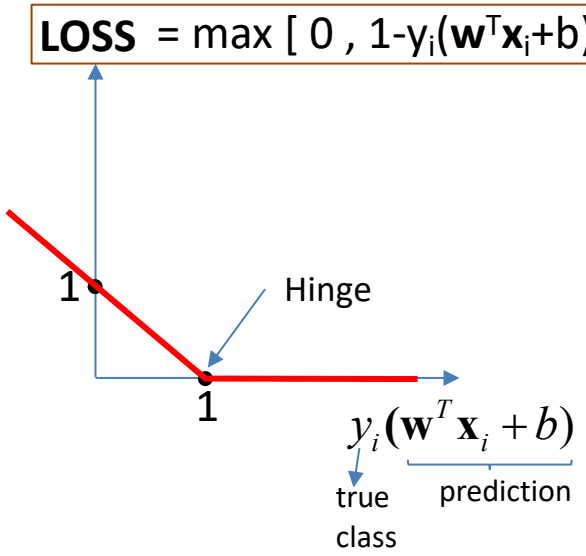
As C goes to infinity, the penalty for having non-zero ξ_i goes to infinity, and thus we force the ξ_i's to be zero, which is exactly the setting of the hard-margin SVM.

- How to find the best C? Via cross validation…
- But first, where is this coming rom?

Soft margin classification

- Concept of Hinge-Loss





A: max [0 , 1-1(>1)] = 0
B: max [0 , 1+1(<-1)] =0
C: max [0 , 1+1(>1)] = >1
D: max [0 , 1+1(bw 0 & -1)] = [0,1]

We're on the right side of the decision boundary, but we're within the margin

- Soft-margin SVM looks for the minimum of the following:

$$\underset{w,b}{\mathbf{Min}}\left\{\frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{N}\left[y_i(\mathbf{w}^T\mathbf{x}_i+b)-1\right]\right\}$$

C controls the tradeoff between maximizing the margin & minimizing the loss

Soft margin classification

$$\underset{w,b}{\mathbf{Min}}\left\{\frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{N}\left[y_i(\mathbf{w}^T\mathbf{x}_i + b) - 1\right]\right\}$$

Max margin Mistakes over training instances

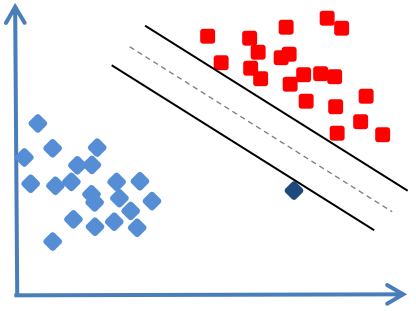
- To minimize this loss, we would like to make $y\mathbf{w}^T\mathbf{x}$ large, which happens when points are far off on the correct side of the decision boundary.
- C determines how aggressively the classifier tries to prevent misclassifications. Hard margin SVMs do not allow for misclassifications and do not require regularization.
- If we set C to a large number, then the SVM will pursue outliers more aggressively, which potentially comes at the cost of a smaller margin and may lead to overfitting on the training data.

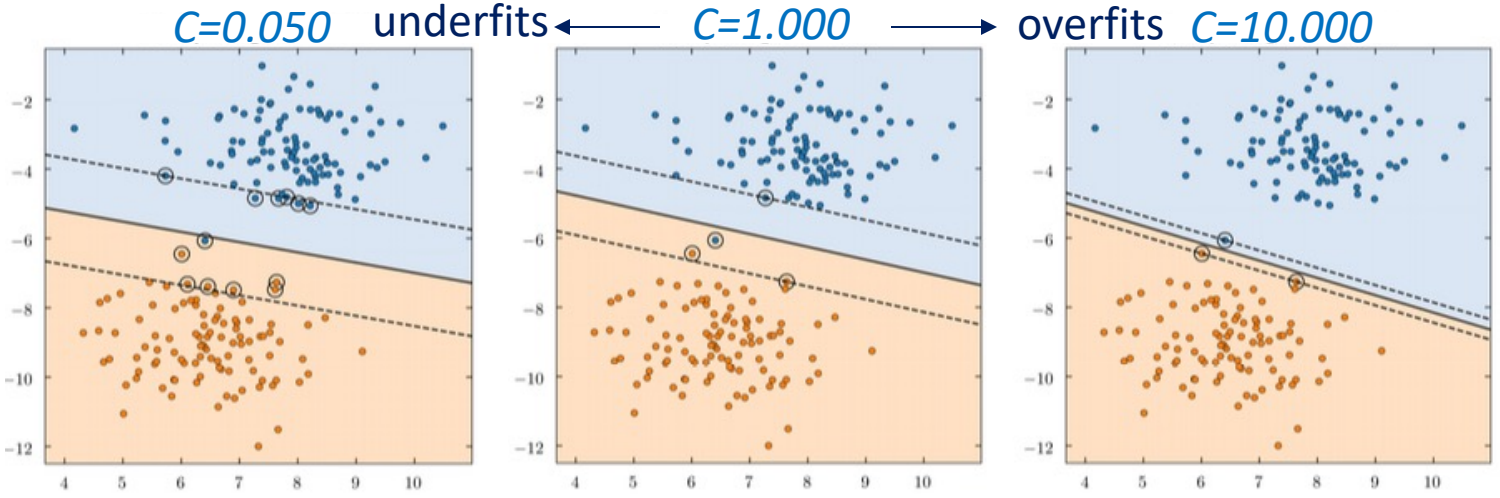
$$\underset{w,b}{\mathbf{min}}\left\{\frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{N}\xi_i\right\} \quad \mathbf{s.t.}\ \ y_i(\mathbf{w}^{\mathbf{T}}\mathbf{x}_i + b) \geq 1 - \xi_i$$

$$\mathbf{and} \quad \xi_i \geq 0 \quad \forall\, \mathbf{data\ point}\ \mathbf{x}_i$$

Soft-margin SVM – cont'd

- Could prefer soft-margin SVM's even for the linearly separable data. SVM is sensitive to outliers and a single outlier can determine the boundary for a hard-margin SVM.
- For hard margin SVM, support vectors are the points which are "on the margin". In the picture below, C=10000 is pretty close to hard-margin SVM, and you can see the circled points are the ones that will touch the margin (margin is essentially the same as the separating hyperplane)





Multi-class SVM

- SVM is natively a binary classifier where a class label can take on 2 values only (+/-, yes/no, etc)
- What to do for problems containing more than 2 classes?
- Pairwise classification
  - For an M-class problem, train each pair of classes together (i.e. train the classifier for one class versus all the rest)
  - Have $\binom{M}{2} = \frac{M!}{2!(M-2)!}$ classifiers

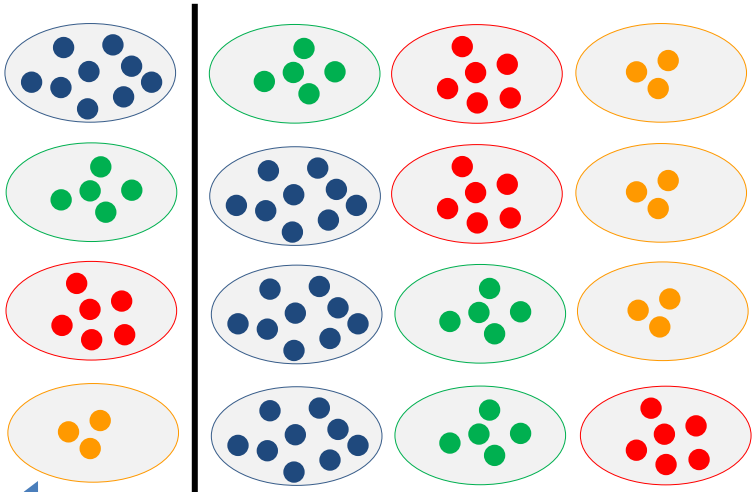
    SVM_1 classifies as "$output_1$" vs "the rest"

    SVM_2 classifies as "$output_2$" vs "the rest" and so on...
  - To predict the output for a new input x, predict with each SVM_i and count the number of times x was assigned to that class
  - Class label with the highest count determines the class that x belongs to

Multi-class SVM – cont'd

- Multi-class comparison

OVR: One vs Rest

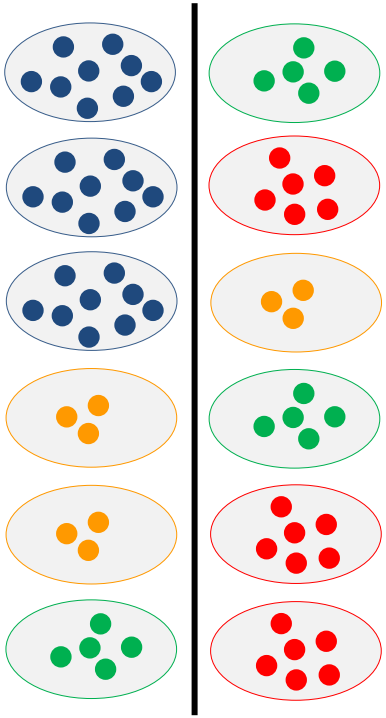


k comparisons (k: # of classes)

Pros : Fewer classifications

Cons: Classes may be imbalanced

OVO: One vs One



C(k,2)=k!/2!(k-2)! comparisons

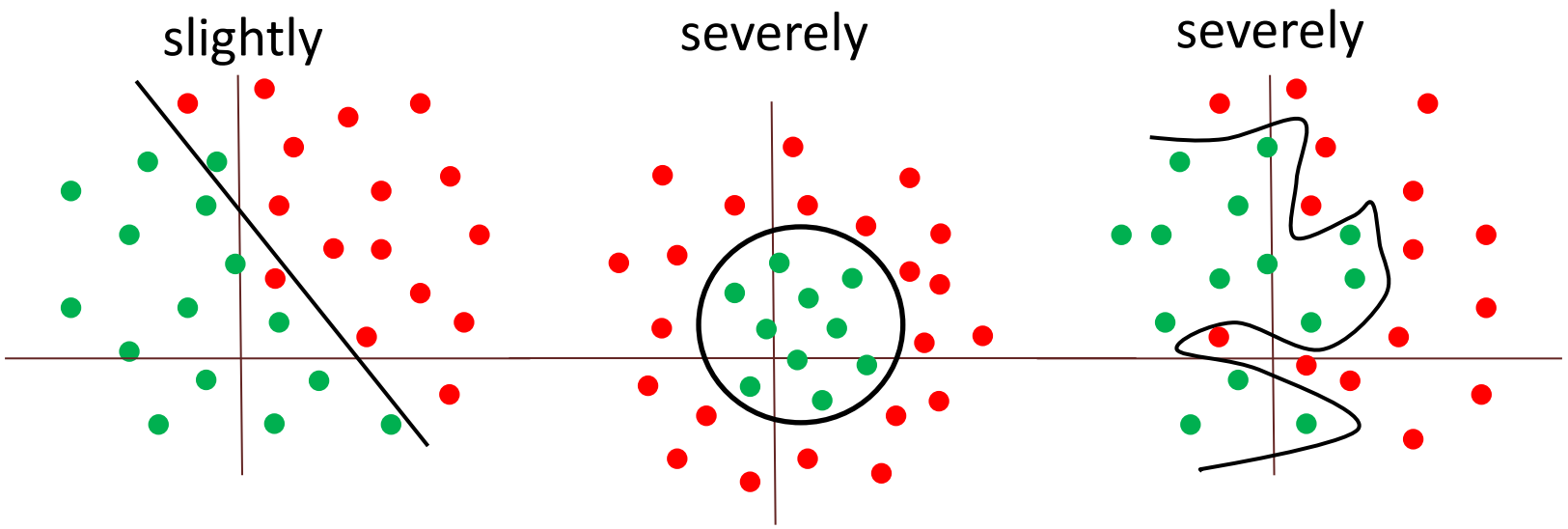
Pros : Less sensitive to imbalance

Cons: More classifications

Note: '**ovr**' is the default for multi-class problems in scikit-learn

Linearly inseparable data

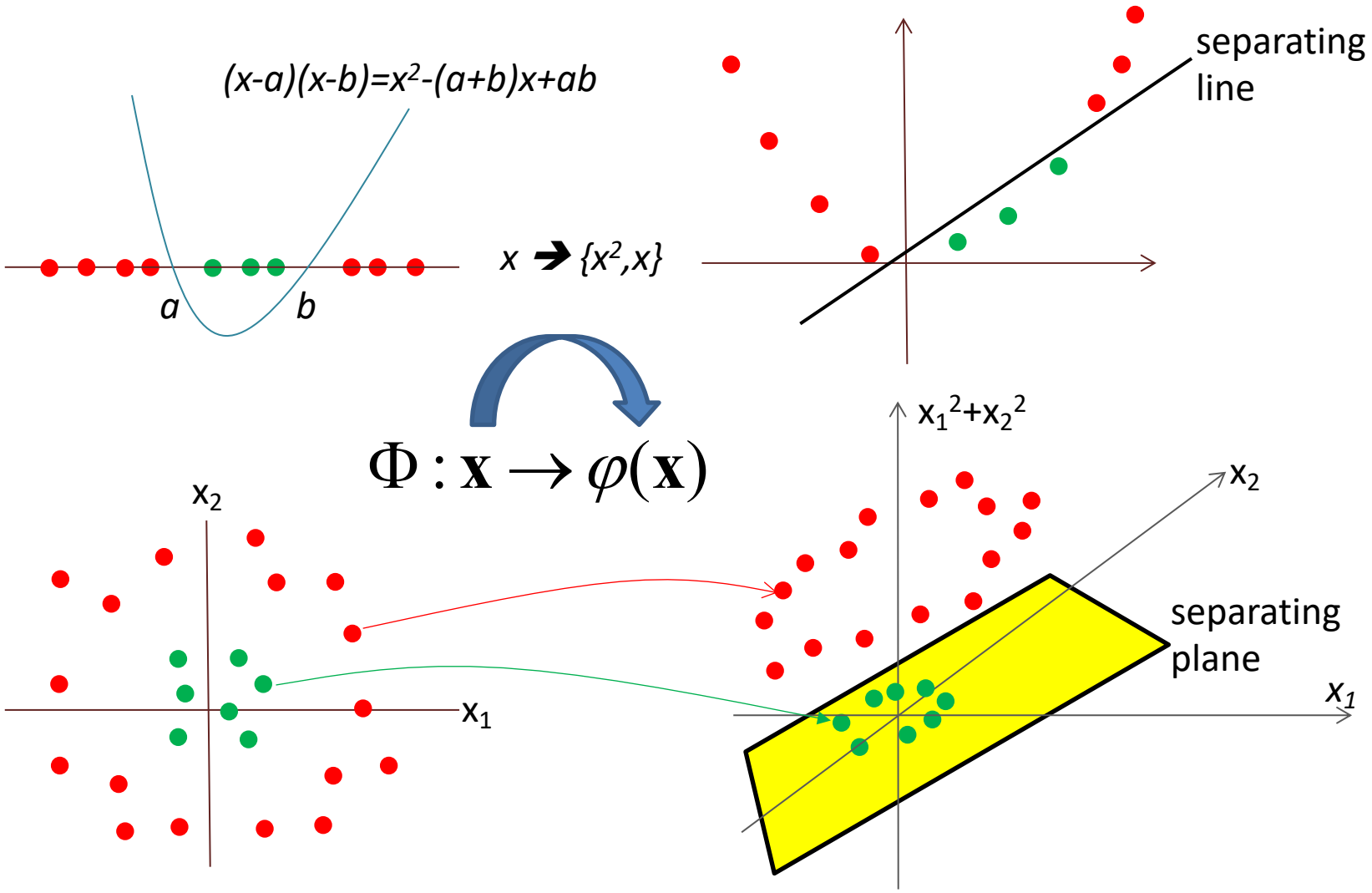
- What if the data is noisy (not linearly separable)?



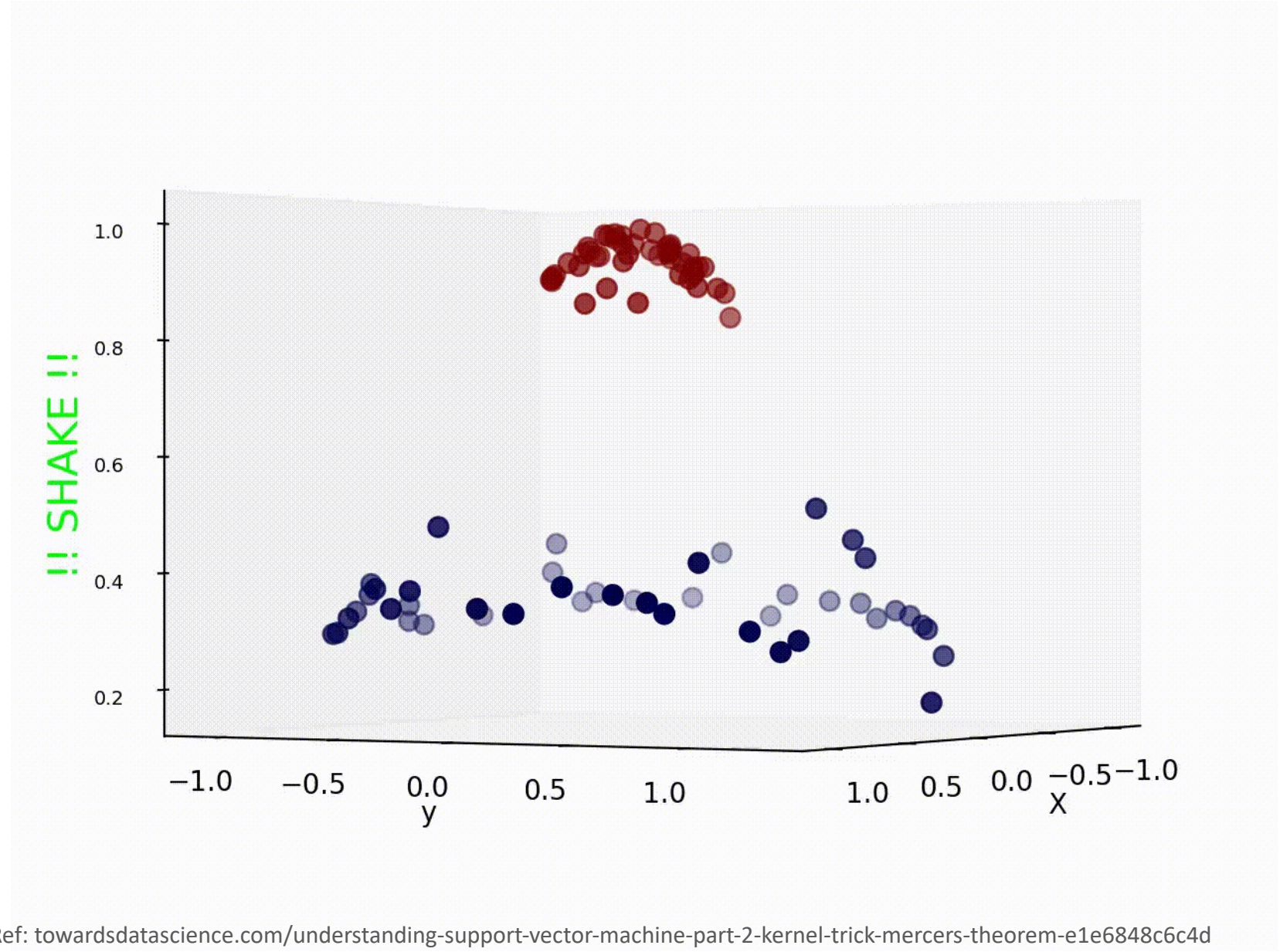
- **Solution 1:** Utilize a soft margin classification
  - Introduce an error term (slack variables) to allow the misclassification of difficult or noisy data points (hence the name soft margin)
- **Solution 2:** Use powerful kernel functions
  - Map the data into a different space where a linear separator can be used to separate the data points

Using a mapping function

The original **input space** is mapped into a higher-dimensional **feature space** where the dataset becomes linearly separable



Kernel trick – cont'd



Ref: towardsdatascience.com/understanding-support-vector-machine-part-2-kernel-trick-mercers-theorem-e1e6848c6c4d

Kernel trick

- The main idea behind dealing with linearly inseparable data is to create nonlinear combinations of the original attributes to project them onto a higher dimensional space through a mapping function called the **kernel**.

$$K(x_1,x_2) = (d_1, d_2, d_3) = (x_1, x_2, x_1^2+x_2^2)$$

- This projection transforms a 2-dimensional data set to a 3-dimensional one and creates a hyperplane that separates the classes (see previous example).
- This is an expensive procedure for problems with a high dimensional feature space as it involves the dot products of vectors.
- So we use a technique called **kernel trick** to replace the dot product between points with a kernel function. Kernels are a way of computing the inner product of 2 vectors of x and y in some vector space.

Kernel trick

- When data is not separable in input space, we can map the data into some other (high dimensional) feature space where data is linearly separable.
- Suppose you want to apply a 2nd degree polynomial transformation to a 2D data set, then train a linear SVM classifier on the transformed data set.
- Coordinates of the 2 data points in the original 2D feature space: $\mathbf{x} = (x_1, x_2)$ and $\mathbf{y} = (y_1, y_2)$
- Assume we need to map x and y to higher dimensional space by employing a 2nd degree polynomial mapping function $\Phi(x)$ such that

$$\phi : \sim^2 \Rightarrow \sim^m \qquad \phi(\mathrm{x}) = \begin{pmatrix} x_1^2 \\ x_1 x_2 \\ x_2 x_1 \\ x_2^2 \end{pmatrix}, \quad \phi(\mathrm{y}) = \begin{pmatrix} y_1^2 \\ y_1 y_2 \\ y_2 y_1 \\ y_2^2 \end{pmatrix}$$

We need these for the transformed space

Kernel trick

- So the inner product of these functions yields:

$$\phi(\mathrm{x})^T \phi(\mathrm{y}) = \begin{pmatrix} x_1^2 & x_1x_2 & x_2x_1 & x_2^2 \end{pmatrix} \begin{pmatrix} y_1^2 \\ y_1y_2 \\ y_2y_1 \\ y_2^2 \end{pmatrix} = x_1^2y_1^2 + 2x_1x_2y_1y_2 + x_2^2y_2^2$$

just a scalar

- If we use a proper kernel function instead of carrying out the complicated calculations in the higher dimensional space, we can get the same result with much less computational cost:

$$K(\mathrm{x},\mathrm{y}) = [\mathrm{x}^{\mathrm{T}}\mathrm{y}]^2 = \left[\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \right]^2 = (x_1y_1 + x_2y_2)^2 = x_1^2y_1^2 + 2x_1y_1x_2y_2 + x_2^2y_2^2$$

- The dot product of the transformed vectors is equal to the square of the dot product of the original vectors:

$$\boxed{\phi(\mathrm{x})^T \phi(\mathrm{y}) = [\mathbf{x}^T\mathbf{y}]^2}$$

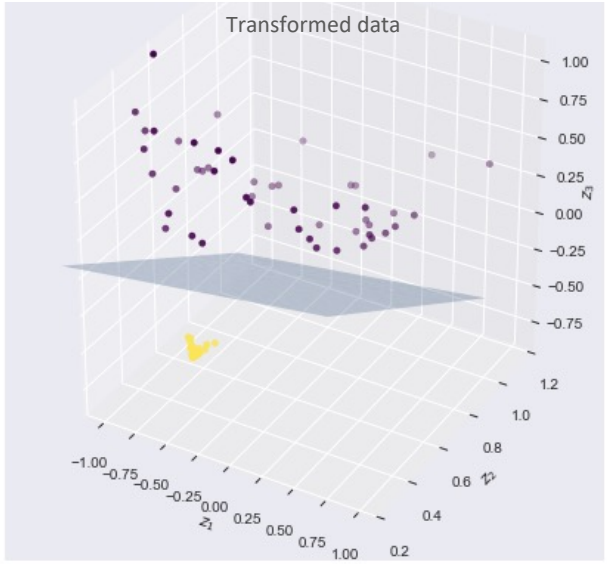
- This is useful because it allows us to operate in the original feature space without computing the coordinates of the data in the higher dimensional (transformed) space.

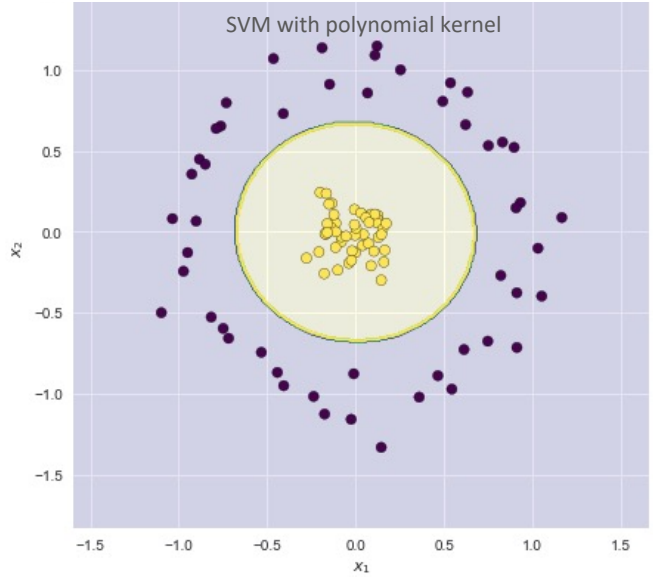
Kernel trick

- We saw that the output of the Kernel function K(x,y) is equal to $\Phi(x)^T\Phi(y)$ where

$$\phi(\mathbf{x}) = \phi\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{pmatrix} \qquad \phi(\mathbf{y}) = \phi\left(\begin{pmatrix} y_1 \\ y_2 \end{pmatrix}\right) = \begin{pmatrix} y_1^2 \\ \sqrt{2}y_1y_2 \\ y_2^2 \end{pmatrix}$$

- which computes a dot product in 3 dimensional space without explicitly visiting that space. Here is the feature map:





Ref: https://stats.stackexchange.com/questions/152897/how-to-intuitively-explain-what-a-kernel-is-

Kernels available

- Examples of Kernel functions: $\Phi : \mathrm{x} \rightarrow \phi(\mathrm{x})$
- Linear: $K(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^T \mathbf{x}_j$
- Polynomial of degree d: $K_d(\mathbf{x}_i, \mathbf{x}_j) = \left[\gamma(\mathbf{x}_i^T \mathbf{x}_j) + r\right]^d$
- Sigmoid: $K_s(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{tanh}\left[\gamma(\mathbf{x}_i^T \mathbf{x}_j) + r\right]$
- Gaussian (**RBF** : radial basis functions – default):

$$K_{RBF}(\mathbf{x}_i, \mathbf{x}_j) = \exp\left[\frac{-\left\|\mathbf{x}_i - \mathbf{x}_j\right\|^2}{2\sigma^2}\right] = \exp\left[-\gamma\left\|\mathbf{x}_i - \mathbf{x}_j\right\|^2\right]$$

r: Independent term in Kernel (default=0) for polynomial and sigmoid

ϒ: Kernel coefficient for polynomial, sigmoid and rbf

$$\text{default } \gamma = \frac{1}{\text{n\_features Var}(X)}$$

Kernels available

- Despite the innocent looking formula, RBF Kernel is indeed quite a powerful kernel.

$$K_{RBF}(\mathbf{x}_i, \mathbf{x}_j) = \exp\left[-\frac{1}{2}\left\|\mathbf{x}_i - \mathbf{x}_j\right\|^2\right]$$

- This could be re-written in the following form:

$$e^{-\frac{1}{2}\|\mathbf{x}_i-\mathbf{x}_j\|^2} = e^{-\frac{1}{2}\left[(\mathbf{x}_i-\mathbf{x}_j)^T(\mathbf{x}_i-\mathbf{x}_j)\right]} = e^{-\frac{1}{2}\left[\mathbf{x}_i^T\mathbf{x}_i-2\mathbf{x}_i^T\mathbf{x}_j+\mathbf{x}_j^T\mathbf{x}_j)\right]}$$

$$= e^{-\frac{1}{2}\left[\mathbf{x}_i^T\mathbf{x}_i+\mathbf{x}_j^T\mathbf{x}_j\right]}e^{\mathbf{x}_i^T\mathbf{x}_j}$$

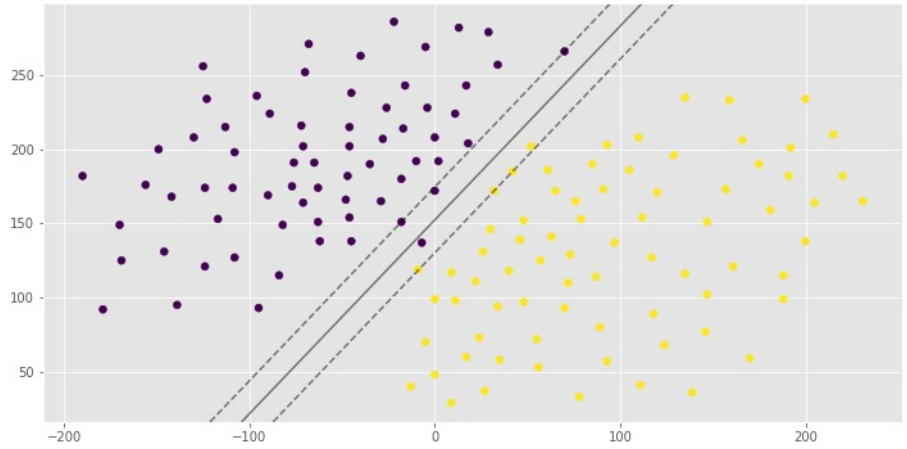
$$K_d(\mathbf{x}_i, \mathbf{x}_j) = \left[(\mathbf{x}_i^T\mathbf{x}_j)+1\right]^d$$

$$= \mathbb{C}e^{\mathbf{x}_i^T\mathbf{x}_j+1}e^{-1} = \mathbb{C}^*e^{\mathbf{x}_i^T\mathbf{x}_j+1} = \mathbb{C}^*\sum_{d=0}^{\infty}\frac{(1+\mathbf{x}_i^T\mathbf{x}_j)^d}{d!} \qquad e^x = \sum_{n=0}^{\infty}\frac{x^n}{n!}$$

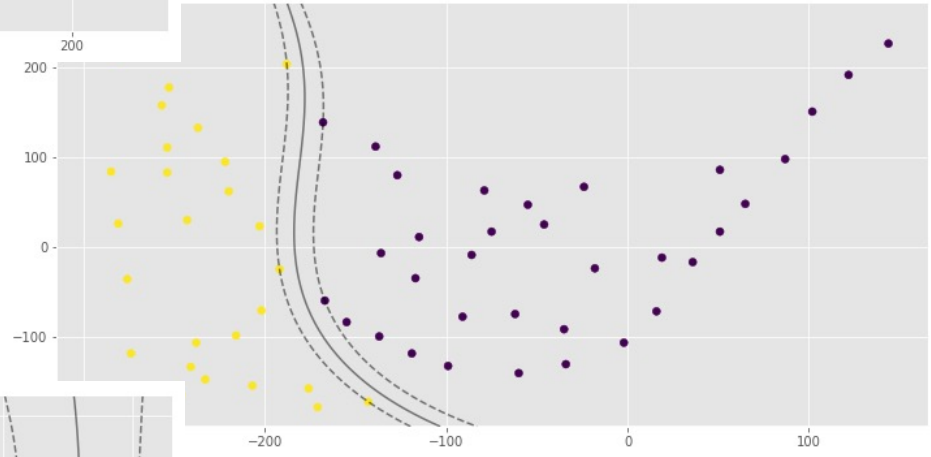
- This is the polynomial kernel of degree "d". So a RBF kernel can be expressed as an infinite sum of polynomial kernels which represents infinite degree interactions between our original data. This is where the power of the RBF kernels come from.

Kernel trick – cont'd

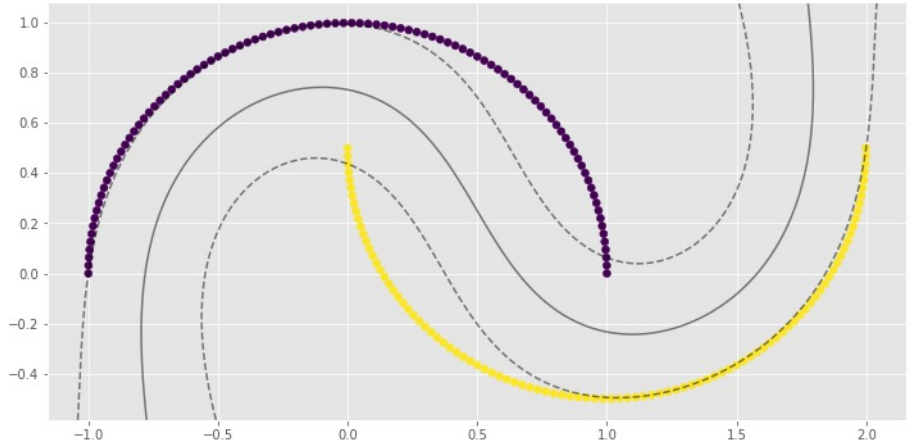
Linear Kernel



Polynomial Kernel



Gaussian Kernel



SVM settings

- Selecting the best SVM settings
  1. Code data in the numerical format needed for SVM
  2. Scale each attribute to have a range 0 to 1, or to have zero mean and unit variance
  3. Use CV to find the best value for C for a linear kernel
  4. Use cross-validation to find the best values for C and gamma (ϒ) for an RBF kernel
  5. Train on the entire available data using the parameter values found to be best via CV
- It is reasonable to start with C = 1 and ϒ = 1, and to try values that are smaller or larger by a factor of 2:

  $< C , \Upsilon > = \{... , 2^{-3}, 2^{-2}, 2^{-1}, 1, 2, 2^2, 2^3, ...\}$ => Grid search

SVM using scikit-learn

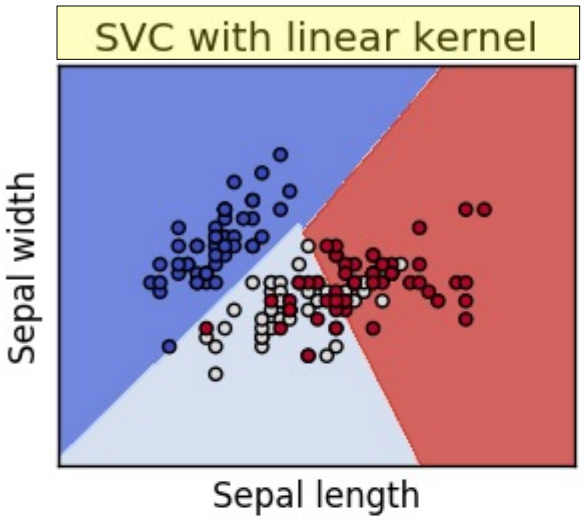
- Determining parameters via GridSearch

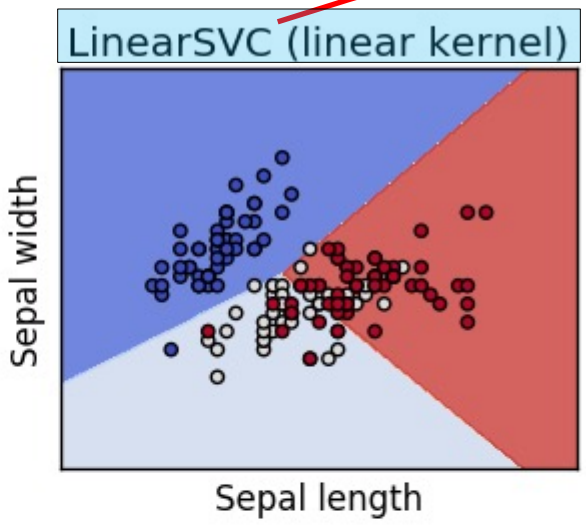
```
parameters = [{'kernel': ['rbf'],
               'gamma': [1e-4, 1e-3, 0.01, 0.1, 0.2, 0.5],
               'C': [0.1, 1, 10, 100, 1000]},
              {'kernel': ['linear'],
               'C': [0.1, 1, 10, 100, 1000]}
             ]
clf = GridSearchCV(svm.SVC(decision_function_shape='ovr'),
                   parameters, cv=5)
clf.fit(X_train, y_train)
```

Higher the value of gamma (γ), more exact fit for the training data set, limiting the generalization ability and causing overfitting.
C allows you to decide how much you want to penalize misclassified points (a tradeoff between smooth decision boundary and classifying the training points correctly.

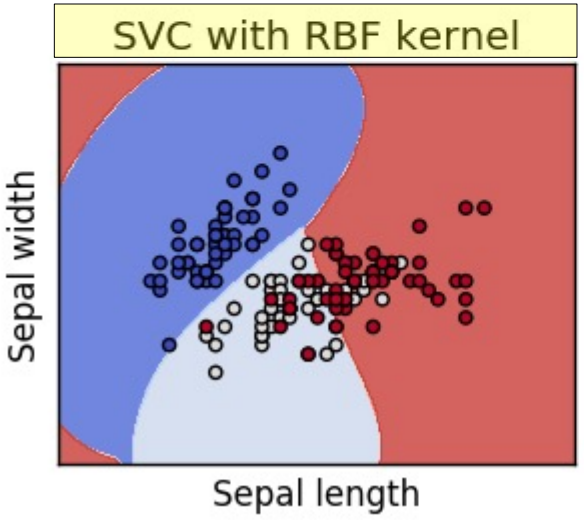
C: Penalty parameter of the error term (default value 1.0)
kernel: The kernel type used => linear, rbf, poly, or sigmoid (default is rbf)
gamma: Kernel coef. for 'rbf', 'poly', and 'sigmoid' (default is 1/n_features)
decision_function_shape: One-vs-Rest (ovr, default) or One-vs-One (ovo)

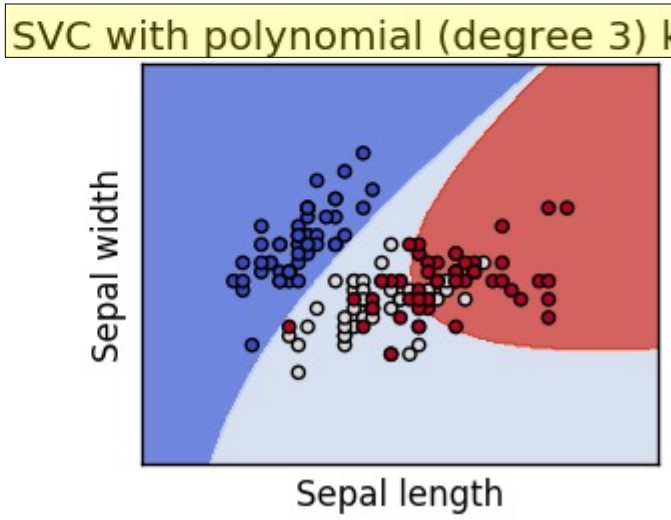
Kernel effect on SVM





Implemented in terms of **liblinear** (as opposed to libsvm for SVM with a linear kernel), has more flexibility in the choice o floss functions and penalties and scales better to large number of samples.



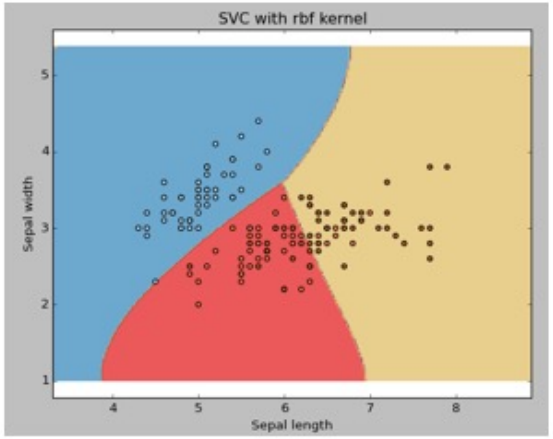


Source: http://scikit-learn.org/stable/auto_examples/svm/plot_iris.html

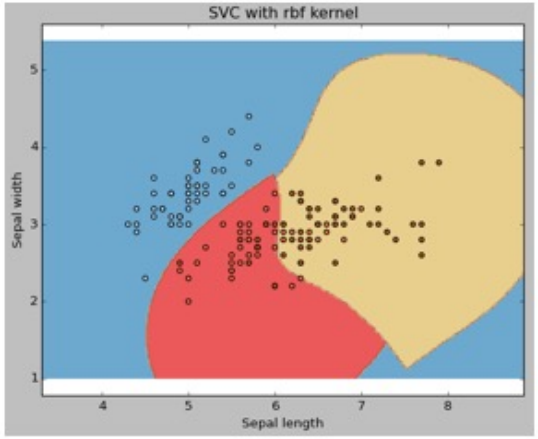
Effect of C and Gamma values

C: Penalty parameter C of the error term. It also controls the trade off between smooth decision boundary and classifying the training points correctly

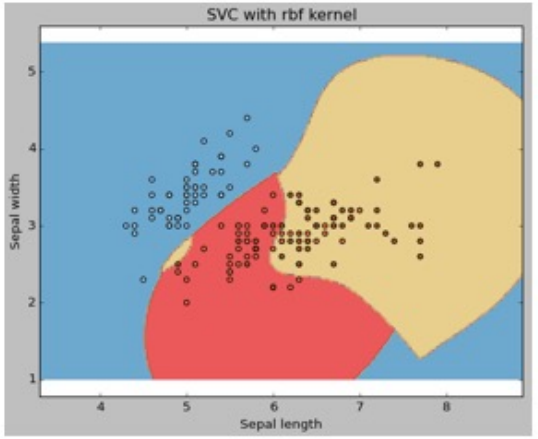
c =1 **(soft/large margin hyperplane)**



C =100

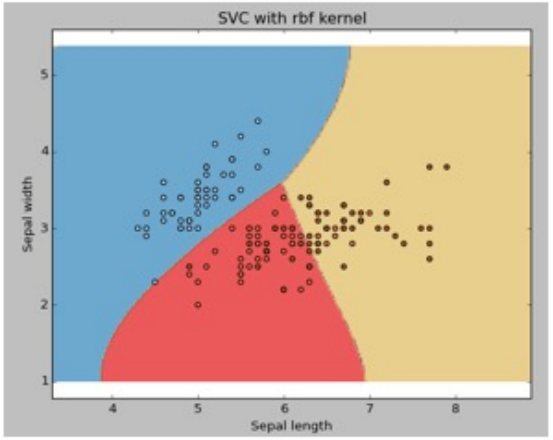


c =1000 **(hard margin)**

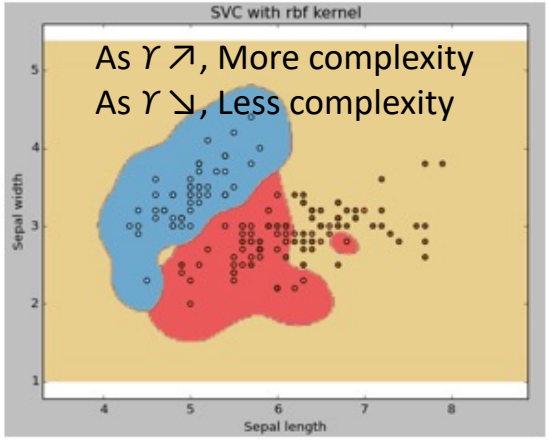


gamma: As it gets higher, classifier will try an exact fit for the training data set limiting the generalization ability (over-fitting problem)

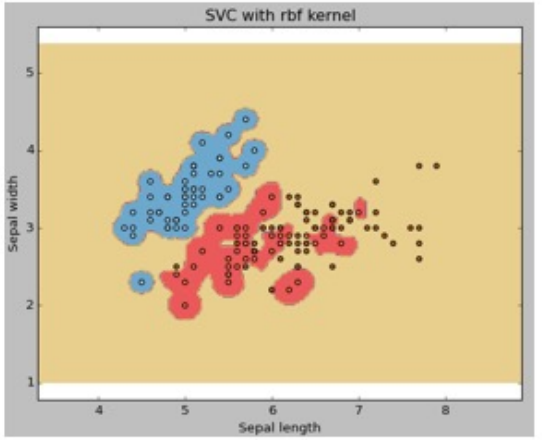
gamma =0



gamma =10



gamma =100



Pros and Cons of SVM

- **Pros**
  - Good when there is not too much training data (a million instances would be the upper bound) and works well with very small training sample sizes
  - Works great with Computer Vision problems
  - Good when high precision is critical
  - Effective in cases where number of dimensions is greater than the number of samples
  - Linear SVM is computationally efficient
  - Low generalization error
  - Can model complex relationships (powerful & flexible kernels)
  - Robust to noise (as they maximize margins)
  - Effective in high dimensional spaces (i.e. Text)

- **Cons**
  - Will not work right away (need parameter tuning and sensitive to tuning parameters & kernel choice)
  - Model parameters are difficult to interpret
  - Computationally expensive for nonlinear SVM when the training data size is large (requires significant memory and processing power)
  - Sensitive to noise in the data
  - Natively for binary classifications only (can be extended to multi-class)

What to use when

- A linear SVM can perform well for problems with many features but limited number of instances.
- A Gaussian kernel is particularly good for complex, nonlinear problems where the number of features is small and number of instances is large.
- SVM with a nonlinear kernel comes handy for complex and linearly not separable problems.
- Most widely used nonlinear kernel is the Gaussian kernel:

$$K_{RBF}(\mathbf{x}_i, \mathbf{x}_j) = \exp\left[-\gamma \left\| \mathbf{x}_i - \mathbf{x}_j \right\|^2\right]$$

  - If γ is small, we have high bias and low variance classifier; if γ is large, then we end up with a low bias and high variance classifier.

SVM w/ Python

```
#Linear Support Vector classification
#using scikit-learn library
from sklearn.svm import LinearSVC
from sklearn.metrics import accuracy_score
Clf = LinearSVC(C=1.0)
Clf.fit(X_train, y_train)
accuracy_score(y_test, Clf.predict(X_test))

#Support Vector with a kernel using scikit-learn
from sklearn.svm import svm
Clf = svm.SVC(probability=True, C=1.0,
              kernel='rbf', gamma=1.0)
Clf.fit(X_train, y_train)
accuracy_score(y_test, Clf.predict(X_test))
```

Caveat: **probabilty=True** enables probability estimates but it slows down the fit, and could be in conflict with the results of the **predict** method.

A Good reference on SVM:
www.hackerearth.com/blog/machine-learning/simple-tutorial-svm-parameter-tuning-python-r/