From Perceptron to SVM

Dr. Fayyaz Minhas

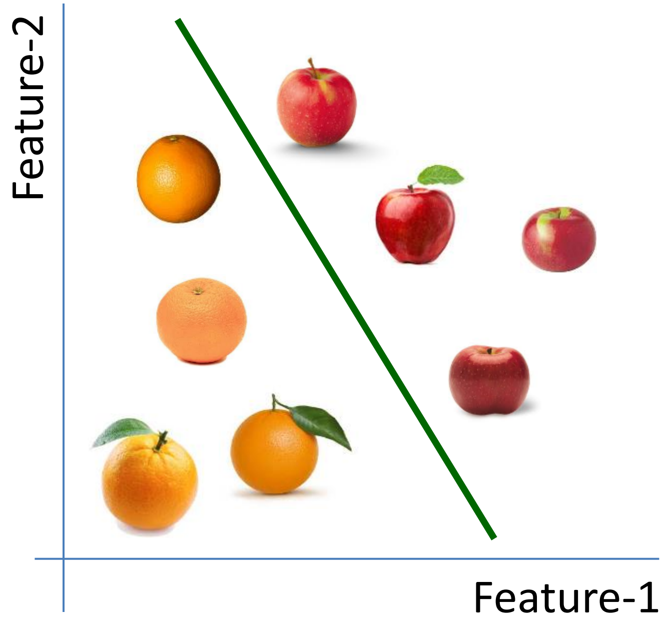
Department of Computer Science

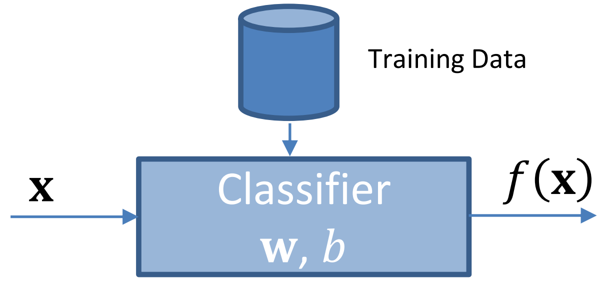
University of Warwick

https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs909/

Classification

- Given
  - A set of labelled training examples
- Find
  - A mathematical function that generalizes well to unseen cases
    - Discriminant function

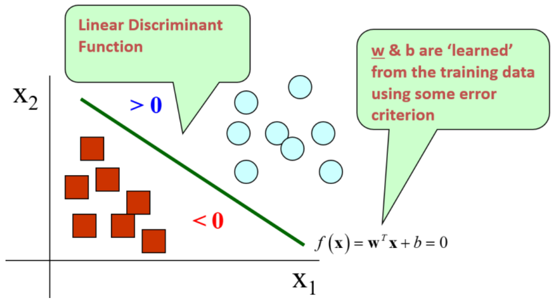




$$f(\mathbf{x}) = w_1\boldsymbol{x}_1 + w_2\boldsymbol{x}_2 + \cdots + w_d\boldsymbol{x}_d + b = \mathbf{w}^T\mathbf{x} + b$$

REO For Perceptron

- **Representation**
  - Features
  - Discriminant
    - Linear



Linear Discriminant Function
w & b are 'learned' from the training data using some error criterion
x_2
> 0
< 0
$f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b = 0$
x_1

$$f(\mathbf{x}) = w_1\boldsymbol{x}_1 + w_2\boldsymbol{x}_2 + \cdots + w_d\boldsymbol{x}_d + b = \mathbf{w}^T\mathbf{x}$$

- **Evaluation**
  - 0/1 (Step) Loss $$l(f(x), y) = \begin{cases} 0 & yf(x) > 0 \\ 1 & yf(x) \le 0 \end{cases}$$
  - Hinge Loss $$l(f(x), y) = \begin{cases} 0 & yf(\boldsymbol{x}) > 1 \\ 1 - yf(\boldsymbol{x}) & yf(\boldsymbol{x}) \le 1 \end{cases} = \max\{0, 1 - y(\boldsymbol{w}^T\boldsymbol{x})\}$$
  - Overall Loss $$L = \frac{1}{N}\sum_{i=1}^{N} l(f(\boldsymbol{x}_i;\mathbf{w})), y_i) \xrightarrow[iid]{} E[l(f(\boldsymbol{x};\mathbf{w})), y)]$$
- **Optimization**
  - Using Gradient Descent $$\nabla \boldsymbol{L} = \frac{1}{N}\sum_{i=1}^{N} \nabla_{\boldsymbol{w}} l(f(\boldsymbol{x}_i;\mathbf{w})), y_i)$$

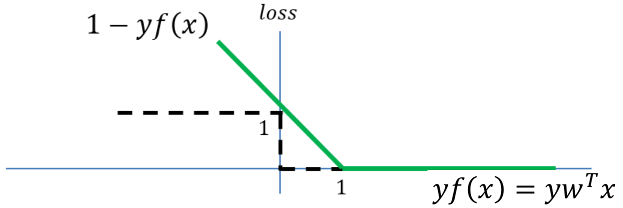
$$\nabla_{\boldsymbol{w}}\max\{0, 1 - y(\boldsymbol{w}^T\boldsymbol{x})\} = \begin{cases} 0 & 1 - yf(\boldsymbol{x};\mathbf{w}) < 0 \\ -y\boldsymbol{x} & else \end{cases} = \begin{cases} -y\boldsymbol{x} & l(f(\boldsymbol{x};\mathbf{w})), y) > 0 \\ 0 & else \end{cases} = \mathrm{I}(l(f(\boldsymbol{x};\mathbf{w})), y))(-y\boldsymbol{x})$$

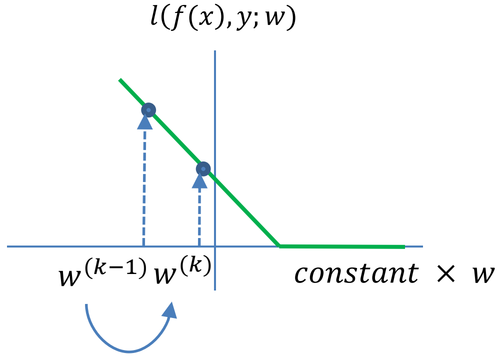
- Given:
  - Training Examples: $\{(\boldsymbol{x}_i, y_i) | i = 1 \ldots N\}, y_i \in \{-1, +1\}$
- Initialize $w^{(0)}$ at random
- Until Convergence (k=1…K)
  - For i = 1…N
    - Pick example $\boldsymbol{x}_i$ with label y_i
    - Compute $f(\boldsymbol{x}_i) = \boldsymbol{w}^{(\boldsymbol{k}-\boldsymbol{1})^{\boldsymbol{T}}}\boldsymbol{x}_i$
    - If $y_i f(\boldsymbol{x}_i) < 1$ then update your weight vector using gradient descent

$$\nabla_{\boldsymbol{w}}\max\{0, 1 - y(\boldsymbol{w}^T\boldsymbol{x})\} = \begin{cases} 0 & 1 - yf(\boldsymbol{x};\mathbf{w}) < 0 \\ -y\boldsymbol{x} & else \end{cases}$$

$$\boldsymbol{w}^{(\boldsymbol{k})} = \boldsymbol{w}^{(\boldsymbol{k}-\boldsymbol{1})} - \alpha\nabla\boldsymbol{l}\left(\boldsymbol{w}^{(\boldsymbol{k}-\boldsymbol{1})}\right) = \boldsymbol{w}^{(\boldsymbol{k}-\boldsymbol{1})} - \alpha(-y_i\boldsymbol{x}_i) = \boldsymbol{w}^{(\boldsymbol{k}-\boldsymbol{1})} + \alpha y_i\boldsymbol{x}_i$$

    - Check for convergence to stop





$$\boldsymbol{w}^{(\boldsymbol{k})} \leftarrow \boldsymbol{w}^{(\boldsymbol{k}-\boldsymbol{1})} - \alpha\nabla\boldsymbol{l}\left(\boldsymbol{w}^{(\boldsymbol{k}-\boldsymbol{1})}\right)$$

$$\boldsymbol{w}^{(\boldsymbol{k})} \leftarrow \boldsymbol{w}^{(\boldsymbol{k}-\boldsymbol{1})} - \boldsymbol{\alpha}\mathrm{I}(l(f(\boldsymbol{x};\mathbf{w})), y))(-y\boldsymbol{x}) = \underline{\boldsymbol{w}^{(\boldsymbol{k}-\boldsymbol{1})} + \alpha\mathrm{I}(l(f(\boldsymbol{x};\mathbf{w})), y))(y\boldsymbol{x})}$$

Coding in Python

```python
import numpy as np
import matplotlib.pyplot as plt
import itertools

class Perceptron:

    def __init__(self,alpha = 0.1, epochs = 200):
        self.alpha = alpha
        self.epochs = epochs
        self.W = np.array([0])
        self.bias = np.random.randn()
        self.Lambda = 0.5
    def fit(self,Xtr,Ytr):
        d = Xtr.shape[1]
        self.W = np.random.randn(d)
        for e in range(self.epochs):
            finished = True
            for i,x in enumerate(Xtr):
                if Ytr[i]!=self.predict(np.atleast_2d(x)):
                    finished = False
                    self.W += self.alpha*Ytr[i]*x
                    self.bias += self.alpha*Ytr[i]
            if finished: break

    def score(self,x):
        return np.dot(x,self.W) + self.bias

    def predict(self,x):
        return np.sign(self.score(x))
```

```python
if __name__=='__main__':
    from plotit import plotit
    Xtr = np.array([[-1,0],[0,1],[4,4],[2,3]])
    ytr = np.array([-1,-1,+1,+1])
    clf = Perceptron()
    clf.fit(Xtr,ytr)
    z = clf.score(Xtr)
    print("Prediction Scores:",z)
    y = clf.predict(Xtr)
    print("Prediction Labels:",y)
    plotit(Xtr,ytr,clf=clf.score,conts=[0],
           extent = [-5,+5,-5,+5])
```

```python
from sklearn.linear_model import Perceptron
clf = Perceptron()
clf.fit(X, y)
clf.predict(X)
```

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Perceptron.html

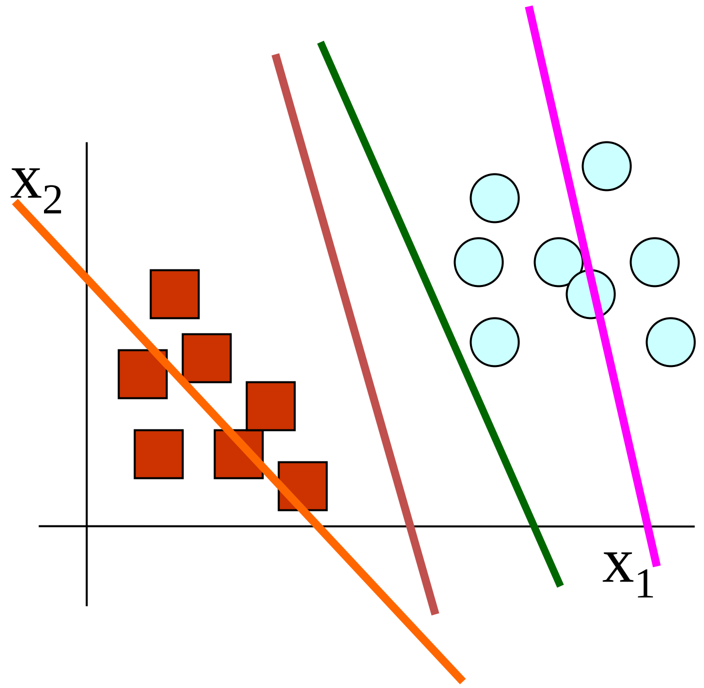
Empirical Risk Minimization

- So far, our machine learning models look like the following (empirical error) minimization:

$$f^* = argmin_f L(\boldsymbol{X_{train}}, \boldsymbol{Y_{train}}; f)$$

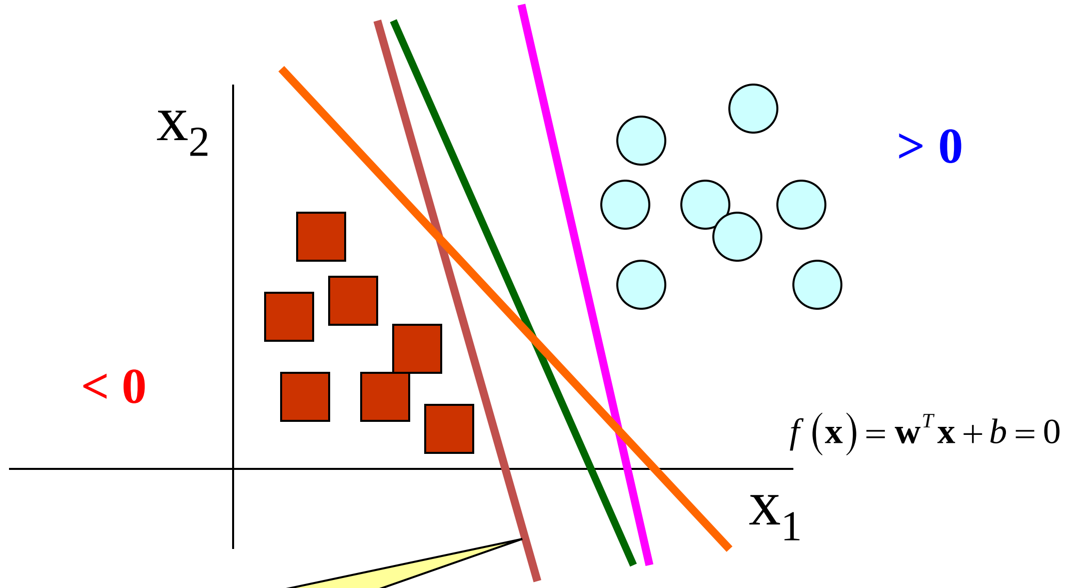
$$\boldsymbol{w}^* = argmin_{\boldsymbol{w}} L(\boldsymbol{X_{train}}, \boldsymbol{Y_{train}}; f)$$

- This is called ERM:
  - Learning only from training data



Issues with empirical risk parameters

- There are a large number of lines (or in general ‘hyperplanes’) separating the two classes

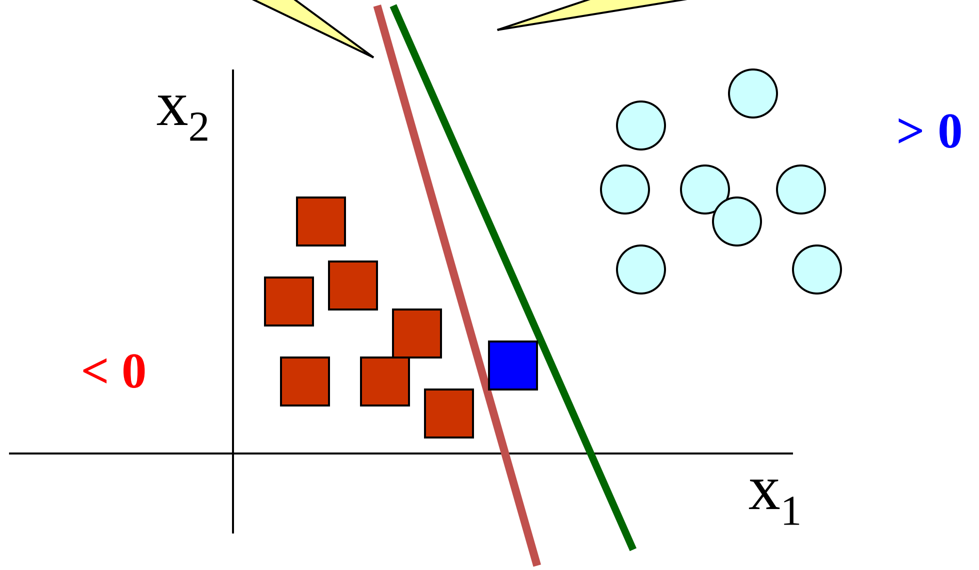


Which separator is the best?

Limitations of Empirical Risk Minimization

The boundary which lies closer to data points has low <u>margin</u> for error: A small change in the input can change the prediction label

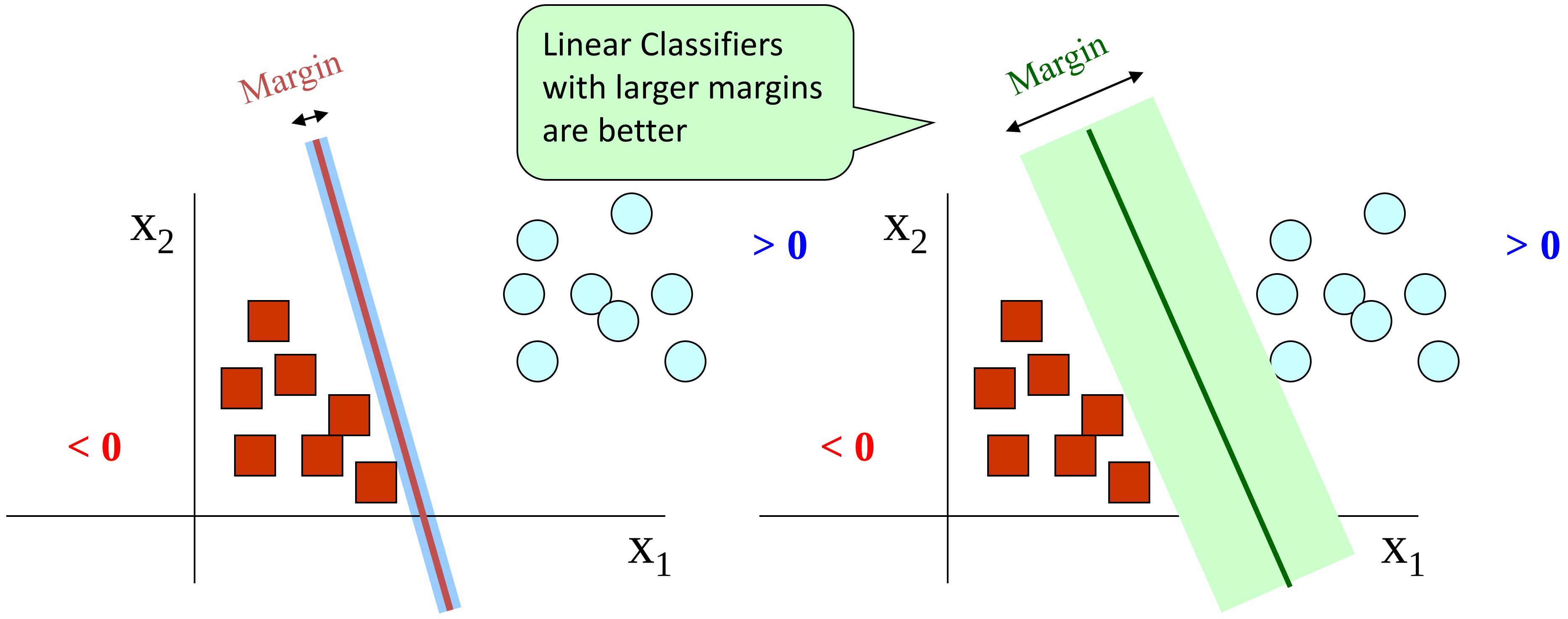
The boundary which lies at the maximum distance from data points of both classes gives better tolerance to noise and better "generalization"*



*Under the assumption that:
Test data is "identically distributed" as the training data

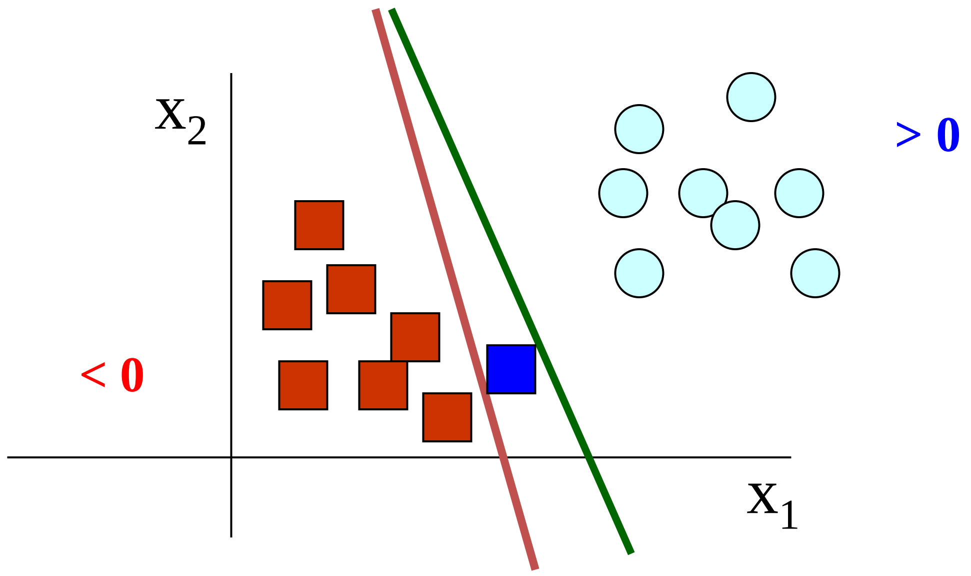
Margin of a linear classifier

- The width by which the boundary of a linear classifier can be increased before hitting a data point is called the margin of the linear classifier



Margin and Regularization

- Large Margin
- Classifiers with large margin have a property
  - Small changes in **x** should cause small changes in output: Regularization
- How can we achieve regularization?



$$f(\mathbf{x}) = w_1\boldsymbol{x}_1 + w_2\boldsymbol{x}_2 + \cdots + w_d\boldsymbol{x}_d + b = \mathbf{w}^T\mathbf{x} + b$$

Structural Risk Minimization

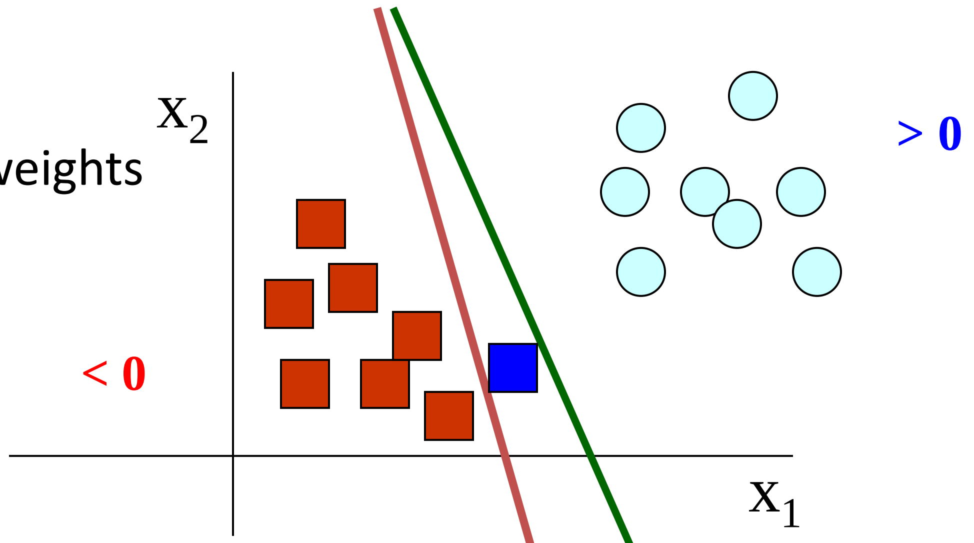
- In order to produce better generalization, we need to do both empirical error minimization but also reduce “Structural Risk”

- Formally, minimizing “structural risk” puts an upper bound on your generalization error
  - Structural risk control, in essence, controls the structure of your prediction model in addition to empirical error minimization

Vladimir Vapnik

Support Vector Machines (SVM)

- Support Vector Machines are linear classifiers that produce the optimal separating boundary (hyper-plane)
  - Find **w** and b in a way so as to:
    - Minimize misclassification error over training data (Empirical Risk Minimization)
    - Maximize the margin
      - Or equivalently, maximize regularization
      - Or equivalently, minimize the individual absolute weights



$f(\mathbf{x}) = w_1\boldsymbol{x}_1 + w_2\boldsymbol{x}_2 + \cdots + w_d\boldsymbol{x}_d + b = \mathbf{w}^T\mathbf{x} + b$

Understanding Regularization

- If **w** is too large (positive or negative)
  - Then a small change in **x** (e.g., due to noise) will cause a large change in the output $\boldsymbol{w}^T\boldsymbol{x} + b$
  - Can lead to errors
  - Controlling for this is called "Regularization"
- Achieved by minimizing:

$$R(f) = \boldsymbol{w}_1^2 + \boldsymbol{w}_2^2 + \cdots + \boldsymbol{w}_d^2 = \boldsymbol{w}^T\boldsymbol{w} = \|\boldsymbol{w}\|^2$$

More important than understanding margin based explanations as the concept of margin gets a bit difficult when moving from classification to other types of machine learning problems.

Small weights limit "the butterfly effect"

- *Let's quantify how sensitive the model is to a perturbation of its input*

- $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + b$
- $\boldsymbol{f}(\boldsymbol{x}+\boldsymbol{\delta x}) = \boldsymbol{w}^T(\boldsymbol{x}+\boldsymbol{\delta x}) + \boldsymbol{b} = \boldsymbol{w}^T\boldsymbol{x} + \boldsymbol{b} + \boldsymbol{w}^T\boldsymbol{\delta x} = \boldsymbol{f}(\boldsymbol{x}) + \boldsymbol{w}^T\boldsymbol{\delta x}$
- $\boldsymbol{f}(\boldsymbol{x}+\boldsymbol{\delta x}) - \boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{\delta x}$
- $\|\boldsymbol{f}(\boldsymbol{x}+\boldsymbol{\delta x}) - \boldsymbol{f}(\boldsymbol{x})\| = \|\boldsymbol{w}^T\boldsymbol{\delta x}\| \le \|\boldsymbol{w}\|\|\boldsymbol{\delta x}\|$ **(using Cauchy-Schwarz inequality)**

- **Therefore,** $\frac{\|\boldsymbol{f}(\boldsymbol{x}+\boldsymbol{\delta x}) - \boldsymbol{f}(\boldsymbol{x})\|}{\|\boldsymbol{\delta x}\|} \le \|\boldsymbol{w}\|$

Change in model output per unit additive change in input is upper bounded by $\|w\|$. Consequently, minimizing the norm of the weight vector (or its square) would lead to a regularization effect as it would limit the effect of any change in the input on the output.

Vapnik showed that **minimizing "structural risk"** (combination of empirical error over training examples and the norm of the weight vector) **leads to minimization of the upper bound on generalization error**.

$$R(w) \le R_{emp}(w) + \Omega\left(\frac{1}{N}, \frac{1}{\|\boldsymbol{w}\|}\right)$$

SRM to SVM

- Representation

  $f(\mathbf{x}) = w_1x^{(1)}+w_2x^{(2)}+\ldots+w_2x^{(d)}+b = \boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x} + b$

- Evaluation & Optimization

$$\min_{\boldsymbol{w}} R(f) + CL(\boldsymbol{X}, \boldsymbol{Y}; \boldsymbol{w})$$

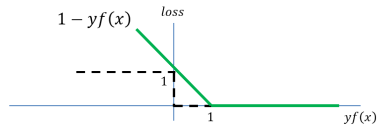
(Inverse of) Margin
AKA
Regularization term

C > 0 is a weighting factor that controls the relative contribution of both

Empirical Error/Loss over Training Data

$$R(f) = \frac{1}{2}\left(\boldsymbol{w}_{\boldsymbol{1}}^{\boldsymbol{2}} + \boldsymbol{w}_{\boldsymbol{2}}^{\boldsymbol{2}} + \cdots + \boldsymbol{w}_{\boldsymbol{d}}^{\boldsymbol{2}}\right) = \frac{1}{2}\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{w} = \frac{1}{2}\|\boldsymbol{w}\|^{\boldsymbol{2}}$$

$$L(\boldsymbol{X}, \boldsymbol{Y}; \boldsymbol{w}) = \frac{1}{N}\sum_{i=1}^{N} \max\{0, 1 - y_i f(\boldsymbol{x}_i; \mathbf{w})\}$$



$$\min_{\boldsymbol{w},b} \frac{1}{2}\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{w} + \frac{C}{N}\sum_{i=1}^{N} \max\{0, 1 - y_i f(\boldsymbol{x}_i)\}$$

SVM Optimization

$$\min_{\boldsymbol{w},b} \frac{1}{2}\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{w} + \frac{C}{N}\sum_{i=1}^{N} \max\{0,1-y_i f(\boldsymbol{x}_i)\}$$

Alternatively, we can use a weighting hyperparameter with the regularization term

$$\min_{w} P(\boldsymbol{w}) = \frac{\lambda}{2}\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{w} + \frac{1}{N}\sum_{i=1}^{N} \max\{0,1-y_i f(\boldsymbol{x}_i;\mathbf{w})\}$$

Gradient Descent Update Rule: $\boldsymbol{w}_k \leftarrow \boldsymbol{w}_{\boldsymbol{k-1}} - \alpha\nabla P(\boldsymbol{w}_{\boldsymbol{k-1}})$

Gradient calculation: $\nabla\boldsymbol{P} = \lambda\boldsymbol{w} - \frac{\partial}{\partial\boldsymbol{w}}\max\{0,1-y(\boldsymbol{w}^T\boldsymbol{x})\}$

$$\frac{\partial}{\partial\boldsymbol{w}}\max\{0,1-y(\boldsymbol{w}^T\boldsymbol{x})\} = \begin{cases} 0 & yf(\boldsymbol{x};\mathbf{w}) > 1 \\ -y\boldsymbol{x} & else \end{cases} = \mathbf{1}(yf(\boldsymbol{x}) < 1)(-y\boldsymbol{x})$$

$$\frac{\partial}{\partial\boldsymbol{w}}\left(\frac{\lambda}{2}\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{w}\right) = \lambda\boldsymbol{w}$$

$$\nabla\boldsymbol{P} = \lambda\boldsymbol{w} - 1(yf(\boldsymbol{x}) < 1)(-y\boldsymbol{x}) = \lambda\boldsymbol{w} + \mathbf{1}(yf(\boldsymbol{x}) < 1)(y\boldsymbol{x})$$

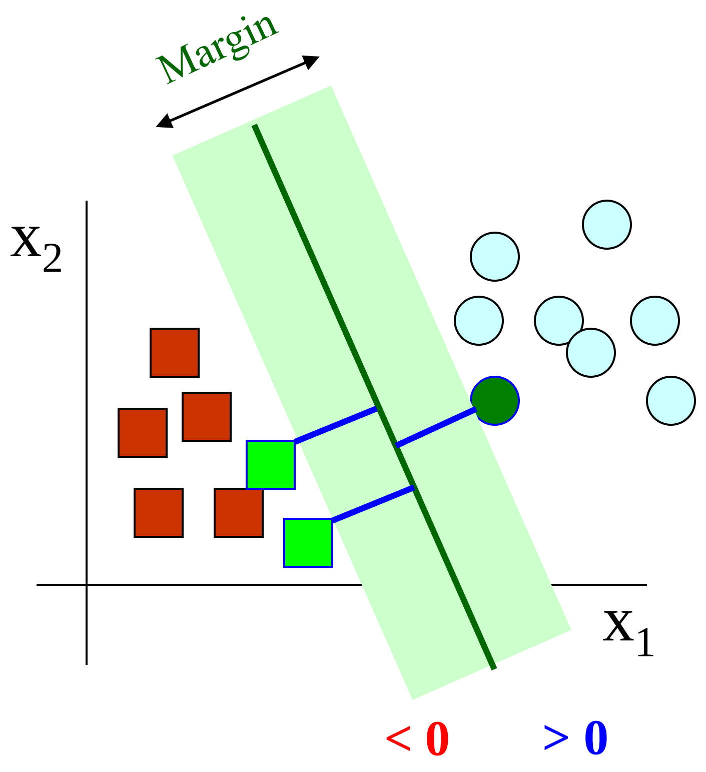
Final Weight Update Rule: $\boldsymbol{w}_k \leftarrow \boldsymbol{w}_{\boldsymbol{k-1}} - \alpha\lambda\boldsymbol{w}_{\boldsymbol{k-1}} - \alpha\mathbf{1}(yf(\boldsymbol{x}) < 1)(y\boldsymbol{x})$

Support Vector Machines

- Support Vector Machines, in their basic form, are linear classifiers that give maximum margin or regularization

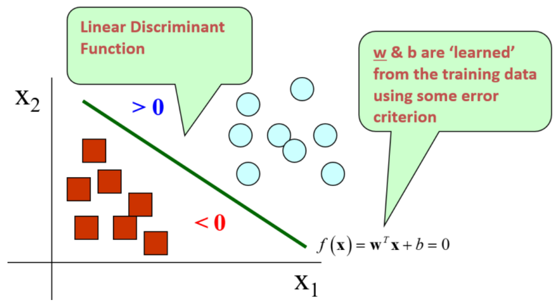
- Principles of Operation
  - Minimize the number of training errors
    - Achieved by minimizing hinge loss
  - Maximize margin
    - Allows noise tolerance
    - Allows Regularization
  - Perform Nonlinear Classification
    - Achieved through feature transformations/kernels

- The points that determine the margin are called Support Vectors



REO For Perceptron

- **Representation**
  - Features
  - Discriminant
    - Linear


Linear Discriminant Function
w & b are 'learned' from the training data using some error criterion
x_2
> 0
< 0
$f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b = 0$
x_1

$$f(\mathbf{x}) = w_1\boldsymbol{x}_1 + w_2\boldsymbol{x}_2 + \cdots + w_d\boldsymbol{x}_d + b = \mathbf{w}^T\mathbf{x}$$

(Ignoring Explicit Bias for Simplicity)

- **Evaluation**
  - 0/1 (Step) Loss $l(f(x), y) = \begin{cases} 0 & yf(x) > 0 \\ 1 & yf(x) \le 0 \end{cases}$
  - Hinge Loss $l(f(x), y) = \begin{cases} 0 & yf(\boldsymbol{x}) > 1 \\ 1 - yf(\boldsymbol{x}) & yf(\boldsymbol{x}) \le 1 \end{cases} = \max\{0, 1 - y(\boldsymbol{w}^T\boldsymbol{x})\}$
  - Overall Loss $L = \frac{1}{N}\sum_{i=1}^{N} l(f(\boldsymbol{x}_i; \mathbf{w})), y_i) \xrightarrow[iid]{} E[l(f(\boldsymbol{x}; \mathbf{w})), y)]$
- **Optimization**
  - Using Gradient Descent $\nabla \boldsymbol{L} = \frac{1}{N}\sum_{i=1}^{N} \nabla_{\boldsymbol{w}} l(f(\boldsymbol{x}_i; \mathbf{w})), y_i)$

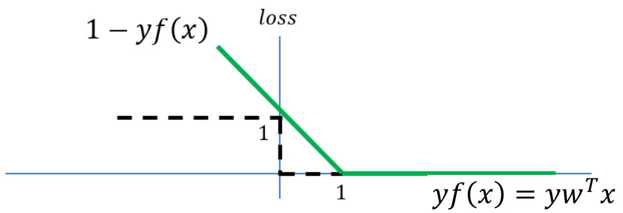
$$\nabla_{\boldsymbol{w}}\max\{0, 1 - y(\boldsymbol{w}^T\boldsymbol{x})\} = \begin{cases} 0 & 1 - yf(\boldsymbol{x}; \mathbf{w}) < 0 \\ -y\boldsymbol{x} & else \end{cases} = \begin{cases} -y\boldsymbol{x} & l(f(\boldsymbol{x}; \mathbf{w})), y) > 0 \\ 0 & else \end{cases} = \mathbf{I}(l(f(\boldsymbol{x}; \mathbf{w})), y) > 0)(-y\boldsymbol{x})$$

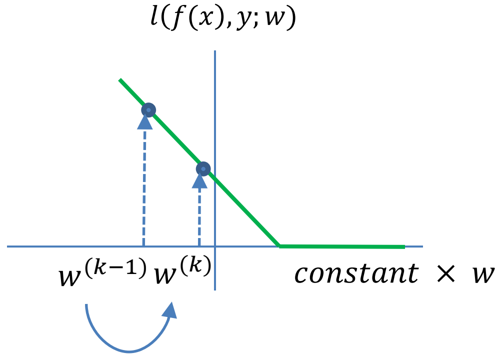
- Given:
  - Training Examples: $\{(\boldsymbol{x}_i, y_i) | i = 1 \dots N\}, y_i \in \{-1, +1\}$
- Initialize $w^{(0)}$ at random
- Until Convergence (k=1…K)
  - For i = 1…N
    - Pick example $\boldsymbol{x}_i$ with label y_i
    - Compute $f(\boldsymbol{x}_i) = \boldsymbol{w}^{(\boldsymbol{k}-\boldsymbol{1})^{\boldsymbol{T}}}\boldsymbol{x}_{\boldsymbol{i}}$
    - If $y_i f(\boldsymbol{x}_i) < 1$ then update your weight vector using gradient descent

$$\nabla_{\boldsymbol{w}}\max\{0, 1 - y(\boldsymbol{w}^T\boldsymbol{x})\} = \begin{cases} 0 & 1 - yf(\boldsymbol{x}; \mathbf{w}) < 0 \\ -y\boldsymbol{x} & else \end{cases}$$

$$\boldsymbol{w}^{(\boldsymbol{k})} = \boldsymbol{w}^{(\boldsymbol{k}-\boldsymbol{1})} - \alpha\nabla\boldsymbol{l}(\boldsymbol{w}^{(\boldsymbol{k}-\boldsymbol{1})}) = \boldsymbol{w}^{(\boldsymbol{k}-\boldsymbol{1})} - \alpha(-y_i\boldsymbol{x}_i) = \boldsymbol{w}^{(\boldsymbol{k}-\boldsymbol{1})} + \alpha y_i\boldsymbol{x}_i$$

    - Check for convergence to stop





$$\boldsymbol{w}^{(\boldsymbol{k})} \leftarrow \boldsymbol{w}^{(\boldsymbol{k}-\boldsymbol{1})} - \alpha\nabla\boldsymbol{l}(\boldsymbol{w}^{(\boldsymbol{k}-\boldsymbol{1})})$$

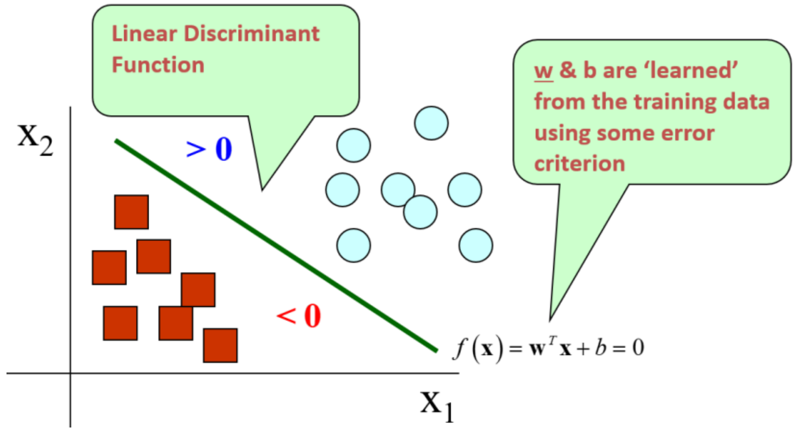
$$\boldsymbol{w}^{(\boldsymbol{k})} \leftarrow \boldsymbol{w}^{(\boldsymbol{k}-\boldsymbol{1})} - \boldsymbol{\alpha}\mathbf{I}(l(f(\boldsymbol{x}; \mathbf{w})), y))(-y\boldsymbol{x}) = \boldsymbol{w}^{(\boldsymbol{k}-\boldsymbol{1})} \underline{+ \alpha\mathbf{I}(l(f(\boldsymbol{x}; \mathbf{w})), y) > 0)(y\boldsymbol{x})}$$

REO For SVM

- **Representation**
  - Features
  - Discriminant
    - Linear

(Ignoring Explicit Bias for Simplicity)

$$f(\mathbf{x};\boldsymbol{w}) = w_1\boldsymbol{x}_1 + w_2\boldsymbol{x}_2 + \cdots + w_d\boldsymbol{x}_d = \mathbf{w}^T\mathbf{x}$$



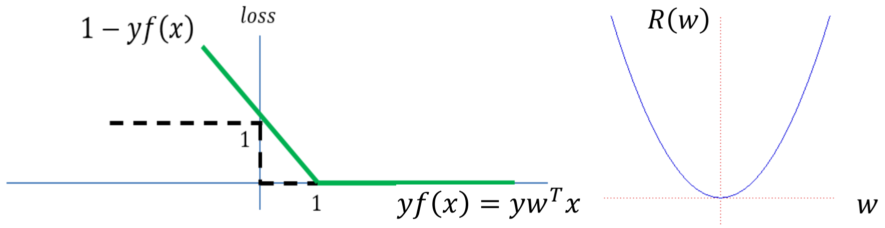
Linear Discriminant Function
w & b are 'learned' from the training data using some error criterion
x_2
> 0
< 0
$f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b = 0$
x_1

- **Evaluation**
  - Hinge Loss $l(f(x),y) = \begin{cases} 0 & yf(\boldsymbol{x}) > 1 \\ 1 - yf(\boldsymbol{x}) & yf(\boldsymbol{x}) \le 1 \end{cases} = \max\{0, 1 - y(\boldsymbol{w}^T\boldsymbol{x})\}$
    - Minimize training error (Empirical Risk Minimization)
  - Regularization $R(f) = R(w) = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w}$
    - Minimize Impact of small changes in examples
  - Structural Risk Minimization: $\min_w P(\boldsymbol{w}) = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + \frac{C}{N}\sum_{i=1}^{N}\max\{0, 1 - y_i f(\boldsymbol{x}_i;\mathbf{w})\}$

$$\min_f R(f) + L(f; X, Y)$$

Regularization Empirical Error

$$\min_w P(\boldsymbol{w}) = \frac{\lambda}{2}\boldsymbol{w}^T\boldsymbol{w} + \frac{1}{N}\sum_{i=1}^{N}\max\{0, 1 - y_i f(x_i;\boldsymbol{w})\}$$



Hyperparameter λ or C: Control the relative weighting of the Regularization and Empirical Error Minimization terms

- **Optimization**
  - Using Gradient Descent $\boldsymbol{w}^{(k)} \leftarrow \boldsymbol{w}^{(k-1)} - \alpha\nabla\boldsymbol{P}\left(\boldsymbol{w}^{(k-1)}\right)$

$$\nabla\boldsymbol{P} = \nabla_{\boldsymbol{w}}\left(\frac{\lambda}{2}w^Tw\right) + \frac{1}{N}\sum_{i=1}^{N}\nabla_{\boldsymbol{w}} l(f(\boldsymbol{x}_i;\mathbf{w})), y_i)$$

$$\nabla_{\boldsymbol{w}}\left(\frac{\lambda}{2}w^Tw\right) = \lambda w \qquad \nabla_{\boldsymbol{w}}\max\{0, 1 - y(\boldsymbol{w}^T\boldsymbol{x})\} = \begin{cases} 0 & 1 - yf(\boldsymbol{x};\mathbf{w}) < 0 \\ -y\boldsymbol{x} & else \end{cases}$$

$$\boxed{\boldsymbol{w}_k \leftarrow \boldsymbol{w}_{k-1} - \alpha\lambda\boldsymbol{w}_{k-1} - \alpha\boldsymbol{I}(yf(\boldsymbol{x}) < 1)(y\boldsymbol{x})}$$

For a single training example

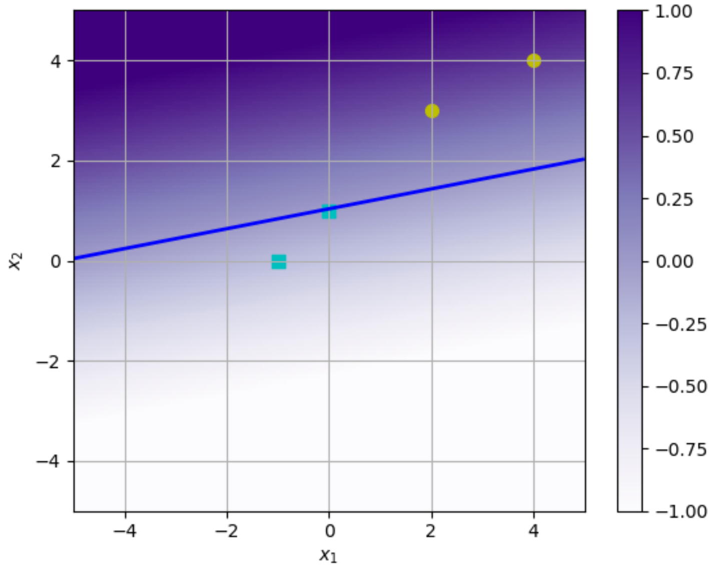
Coding in Python

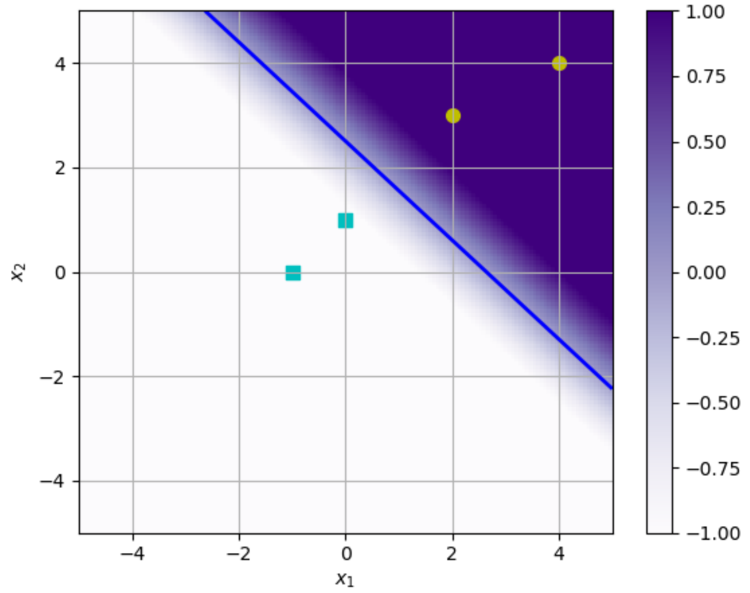
```python
import numpy as np
import matplotlib.pyplot as plt
import itertools

class RegularizedPerceptron:
    def __init__(self,Lambda = 0.0, margin = 0.0, alpha = 0.1, epochs = 1000):
        self.alpha = alpha
        self.epochs = epochs
        self.W = np.array([0])
        self.bias = np.random.randn()
        self.Lambda = Lambda #not used in perceptron
        self.Margin = margin #0.0 in Perceptron
    def fit(self,Xtr,Ytr):
        d = Xtr.shape[1]
        self.W = np.random.randn(d)
        for e in range(self.epochs):
            finished = True
            for i,x in enumerate(Xtr):
                if self.score(np.atleast_2d(x))*Ytr[i]<self.Margin:
                    self.W += self.alpha*Ytr[i]*x
                    self.bias += self.alpha*Ytr[i]

            self.W = self.W-self.alpha*self.Lambda*self.W #Regularization update

    def score(self,x):
        return np.dot(x,self.W) + self.bias

    def predict(self,x):
        return np.sign(self.score(x))
```

```python
if __name__=='__main__':
    from plotit import plotit
    Xtr = np.array([[-1,0],[0,1],[4,4],[2,3]])
    ytr = np.array([-1,-1,+1,+1])
    clf = RegularizedPerceptron(Lambda = 0.1, margin = 1.0)
    clf.fit(Xtr,ytr)
    z = clf.score(Xtr)
    print("Prediction Scores:",z)
    y = clf.predict(Xtr)
    print("Prediction Labels:",y)
    plotit(Xtr,ytr,clf=clf.score,conts=[0],extent = [-5,+5,-5,+5])
```

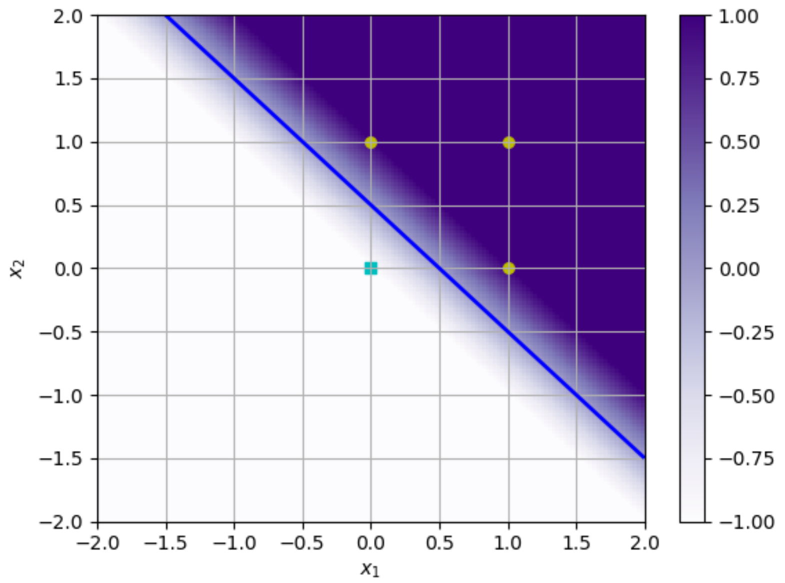
https://github.com/foxtrotmike/CS909/blob/master/regper.ipynb

Difference between Perceptron and SVM



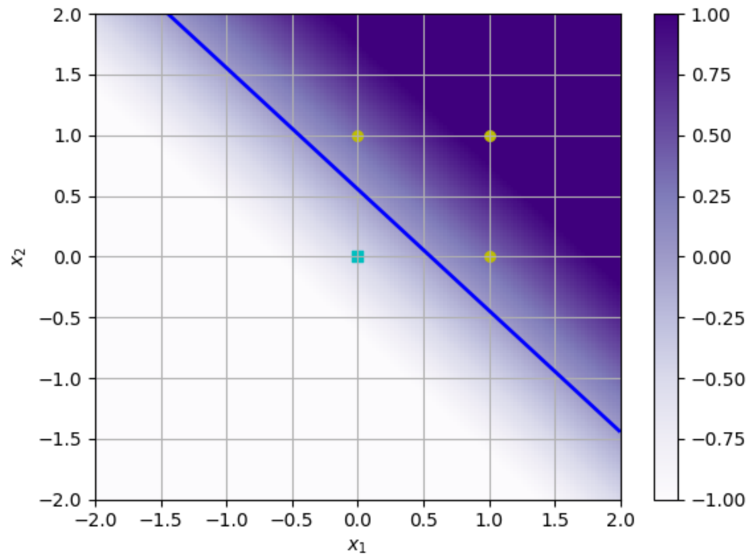


SVM in Sklearn



C=1000

w=[2,2],b=-1



C=1

w = [0.84 0.84],b=-0.465

```
import numpy as np
from sklearn.svm import LinearSVC as Classifier

X = np.array([[0,0],[0,1],[1,0],[1,1]])
y =np.array([-1,1,1, 1])
clf = Classifier(class_weight='balanced',C=100)
clf.fit(X, y)
f = clf.predict(X)
print('Coefficients before adding additional feature:', clf.coef_,clf.intercept_)
print('Predictions before adding additional feature:',f)
plotit(X,y,clf = clf.decision_function,conts=[0],extent=[-2,+2,-2,+2])
```

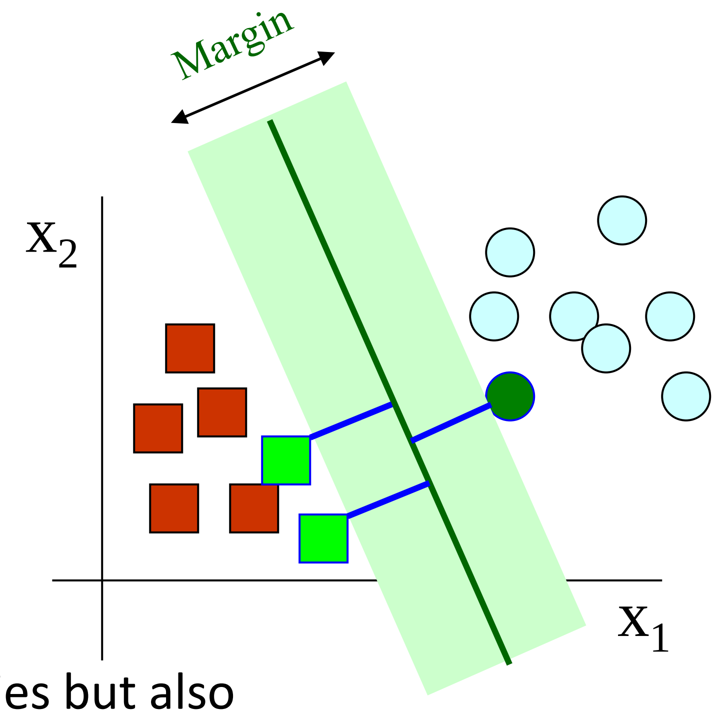
Wanna Play?

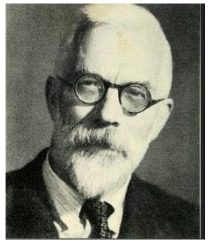
- Use the Java Applet at:
- https://www.csie.ntu.edu.tw/~cjlin/libsvm/
- Set “-t 0 -c 100”

SVMs up till now

$$\min_{\boldsymbol{w},b} \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + \frac{C}{N}\sum_{i=1}^{N}\max\{0,1-y_i f(\boldsymbol{x}_i)\}$$

- Vapnik and Chervonenkis:
  - Theoretical foundations for SVMs
  - Structural Risk Minimization
- Corinna Cortes
  - Soft SVM (1995)
- Bernard Scholkopf (1997)
  - Representer Theorem
  - Complete Kernel trick!
  - Kernels not only allow nonlinear boundaries but also allow representation of non-vectoral data




R. A. Fisher
1890-1962

Rosenblatt
1928-1971

V. Vapnik
1936 -

Chervonenkis
1938 - 2014

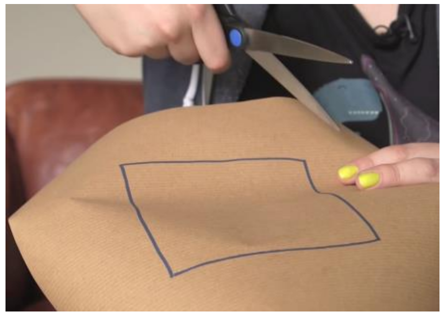
C. Cortes
1961 -

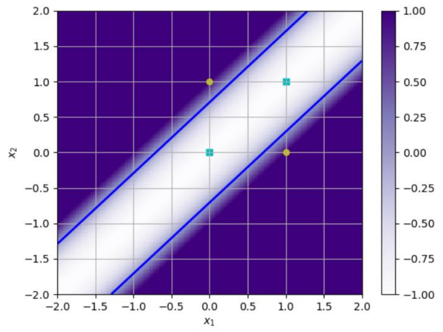
Scholkopf
1968 -

http://www.svms.org/history.html

How to achieve non-linear classification boundaries?

- So far we have only discussed linear classification
- How can we solve non-linear classification?
  - By folding the space on which examples lie and then making a single straight cut
    - Notice how folding changes the distance between points
  - How to achieve such folding?
    - One way is to transform the data





https://www.youtube.com/watch?v=ZREp1mAPKTM

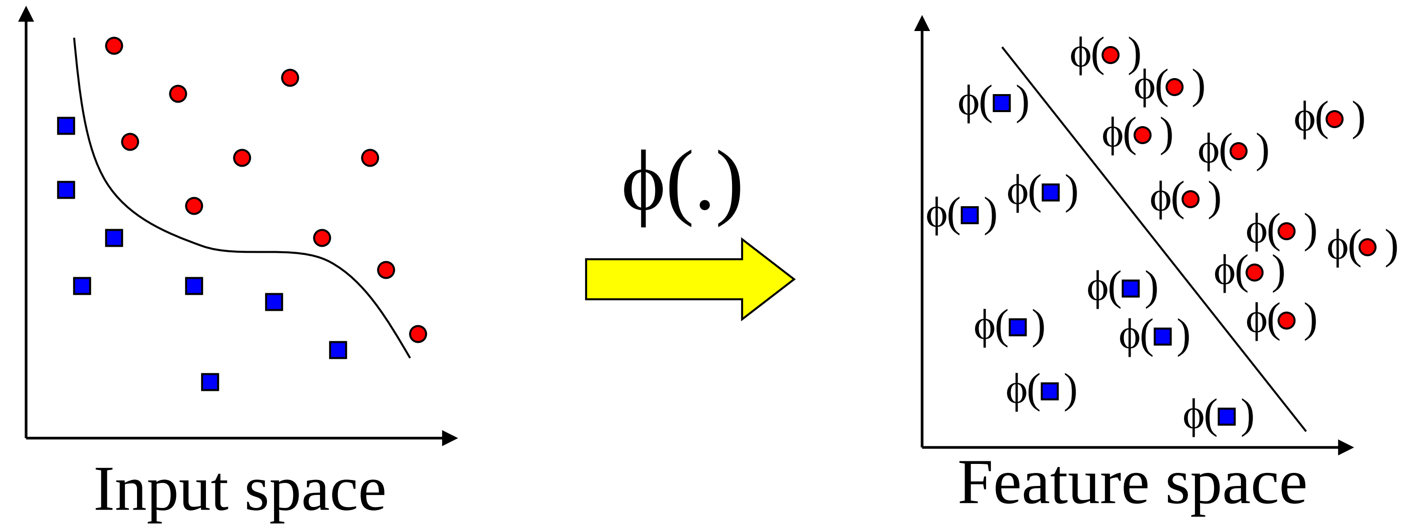
The Fold-and-Cut Theorem implies that any pattern can be achieved with a single straight cut if the paper (or space) is folded appropriately.

Thus, it is theoretically possible to partition any space into regions containing positive and negative training examples no matter how complex such a boundary is by simply folding the feature space appropriately and using a linear classifier (single straight cut).

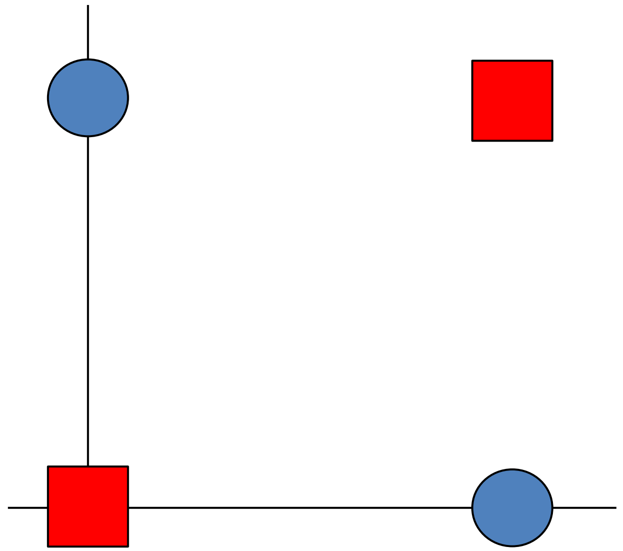
https://en.wikipedia.org/wiki/Fold-and-cut_theorem

Nonlinear Separation through Transformation

- Given a classification problem with a nonlinear boundary, we can, at times, find a mapping or transformation of the feature space which makes the classification problem linear separable in the transformed space



Examples: Transformation



$f(\mathbf{x};\boldsymbol{\theta}) = w_1x^{(1)}+w_2x^{(2)}+b= 0$

(0,0): $b < 0$
(0,1): $w_2+b > 0$
(1,0): $w_1+b > 0$
(1,1): $w_1+ w_2 + b <0$

$f(\mathbf{x};\boldsymbol{\theta}) = w_1x^{(1)}+w_2x^{(2)}+w_3x^{(3)} +b= 0$

(0,0,0): $b < 0$
(0,1,0): $w_2+b > 0$
(1,0,0): $w_1+b > 0$
$(1,1,\sqrt{2})$: $w_1+ w_2 + \sqrt{2}\, w_3 + b <0$

$w_1 = 2,\ w_2 = 2,\ w_3 = -3,\ b = -1$

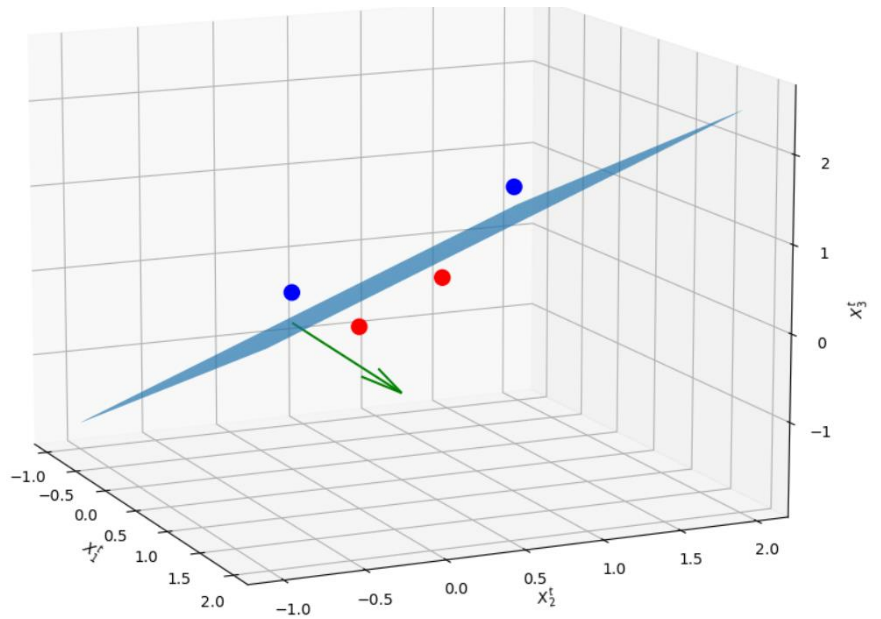
$x^{(1)}$	$x^{(2)}$	y
0	0	-1
0	1	+1
1	0	+1
1	1	-1

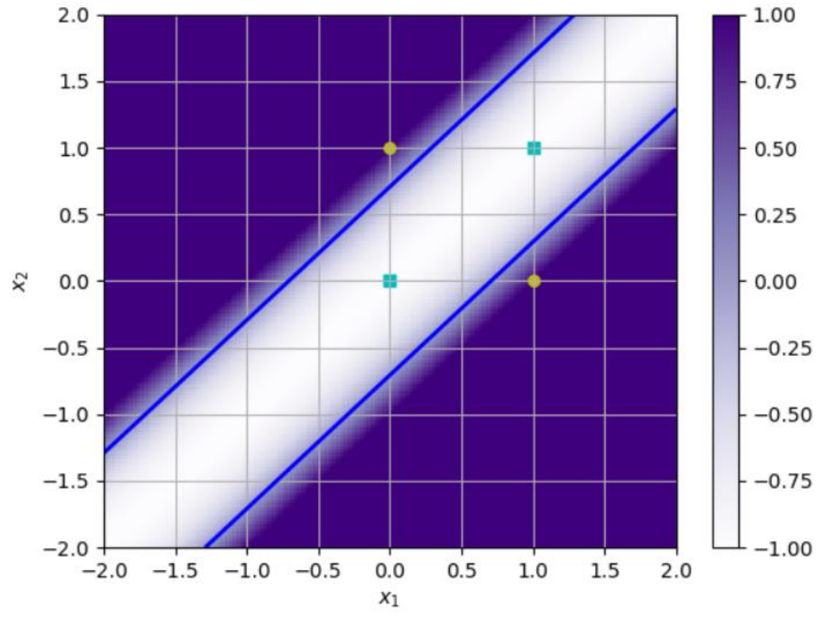
$$\boldsymbol{\phi}\left(\begin{bmatrix} \boldsymbol{x}^{(1)} \\ \boldsymbol{x}^{(2)} \end{bmatrix}\right) = \begin{bmatrix} \boldsymbol{x}^{(1)^2} \\ \boldsymbol{x}^{(2)^2} \\ \sqrt{2}\boldsymbol{x}^{(1)}\boldsymbol{x}^{(2)} \end{bmatrix} \longrightarrow$$

$x'^{(1)}$	$x'^{(2)}$	$x'^{(3)}$	y
0	0	0	-1
0	1	0	+1
1	0	0	+1
1	1	$\sqrt{2}$	-1

```
import numpy as np
from sklearn.svm import LinearSVC as Classifier
from plotit import *
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm

X = np.array([[0,0],[0,1],[1,0],[1,1]])
y = np.array([-1,1,1,-1])
clf2d = Classifier(C=1000).fit(X, y)
f = clf2d.predict(X)
print('Coefficients before Transformation:', clf2d.coef_,clf2d.intercept_)
print('Predictions before Transformation:',f)

transform = lambda x: np.hstack((x**2,np.atleast_2d(np.sqrt(2)*x[:,0]*x[:,1]).T))
Xt = transform(X)
print(Xt)
clf = Classifier(C=1000).fit(Xt, y)
f = clf.predict(Xt)
print('Coefficients after Transformation:',clf.coef_,clf.intercept_)
print('Predictions after Transformation:',f)

# showing the plane in 3d
xx,yy = np.arange(-1, 2, 0.01), np.arange(-1, 2, 0.01)
yy = xx, yy = np.meshgrid(xx, yy)
zz = -(clf.coef_[0,0]*xx+clf.coef_[0,1]*yy+clf.intercept_[0])/(clf.coef_[0,2])
fig = plt.figure(); ax = fig.add_subplot(111, projection='3d')
Xp,Xn = Xt[y==1,:],Xt[y!=1,:]
ax.scatter(Xp[:,0], Xp[:,1], Xp[:,2],color = 'red',alpha=1,s=100)
ax.scatter(Xn[:,0], Xn[:,1], Xn[:,2],color = 'blue',alpha=1,s=100)
ax.plot_surface(xx, yy, zz,linewidth=0, antialiased=True)
ax.set_xlabel('$X^t_1$'); ax.set_ylabel('$X^t_2$'); ax.set_zlabel('$X^t_3$')
# Normal vector (coef_ of the SVM)
normal_vector = clf.coef_[0]
start_point = [0, 0, -clf.intercept_[0] / clf.coef_[0,2]]
ax.quiver(start_point[0], start_point[1], start_point[2],
          normal_vector[0], normal_vector[1], normal_vector[2], length=1, color='green', normalize=True)

# showing the boundary in 2d
plt.figure(); plotit(X,y,clf = clf.decision_function,transform = transform,conts=[0],extent=[-2,+2,-2,+2])
```

https://github.com/foxtrotmike/CS909/blob/master/transformations.ipynb



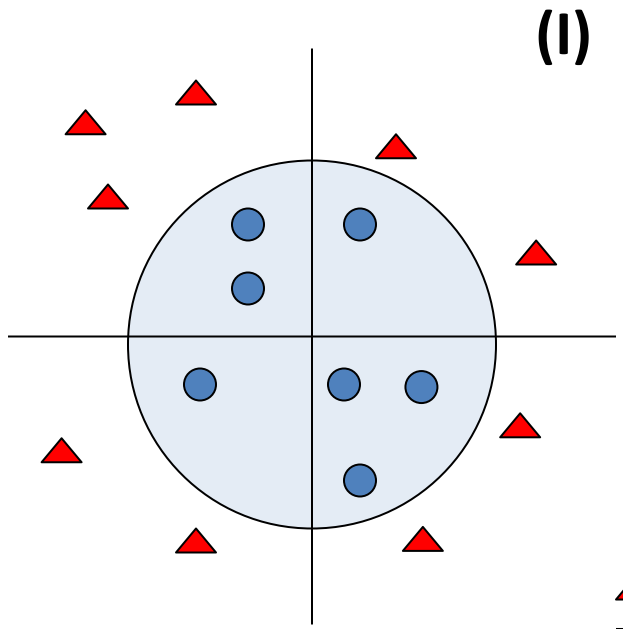


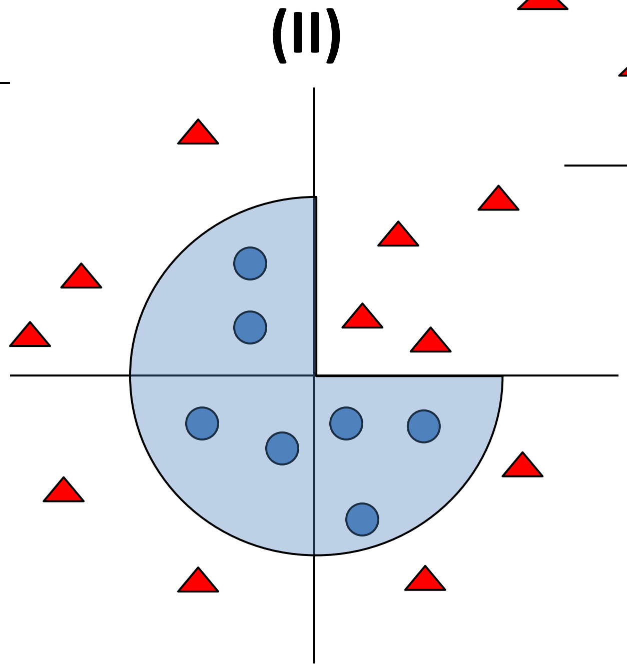
Examples: Transformation

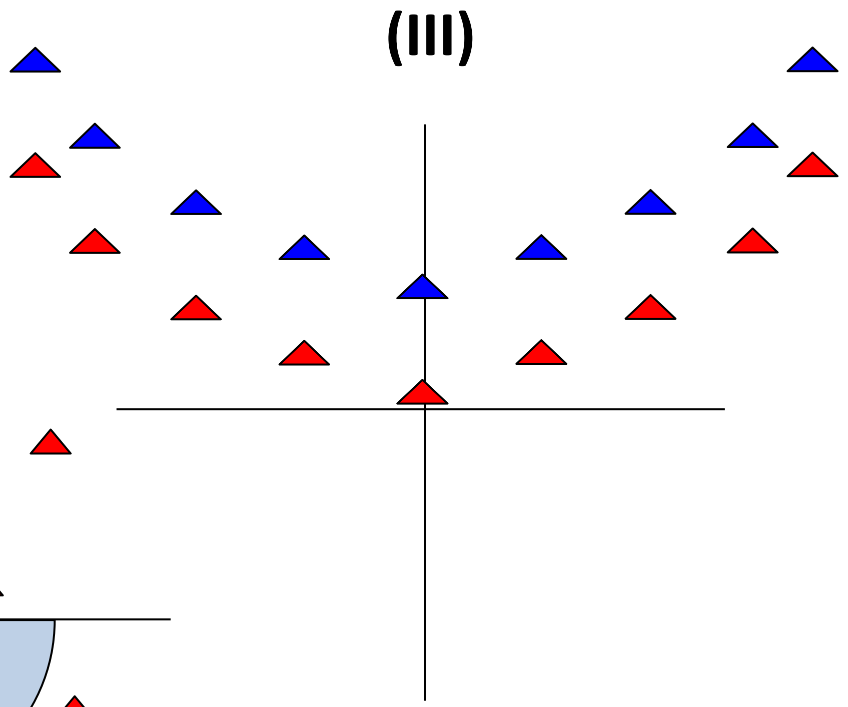
- Does this mapping do it?
  - $\boldsymbol{\phi}\left(\begin{bmatrix}\boldsymbol{x}^{(\mathbf{1})}\\ \boldsymbol{x}^{(\mathbf{2})}\end{bmatrix}\right) = \begin{bmatrix}\boldsymbol{x}^{(\mathbf{1})^{\mathbf{2}}}\\ \boldsymbol{x}^{(\mathbf{2})^{\mathbf{2}}}\\ \sqrt{2}\boldsymbol{x}^{(\mathbf{1})}\boldsymbol{x}^{(\mathbf{2})}\end{bmatrix}$
- What about this one?
  - $\boldsymbol{\phi}\left(\begin{bmatrix}\boldsymbol{x}^{(\mathbf{1})}\\ \boldsymbol{x}^{(\mathbf{2})}\end{bmatrix}\right) = \left(\boldsymbol{x}^{(\mathbf{1})} + \boldsymbol{x}^{(\mathbf{2})} - 1\right)^{\mathbf{2}}$

Transformation Examples

- Can you find a transform that makes the following classification problems linear separable? Can you draw the data points in the new transformed feature space?







Feature Transformation ⟷ Distance Change

- Feature transformations change the concept of distance or dot product between two points
  - Consider:

$$d(\boldsymbol{a},\boldsymbol{b}) = \|\boldsymbol{a}-\boldsymbol{b}\|^2 = (\boldsymbol{a}-\boldsymbol{b})^T(\boldsymbol{a}-\boldsymbol{b})$$
$$= \boldsymbol{a}^T\boldsymbol{a} + \boldsymbol{b}^T\boldsymbol{b} - 2\boldsymbol{a}^T\boldsymbol{b}$$

  - After transformation: $\boldsymbol{x} \rightarrow \boldsymbol{\phi}(\boldsymbol{x})$, the value of the distance between points changes.

```
from scipy.spatial.distance import pdist, squareform
D = squareform(pdist(Xt,metric='sqeuclidean'))
```

$x^{(1)}$	$x^{(2)}$	y
0	0	-1
0	1	+1
1	0	+1
1	1	-1

$d(\boldsymbol{a},\boldsymbol{b})$

i	1	2	3	4
1	0	1	1	2
2	1	0	2	1
3	1	2	0	1
4	2	1	1	0

$$\downarrow$$

$$\boldsymbol{\phi}\left(\begin{bmatrix} \boldsymbol{x}^{(1)} \\ \boldsymbol{x}^{(2)} \end{bmatrix}\right) = \begin{bmatrix} \boldsymbol{x}^{(1)^2} \\ \boldsymbol{x}^{(2)^2} \\ \sqrt{2}\boldsymbol{x}^{(1)}\boldsymbol{x}^{(2)} \end{bmatrix}$$

$x'^{(1)}$	$x'^{(2)}$	$x'^{(3)}$	y
0	0	0	-1
0	1	0	+1
1	0	0	+1
1	1	$\sqrt{2}$	-1

i	1	2	3	4
1	0	1	1	4
2	1	0	2	3
3	1	2	0	1
4	4	3	3	0

$d_{\phi}(\boldsymbol{a},\boldsymbol{b})$

Dot Product Change $\leftrightarrow$ Distance Change

- Notice how the distance formula contains nothing but dot products?

$$d(\boldsymbol{a},\boldsymbol{b}) = \|\boldsymbol{a}-\boldsymbol{b}\|^2 = (\boldsymbol{a}-\boldsymbol{b})^T(\boldsymbol{a}-\boldsymbol{b})$$
$$= \boldsymbol{a}^T\boldsymbol{a} + \boldsymbol{b}^T\boldsymbol{b} - 2\boldsymbol{a}^T\boldsymbol{b}$$

After transformation, the distance is defined in terms of dot products in the transformed space

$$d_\phi(\boldsymbol{a},\boldsymbol{b}) = \|\boldsymbol{\phi}(\boldsymbol{a}) - \boldsymbol{\phi}(\boldsymbol{b})\|^2$$
$$= (\boldsymbol{\phi}(\boldsymbol{a}) - \boldsymbol{\phi}(\boldsymbol{b}))^T(\boldsymbol{\phi}(\boldsymbol{a}) - \boldsymbol{\phi}(\boldsymbol{b}))$$
$$= \boldsymbol{\phi}(\boldsymbol{a})^T\boldsymbol{\phi}(\boldsymbol{a}) + \boldsymbol{\phi}(\boldsymbol{b})^T\boldsymbol{\phi}(\boldsymbol{b}) - 2\boldsymbol{\phi}(\boldsymbol{a})^T\boldsymbol{\phi}(\boldsymbol{b})$$

We call dot products in the transformed space “Kernels”

$$d_\phi(\boldsymbol{a},\boldsymbol{b}) = k_\phi(\boldsymbol{a},\boldsymbol{a}) + k_\phi(\boldsymbol{b},\boldsymbol{b}) - 2k_\phi(\boldsymbol{a},\boldsymbol{b})$$

With

$$k_\phi(\boldsymbol{a},\boldsymbol{b}) = \boldsymbol{\phi}(\boldsymbol{a})^T\boldsymbol{\phi}(\boldsymbol{b})$$

$x^{(1)}$	$x^{(2)}$	y
0	0	-1
0	1	+1
1	0	+1
1	1	-1

$$\Big\downarrow \boldsymbol{\phi}$$

$x'^{(1)}$	$x'^{(2)}$	$x'^{(3)}$	y
0	0	0	-1
0	1	0	+1
1	0	0	+1
1	1	$\sqrt{2}$	-1

Transformed Data

$k(a,b) = \boldsymbol{a}^T\boldsymbol{b}$

i	1	2	3	4
1	0	0	0	0
2	0	1	0	1
3	0	0	1	1
4	0	1	1	2

$d(\boldsymbol{a},\boldsymbol{b})$

i	1	2	3	4
1	0	1	1	2
2	1	0	2	1
3	1	2	0	1
4	2	1	1	0

Kernel Matrix

Distance Matrix

$k_\phi(a,b) = \boldsymbol{\phi}(\boldsymbol{a})^T\boldsymbol{\phi}(\boldsymbol{b})$

i	1	2	3	4
1	0	0	0	0
2	0	1	0	1
3	0	0	1	1
4	0	1	1	4

$d_\phi(\boldsymbol{a},\boldsymbol{b})$

i	1	2	3	4
1	0	1	1	4
2	1	0	2	3
3	1	2	0	1
4	4	3	3	0

Feature Transformation ↔ Distance Change ↔ Kernels

- So far, we have established that:
  - Whenever the features are transformed
    - The distance changes
    - The dot product values changes
- But **it also means that**
  - If I change the distance between points, I will be applying a transformation
  - If I change the dot product or kernel, we will change the distance or in essence achieve an implicit transformation

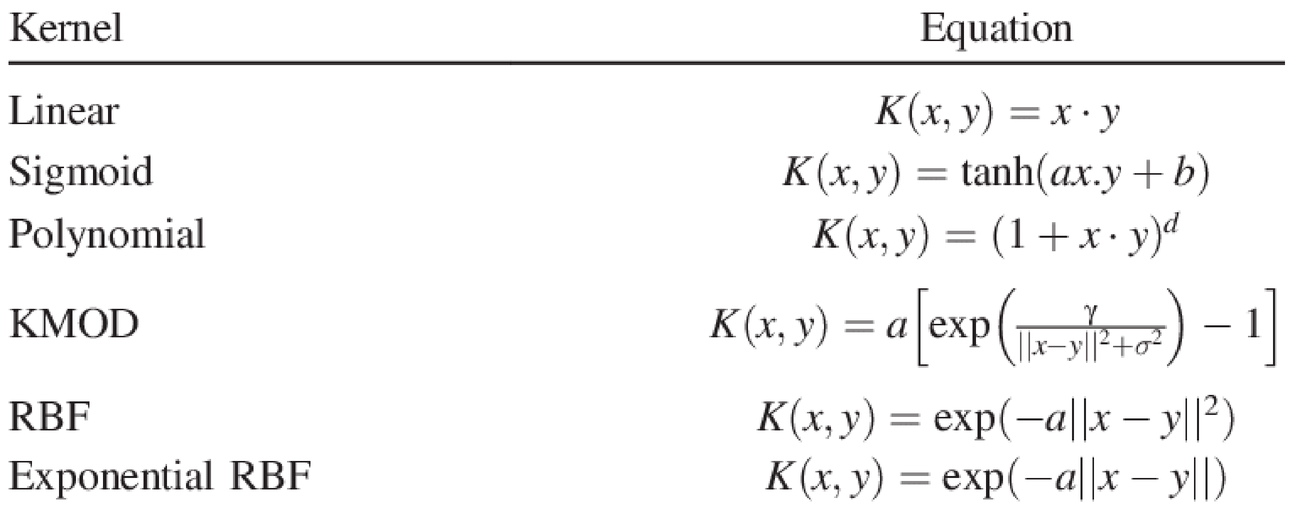
Feature Transformation $\leftrightarrow$ Kernel

- Let's say, we have 2D data, then
  - $k(\boldsymbol{a},\boldsymbol{b}) = \boldsymbol{a}^{\boldsymbol{T}}\boldsymbol{b} = a^{(1)}b^{(1)} + a^{(2)}b^{(2)}$
- Let's change the definition of the dot product or kernel as follows
  - $k_\phi(\boldsymbol{a},\boldsymbol{b}) = \left(\boldsymbol{a}^{\boldsymbol{T}}\boldsymbol{b}\ \right)^2 = \left(a^{(1)}b^{(1)} + a^{(2)}b^{(2)}\right)^2 = \left(a^{(1)}b^{(1)}\right)^2 + \left(a^{(2)}b^{(2)}\right)^2 + 2a^{(1)}a^{(2)}b^{(1)}b^{(2)} = \left(a^{(1)}\right)^2\left(b^{(1)}\right)^2 + \left(a^{(2)}\right)^2\left(b^{(2)}\right)^2 + \sqrt{2}a^{(1)}a^{(2)}\sqrt{2}b^{(1)}b^{(2)} = \begin{bmatrix} {a^{(1)}}^2 & {a^{(2)}}^2 & \sqrt{2}a^{(1)}a^{(2)} \end{bmatrix}\begin{bmatrix} {b^{(1)}}^2 \\ {b^{(2)}}^2 \\ \sqrt{2}b^{(1)}b^{(2)} \end{bmatrix} = \boldsymbol{\phi(a)^T\phi(b)}$
- Thus, $k_\phi(\boldsymbol{a},\boldsymbol{b}) = \left(\boldsymbol{a}^{\boldsymbol{T}}\boldsymbol{b}\ \right)^2$ implies the transformation

$$\boldsymbol{\phi(u)} = \begin{bmatrix} {u^{(1)}}^2 \\ {u^{(2)}}^2 \\ \sqrt{2}u^{(1)}u^{(2)} \end{bmatrix}$$

Other Kernels

- We can change the definition of dot products to any other function
  - Each kernel will have its own underlying feature representation
    - Formally: Moore–Aronszajn theorem



Kernel	Equation
Linear	$K(x, y) = x \cdot y$
Sigmoid	$K(x, y) = \tanh(ax.y + b)$
Polynomial	$K(x, y) = (1 + x \cdot y)^d$
KMOD	$K(x, y) = a\left[\exp\left(\frac{\gamma}{\lvert\lvert x-y \rvert\rvert^2+\sigma^2}\right) - 1\right]$
RBF	$K(x, y) = \exp(-a\lvert\lvert x - y\rvert\rvert^2)$
Exponential RBF	$K(x, y) = \exp(-a\lvert\lvert x - y\rvert\rvert)$

Requirements for being a kernel

Any function k can be a kernel if its pairwise kernel or 'Gram' matrix

$$K = \begin{bmatrix} k(x_1, x_1) & k(x_1, x_2) & k(x_1, x_3) \\ k(x_2, x_1) & k(x_2, x_2) & k(x_2, x_3) \\ k(x_3, x_1) & k(x_3, x_2) & k(x_3, x_3) \end{bmatrix}$$

is symmetric, positive semi-definite (for all given data).
And for any valid kernel, a corresponding transformation $\boldsymbol{x} \rightarrow \boldsymbol{\phi}(\boldsymbol{x})$ exists such that $\mathrm{k}(\boldsymbol{a}, \boldsymbol{b}) = \boldsymbol{\phi}(\boldsymbol{a})^T \boldsymbol{\phi}(\boldsymbol{b})$.

Kernels and their underlying transformations

Kernel	Transform (for 2D Input)
Linear: $k(\boldsymbol{a},\boldsymbol{b}) = \boldsymbol{a}^{\boldsymbol{T}}\boldsymbol{b}$	$\phi(\boldsymbol{u}) = \boldsymbol{u} = \left[u^{(1)} \quad u^{(2)}\right]^{\boldsymbol{T}}$
Polynomial degree 2: $k(\boldsymbol{a},\boldsymbol{b}) = \left(\boldsymbol{a}^{T}\boldsymbol{b}\right)^2$ (Homogeneous)	$\phi(\boldsymbol{u}) = \left[{u^{(1)}}^2 \quad {u^{(2)}}^2 \quad \sqrt{2}u^{(1)}u^{(2)}\right]^T$
Polynomial degree 2: $k(\boldsymbol{a},\boldsymbol{b}) = \left(\boldsymbol{a}^{T}\boldsymbol{b}+1\right)^2$	$\phi(\boldsymbol{u}) = \left[1 \;\; \sqrt{2}u^{(1)} \;\; \sqrt{2}u^{(2)} \;\; {u^{(1)}}^2 \;\; {u^{(2)}}^2 \;\; \sqrt{2}u^{(1)}u^{(2)}\right]^T$
RBF Kernel: $k(\boldsymbol{a},\boldsymbol{b}) = \exp(-\gamma\\|\boldsymbol{a}-\boldsymbol{b}\\|^2)$	Infinite dimensional (depending upon hyperpameter $\gamma > 0$ See: https://en.wikipedia.org/wiki/Radial_basis_function_kernel

Let's build a support vector machine on this idea!

Kernelized SVM: Representation

- We know that the discriminant function of the SVM can be written as:

$$f(\boldsymbol{x}) = w^T \boldsymbol{x} + b$$

- The Representer theorem (Scholkopf 2001) allows us to represent the weight vector as a linear combination of input vectors with each example's contribution weighted by a factor α_i

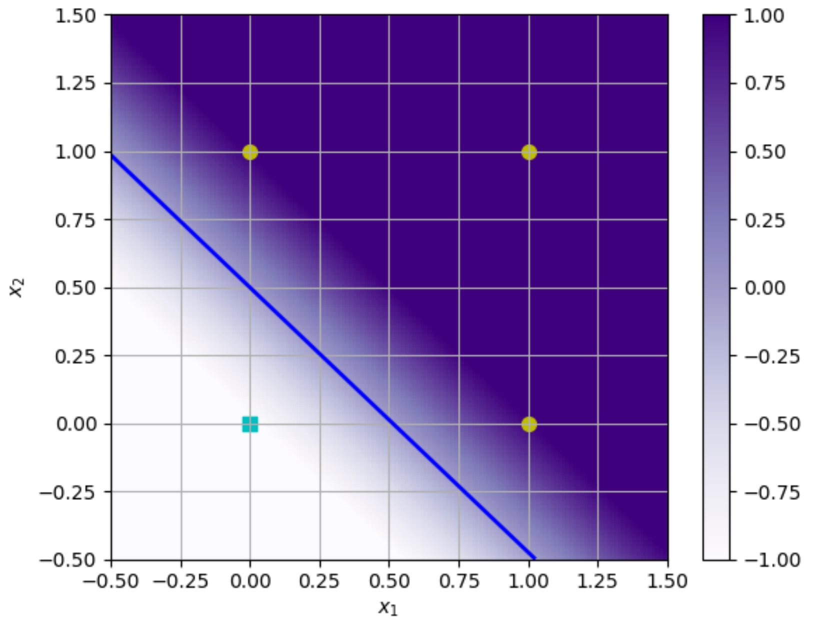
$$\boldsymbol{w} = \sum_{i=1}^{N} \alpha_i \boldsymbol{x}_i$$

Thus,

$$f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} + b = b + \sum_{j=1}^{N} \alpha_j \boldsymbol{x}_j^T \boldsymbol{x}$$

Notice how the prediction function involves only dot-products. Generalizing the dot product to a kernel function: $k(\boldsymbol{u}, \boldsymbol{v}) = \boldsymbol{u}^T \boldsymbol{v}$

$$f(\boldsymbol{x}) = b + \boldsymbol{w}^T \boldsymbol{x} = b + \sum_{j=1}^{N} \alpha_j k(\boldsymbol{x}_j, \boldsymbol{x})$$



$$\mathbf{x}^i = \begin{bmatrix}0\\0\end{bmatrix} \quad \begin{bmatrix}0\\1\end{bmatrix} \quad \begin{bmatrix}1\\0\end{bmatrix} \quad \begin{bmatrix}1\\1\end{bmatrix}$$

$$y = -1 \quad +1 \quad +1 \quad +1$$

$$\alpha_i = -2 \quad +1 \quad +1 \quad 0$$

$$\boldsymbol{w} = \sum_{i=1}^{N} \alpha_i \boldsymbol{x}_i = \begin{bmatrix}1\\1\end{bmatrix}$$

$$\boldsymbol{b} = -1$$

Kernel SVM: Optimization

$$f(\boldsymbol{x}) = b + \boldsymbol{w}^T\boldsymbol{x} = b + \sum_{j=1}^{N} \alpha_j k(\boldsymbol{x}_j, \boldsymbol{x})$$

$$\boldsymbol{w}^T\boldsymbol{w} = \left(\sum_{i=1}^{N} \alpha_i \boldsymbol{x}_i\right)^T \sum_{j=1}^{N} \alpha_j \boldsymbol{x}_j = \sum_{i,j=1}^{N} \alpha_i \alpha_j k(\boldsymbol{x}_i, \boldsymbol{x}_j)$$

$$\boxed{\min_w \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + \frac{C}{N}\sum_{i=1}^{N} \max\{0, 1 - y_i f(\boldsymbol{x}_i; \boldsymbol{w})\}} \xrightarrow{\boldsymbol{w} = \sum_{i=1}^{N} \alpha_i \boldsymbol{x}_i} \boxed{\min_{\alpha,b} \frac{1}{2}\sum_{i,j=1}^{N} \alpha_i \alpha_j k(\boldsymbol{x}_i, \boldsymbol{x}_j) + \frac{C}{N}\sum_{i=1}^{N} \max\left\{0, 1 - y_i\left(b + \sum_{j=1}^{N} \alpha_j k(\boldsymbol{x}_i, \boldsymbol{x}_j)\right)\right\}}$$

$$\max\left\{0, 1 - y_i\left(b + \sum_{j=1}^{N} \alpha_j k(\boldsymbol{x}_i, \boldsymbol{x}_j)\right)\right\}$$

Kernel SVM: Optimization with GD

$$\min_{\boldsymbol{\alpha},b} D(\boldsymbol{\alpha},\boldsymbol{b}) = \frac{1}{2}\sum_{i,j=1}^{N}\alpha_i\alpha_j k\left(\boldsymbol{x}_i,\boldsymbol{x}_j\right) + \frac{C}{N}\sum_{i=1}^{N}\max\left\{0, 1 - y_i\left(b + \sum_{j=1}^{N}\alpha_j k\left(\boldsymbol{x_i},\boldsymbol{x}_j\right)\right)\right\}$$

$$\nabla_{\alpha_i} D = \sum_{j=1}^{N}\alpha_j k\left(\boldsymbol{x}_i,\boldsymbol{x}_j\right) + \begin{cases} -\frac{C}{N}\sum_{j=1}^{N} y_j k\left(\boldsymbol{x_i},\boldsymbol{x}_j\right) & if\ 1 - y_i\left(b + \sum_{j=1}^{N}\alpha_j k\left(\boldsymbol{x_i},\boldsymbol{x}_j\right)\right) > 0 \\ 0 & else \end{cases}$$

$$\nabla_b D = \begin{cases} -\frac{C}{N}\sum_{j=1}^{N} y_j & if\ 1 - y_i\left(b + \sum_{j=1}^{N}\alpha_j k\left(\boldsymbol{x_i},\boldsymbol{x}_j\right)\right) > 0 \\ 0 & else \end{cases}$$

$$\boldsymbol{\alpha}^{(\boldsymbol{m})} \leftarrow \boldsymbol{\alpha}^{(\boldsymbol{m-1})} - \eta\nabla\boldsymbol{D}_{\boldsymbol{\alpha}}\left(\boldsymbol{\alpha}^{(\boldsymbol{k-1})}\right)$$

Kernelized SVM

- Things to note In this formulation

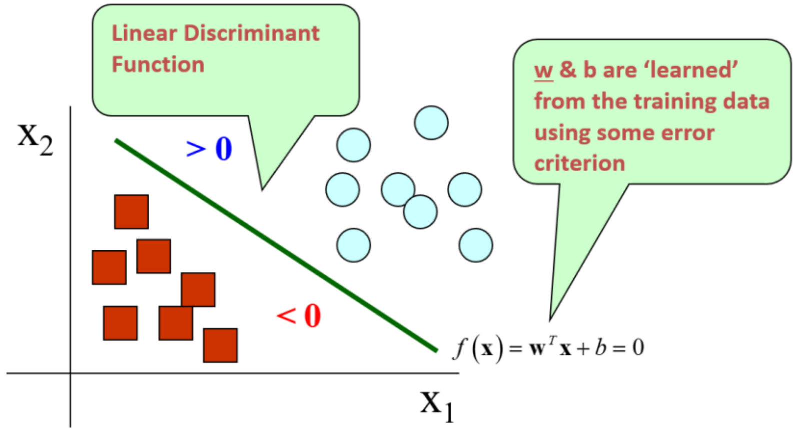
$$\min_{\boldsymbol{\alpha},b} = \frac{1}{2}\sum_{i,j=1}^{N}\alpha_i\alpha_j k(\boldsymbol{x}_i,\boldsymbol{x}_j) + \frac{C}{N}\sum_{i=1}^{N}\max\left\{0,1-y_i\left(b+\sum_{j=1}^{N}\alpha_j k(\boldsymbol{x_i},\boldsymbol{x_j})\right)\right\}$$

- The weight vector is not present
  - The formulation only involves dot products or kernel function values
  - Thus, we do not need explicit feature representations
- All the dot products have been replaced with a kernel function $k(\boldsymbol{x}_j,\boldsymbol{x}_i)$
- We assume that we know $k(\boldsymbol{x}_i,\boldsymbol{x}_j)$ for any two given training examples
- The optimization solution will be to obtain $\boldsymbol{\alpha}$ **and** b
- Once we solve the optimization problem, we can calculate the prediction score for any example based only on its kernel function values with training examples

$$f(\boldsymbol{x}) = b + \sum_{j=1}^{N}\alpha_j k(\boldsymbol{x}_j,\boldsymbol{x})$$

REO For SVM

- **Representation**
  - Features
  - Discriminant
    - Linear $f(\mathbf{x};\boldsymbol{w}) = w_1\boldsymbol{x}_1 + w_2\boldsymbol{x}_2 + \cdots + w_d\boldsymbol{x}_d = \mathbf{w}^T\mathbf{x}$

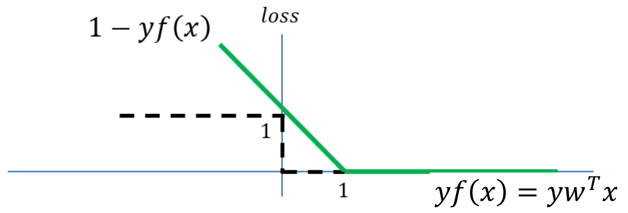


Linear Discriminant Function
w & b are 'learned' from the training data using some error criterion
x_2
> 0
< 0
$f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b = 0$
x_1

- **Evaluation**
  - Hinge Loss $l(f(x),y) = \begin{cases} 0 & yf(\boldsymbol{x}) > 1 \\ 1 - yf(\boldsymbol{x}) & yf(\boldsymbol{x}) \le 1 \end{cases} = \max\{0, 1 - y(\boldsymbol{w}^T\boldsymbol{x})\}$
  - Regularization $\min_w P(\boldsymbol{w}) = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + \frac{C}{N}\sum_{i=1}^{N}\max\{0, 1 - y_i f(\boldsymbol{x}_i;\mathbf{w})\}$
  - SRM:

Regularization Empirical Error

$$\min_w P(\boldsymbol{w}) = \frac{\lambda}{2}\boldsymbol{w}^T\boldsymbol{w} + \frac{1}{N}\sum_{i=1}^{N}\max\{0, 1 - y_i f(x_i;\boldsymbol{w})\}$$



Hyperparameter λ or C: Control the relative weighting of the Regularization and Empirical Error Minimization terms

- **Optimization**
  - Using Gradient Descent $\boldsymbol{w}^{(k)} \leftarrow \boldsymbol{w}^{(k-1)} - \alpha\nabla\boldsymbol{P}\left(\boldsymbol{w}^{(k-1)}\right)$

$$\nabla\boldsymbol{P} = \nabla_{\boldsymbol{w}}\left(\frac{\lambda}{2}w^Tw\right) + \frac{1}{N}\sum_{i=1}^{N}\nabla_{\boldsymbol{w}}l(f(\boldsymbol{x}_i;\mathbf{w})), y_i)$$

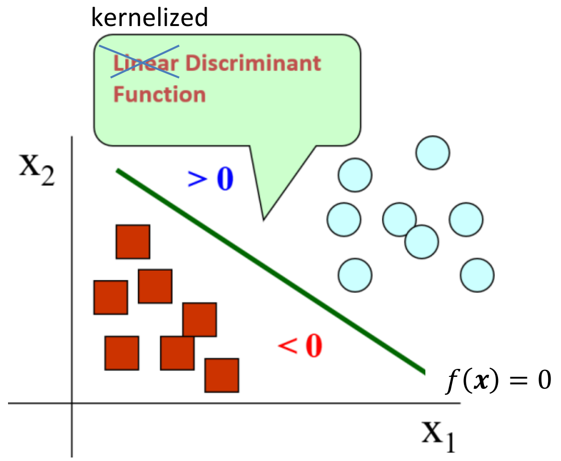
$$\nabla_{\boldsymbol{w}}\left(\frac{\lambda}{2}w^Tw\right) = \lambda w \qquad \nabla_{\boldsymbol{w}}\max\{0, 1 - y(\boldsymbol{w}^T\boldsymbol{x})\} = \begin{cases} 0 & 1 - yf(\boldsymbol{x};\mathbf{w}) < 0 \\ -y\boldsymbol{x} & else \end{cases}$$

$$\boldsymbol{w}_k \leftarrow \boldsymbol{w}_{k-1} - \alpha\lambda\boldsymbol{w}_{k-1} - \alpha\mathbf{1}(yf(\boldsymbol{x}) < 1)(y\boldsymbol{x})$$

REO For Kernelized SVM

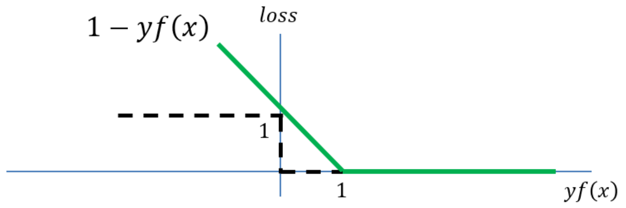
- **Representation**
  - Features
  - Discriminant

$$f(\boldsymbol{x}) = \sum_{j=1}^{N} \alpha_j k(\boldsymbol{x}_j\ , \boldsymbol{x})$$



- **Evaluation**
  - Hinge Loss

$$l(f(x), y) = \begin{cases} 0 & yf(\boldsymbol{x}) > 1 \\ 1 - yf(\boldsymbol{x}) & yf(\boldsymbol{x}) \le 1 \end{cases}$$



  - SRM:

$$D(\boldsymbol{\alpha}) = \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i \alpha_j k(\boldsymbol{x}_i, \boldsymbol{x}_j) + \frac{C}{N}\sum_{i=1}^{N} \max\left\{0, 1 - y_i\left(\sum_{j=1}^{N} \alpha_j k(\boldsymbol{x}_i, \boldsymbol{x}_j)\right)\right\}$$

Regularization Empirical Error

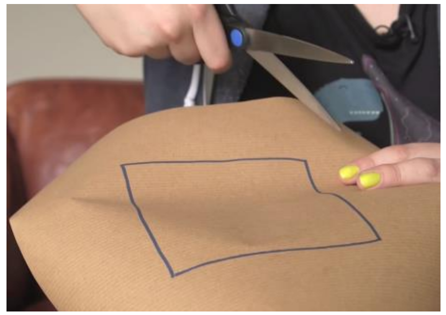
- **Optimization**
  - Using Gradient Descent

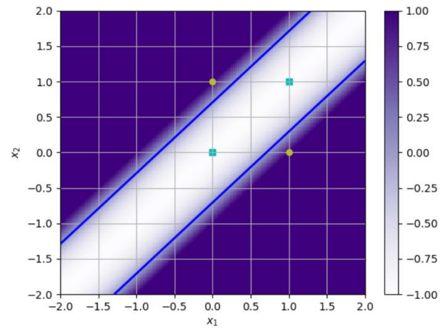
$$\boldsymbol{\alpha}^{(k)} \leftarrow \boldsymbol{\alpha}^{(k-1)} - \alpha \nabla \boldsymbol{D}\left(\boldsymbol{\alpha}^{(k-1)}\right) \qquad \nabla \boldsymbol{D} = \nabla_{\boldsymbol{\alpha}} D(\boldsymbol{\alpha})$$

https://github.com/foxtrotmike/CS909/blob/master/kernelizedSVM_pytorch.py (uses PyTorch for optimization so ignore for now!)
https://github.com/foxtrotmike/CS909/blob/master/mosvm_pytorch.ipynb

But how does a kernelized SVM achieve nonlinear classification?

- Put simply, a kernel $k(\boldsymbol{a}, \boldsymbol{b})$ is simply a way of quantifying the degree of similarity between two examples or objects
  - If we can change the definition of how similar two things are (by switching to a different kernel), we can achieve an implicit transformation of the example that may allow us to solve non-linear classification problems
- Choosing a kernel function allows us to not worry about defining explicit transformations to achieve non-linear separation
  - Moore–Aronszajn theorem states that for every kernel an underlying feature transformation exists.
  - **A way of achieving a paper fold!!**
- Together with the fold-and-cut theorem, this means that
  - ***If I choose my kernel appropriately, I should be able to achieve non-linear classification no matter how complex the data!***
- ***Thus, an (appropriately) kernelized SVM can, in principle, memorize any training data set***
- ***However, being based on Structural Risk Minimization, an SVM comes with a good regularization control to help it generalize!!***





https://www.youtube.com/watch?v=ZREp1mAPKTM

The Fold-and-Cut Theorem implies that any pattern can be achieved with a single straight cut if the paper (or space) is folded appropriately.

Thus, it is theoretically possible to partition any space into regions containing positive and negative training examples no matter how complex such a boundary is by simply folding the feature space appropriately and using a linear classifier (single straight cut).

An SVM allows us to do it by using kernel functions.

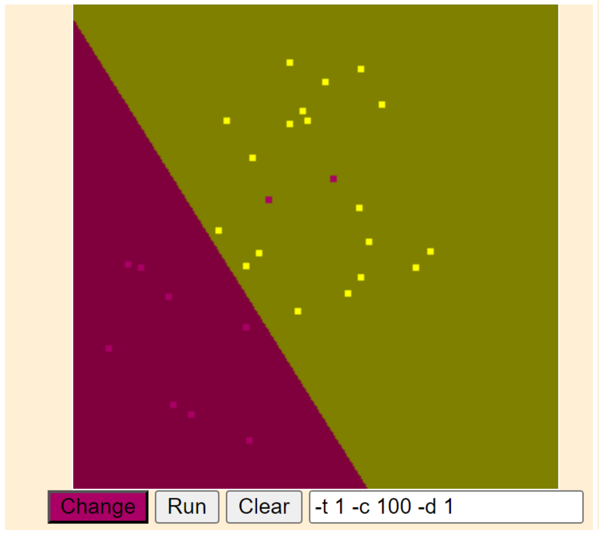
Where does the name SVM Come From?

- The training examples for which the values of α_i are non-zero after optimization are the only ones contributing to the decision

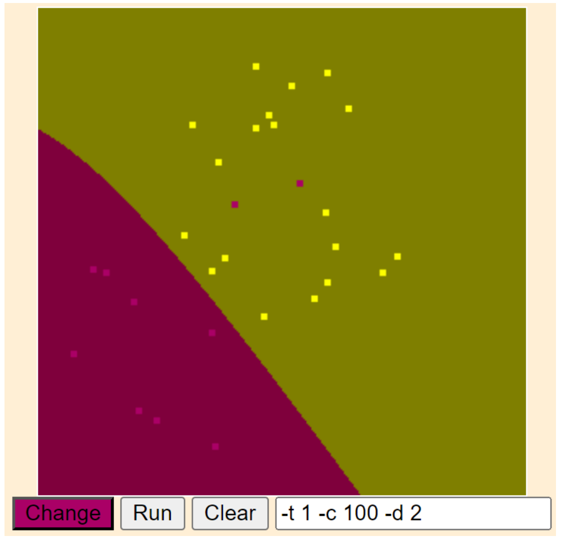
$$f(\boldsymbol{x}) = b + \sum_{j=1}^{N} \alpha_j k(\boldsymbol{x}_j, \boldsymbol{x})$$

- These examples are called “Support Vectors” as they support the decision or prediction!

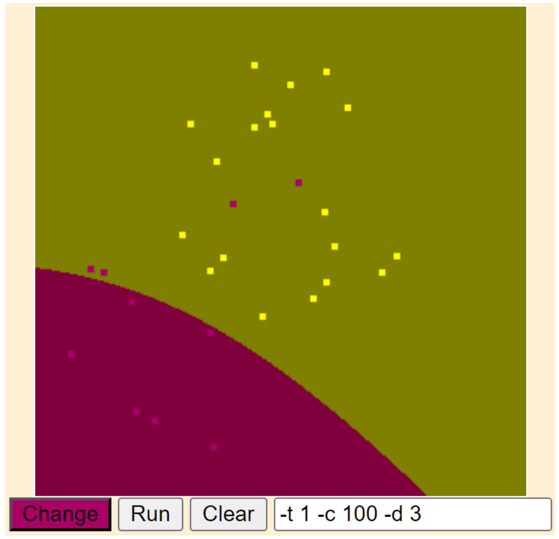
- Use the Applet at: https://www.csie.ntu.edu.tw/~cjlin/libsvm/
- Study the impact of changing kernel type, kernel hyperparameters and C



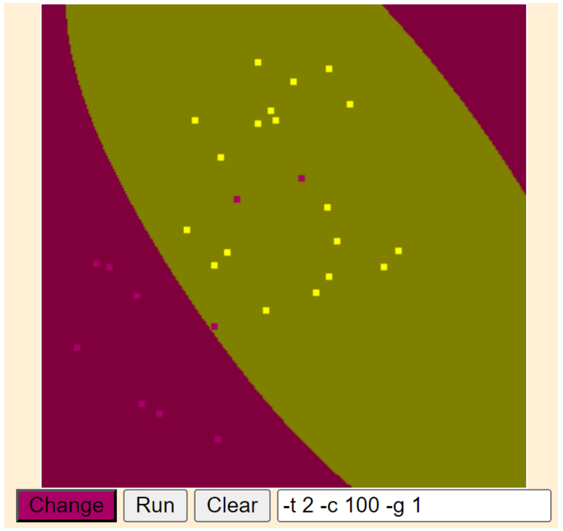
-t 1 -c 100 -d 1



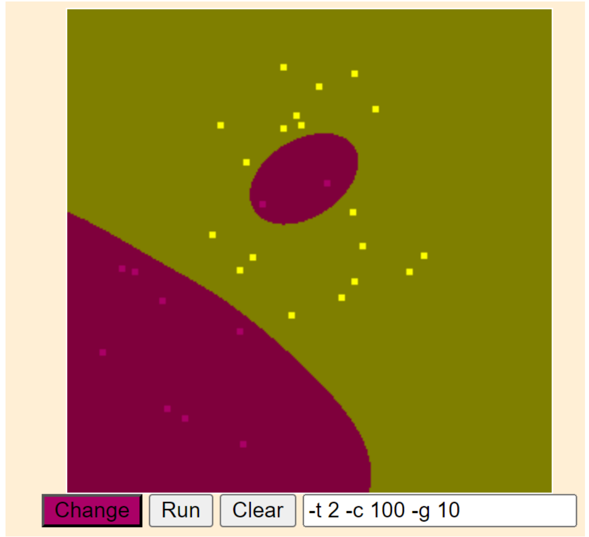
-t 1 -c 100 -d 2



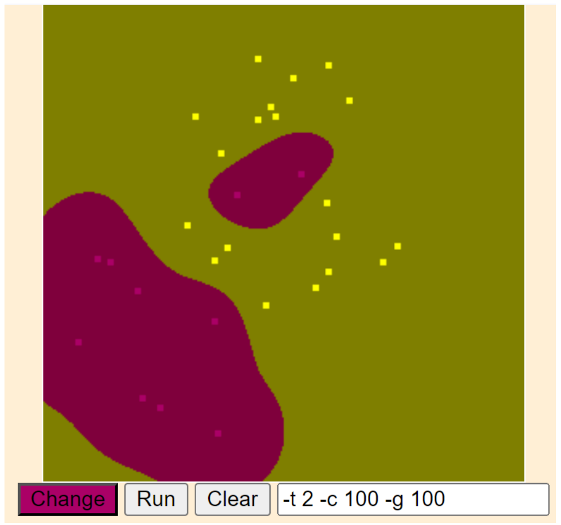
-t 1 -c 100 -d 3



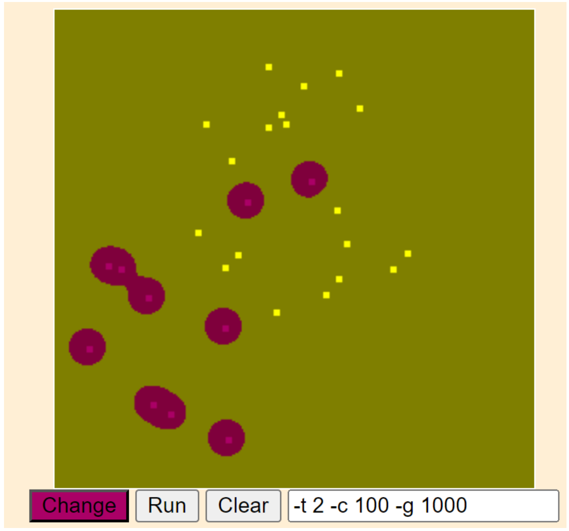
-t 2 -c 100 -g 1



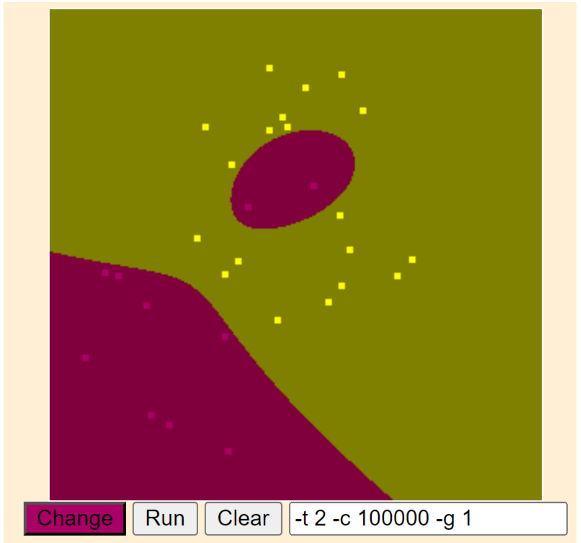
-t 2 -c 100 -g 10



-t 2 -c 100 -g 100



-t 2 -c 100 -g 1000



-t 2 -c 100000 -g 1

Example: Solution of the OR problem

```
import numpy as np
from sklearn.svm import SVC as Classifier

X = np.array([[0,0],[0,1],[1,0],[1,1]])
y =np.array([-1,1,1,1])
clf = Classifier(kernel = 'poly', degree = 1, C = 10).fit(X, y)
plotit(X,y,clf = clf.decision_function,conts=[0],extent=[-2,+2,-2,+2])
print("Alpha: ",clf.dual_coef_)
print(clf.support_vectors_)
print(clf.intercept_)
```

$$k(a,b) = a^T b$$

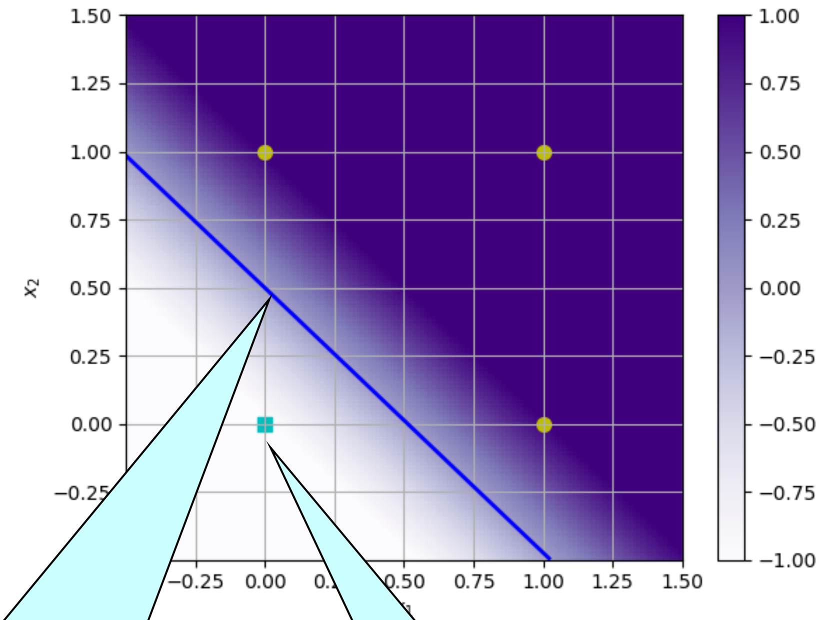
$$\mathbf{x}^i = \begin{bmatrix}0\\0\end{bmatrix} \quad \begin{bmatrix}0\\1\end{bmatrix} \quad \begin{bmatrix}1\\0\end{bmatrix} \quad \begin{bmatrix}1\\1\end{bmatrix}$$

$$y = -1 \quad +1 \quad +1 \quad +1$$

$$\alpha_i = -2 \ +1 \ +1 \quad 0$$

$$\mathbf{w}^* = \begin{bmatrix}1\\1\end{bmatrix}$$

$$b^* = -1$$



Optimal separating boundary

One of the 3 SVs

See: https://github.com/foxtrotmike/svmtutorial/blob/master/svmtutorial.ipynb

XOR

```
import numpy as np
from sklearn.svm import SVC as Classifier

X = np.array([[0,0],[0,1],[1,0],[1,1]])
y =np.array([-1,1,1,-1])
clf = Classifier(kernel = 'poly', degree = 2, C = 1).fit(X, y)
plotit(X,y,clf = clf.decision_function,conts=[0],extent=[-2,+2,-2,+2])
print("Alpha: ",clf.dual_coef_)
print(clf.support_vectors_)
print(clf.intercept_)
```

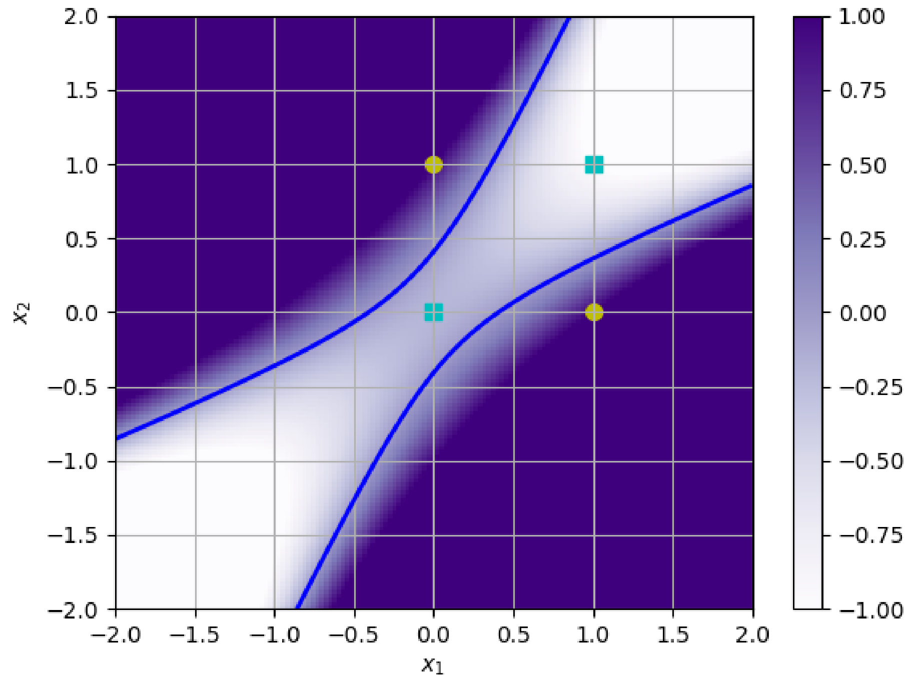
$$k(a,b) = (a^T b)^2$$

$$\mathbf{x}^i = \begin{bmatrix}0\\0\end{bmatrix} \quad \begin{bmatrix}0\\1\end{bmatrix} \quad \begin{bmatrix}1\\0\end{bmatrix} \quad \begin{bmatrix}1\\1\end{bmatrix}$$

$$y = \ -1 \quad +1 \quad +1 \quad -1$$

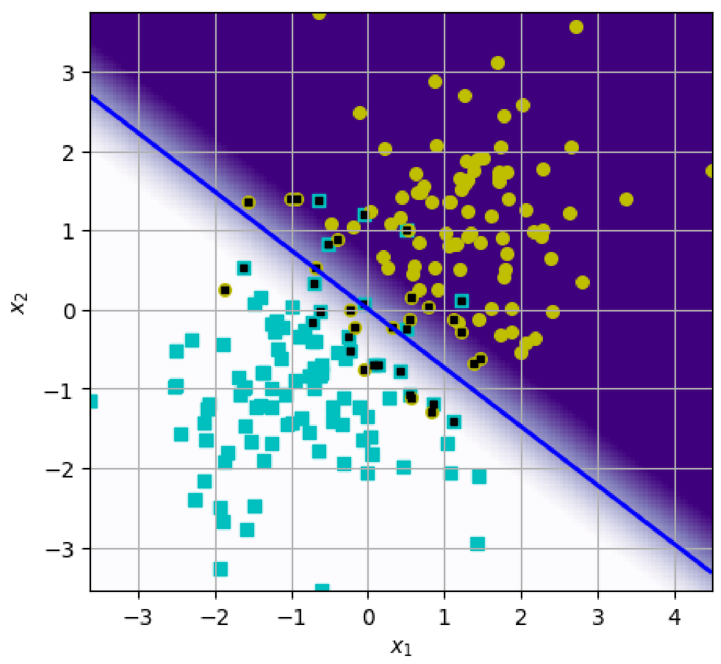
$$\alpha_i = \ -1 \ +0.7 \ +0.7 \quad -0.4$$

$$b^* = -1$$

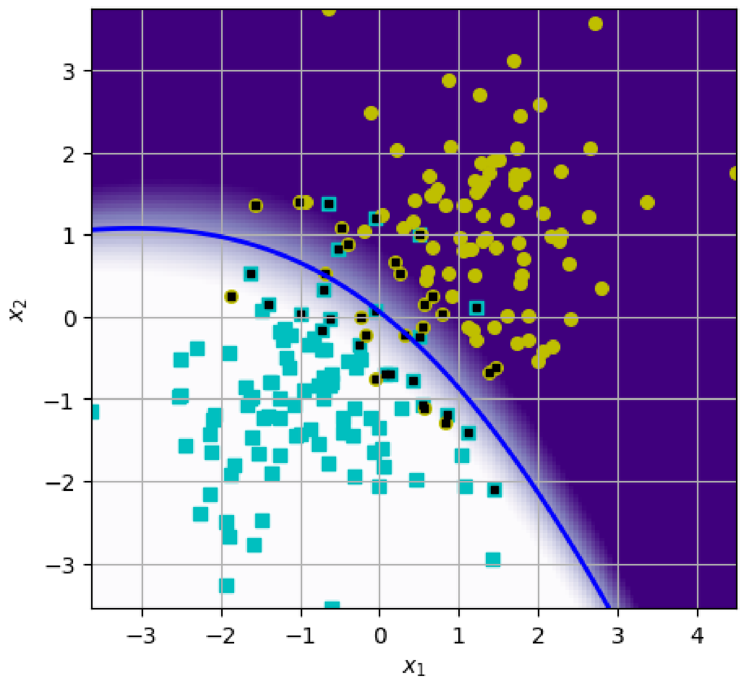


See: https://github.com/foxtrotmike/svmtutorial/blob/master/svmtutorial.ipynb

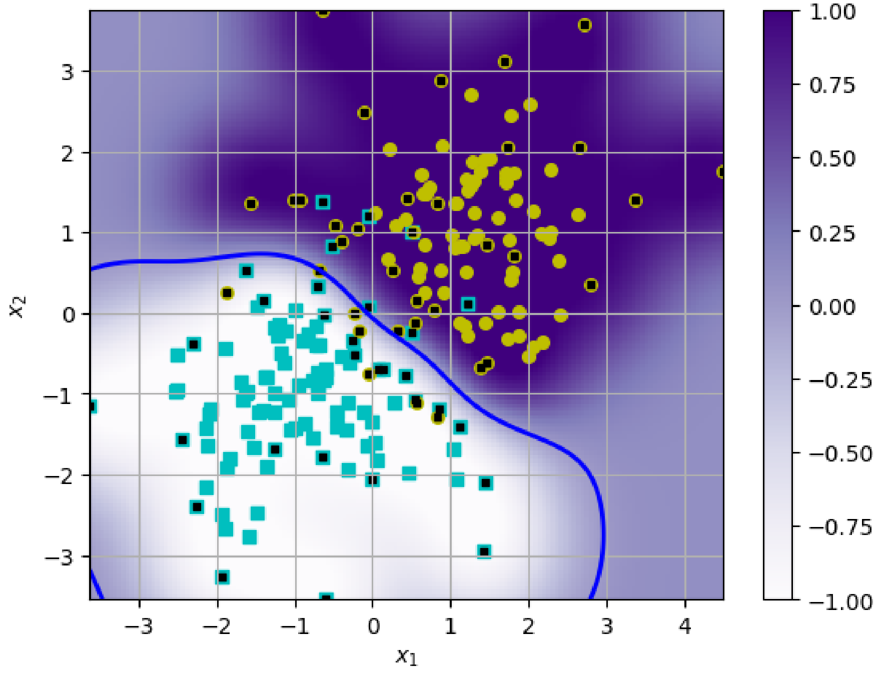
SVM with linear Kernel



SVM with polynomial Kernel

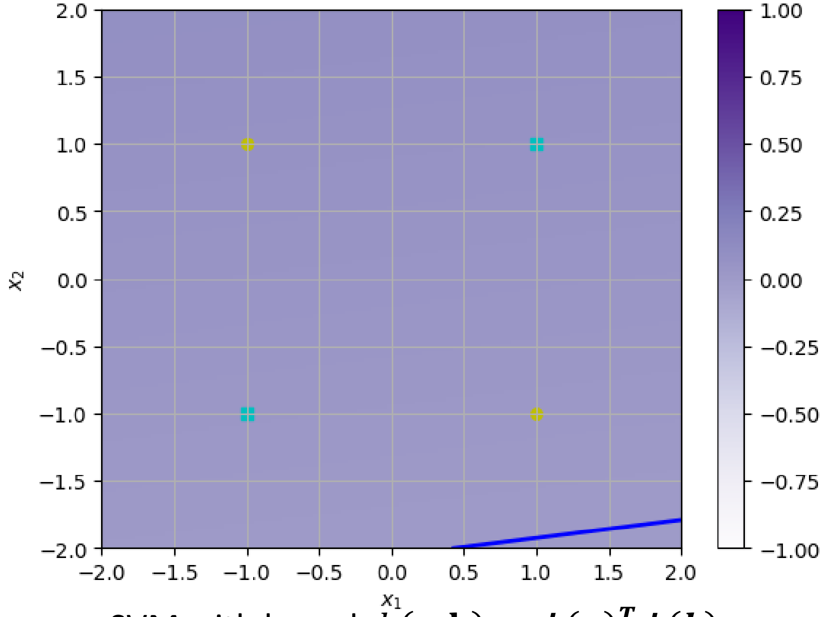


SVM with RBF Kernel

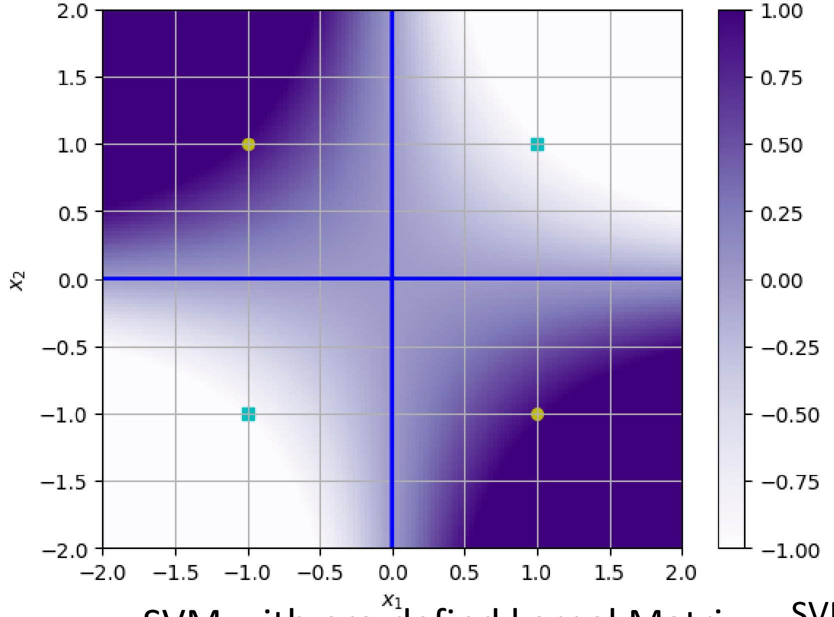


See: https://github.com/foxtrotmike/svmtutorial/blob/master/svmtutorial.ipynb

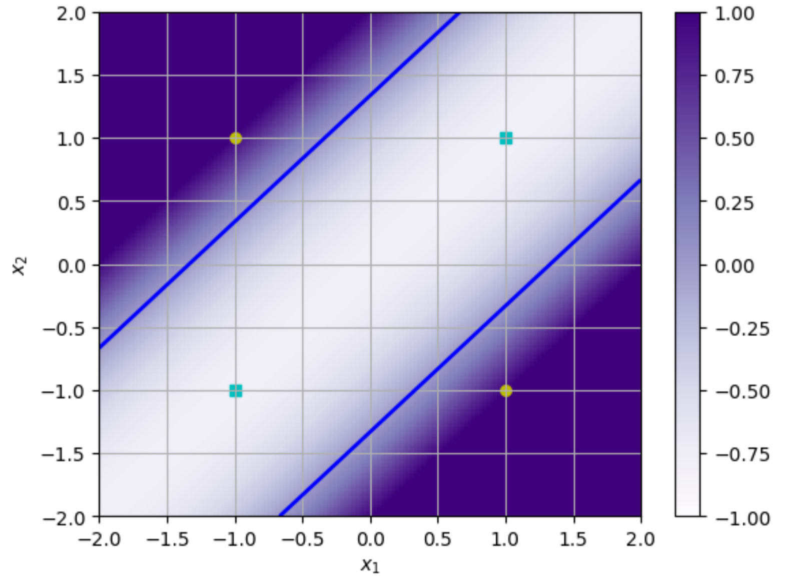
Linear SVM without Transform



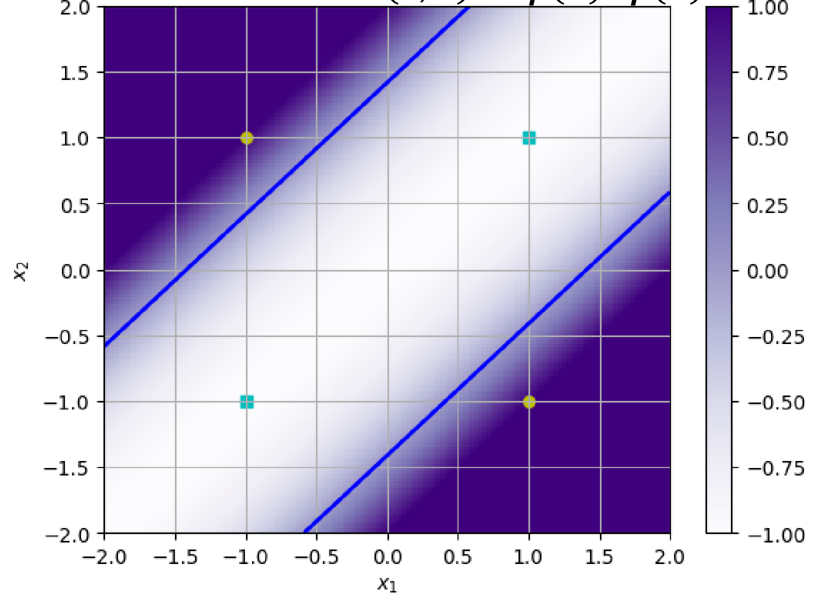
Linear SVM with transform
$\phi(\boldsymbol{u}) = \left[u^{(1)^2} \quad u^{(2)^2} \quad \sqrt{2}u^{(1)}u^{(2)}\right]^T$



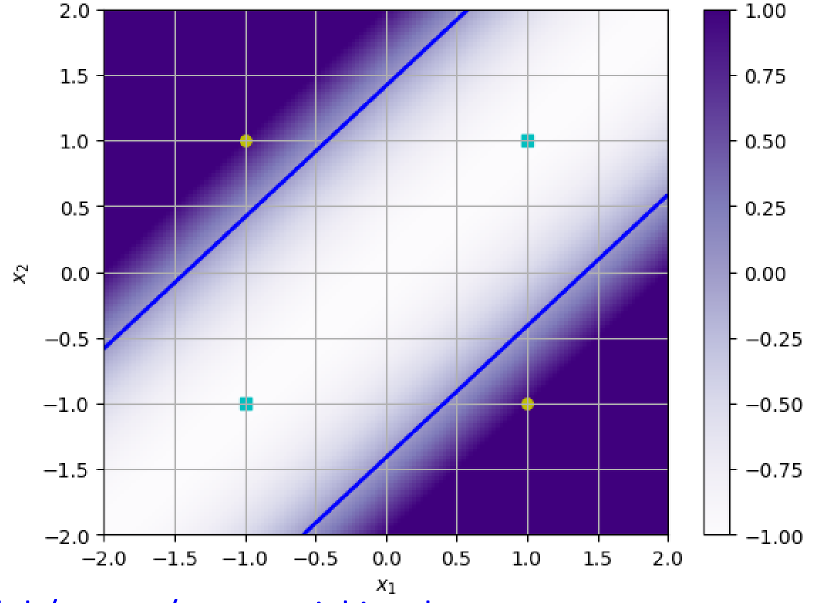
Linear SVM with transform
$\boldsymbol{\phi}(\boldsymbol{u}) = \left(\boldsymbol{u}^{(1)} + \boldsymbol{u}^{(2)} - 1\right)^2$



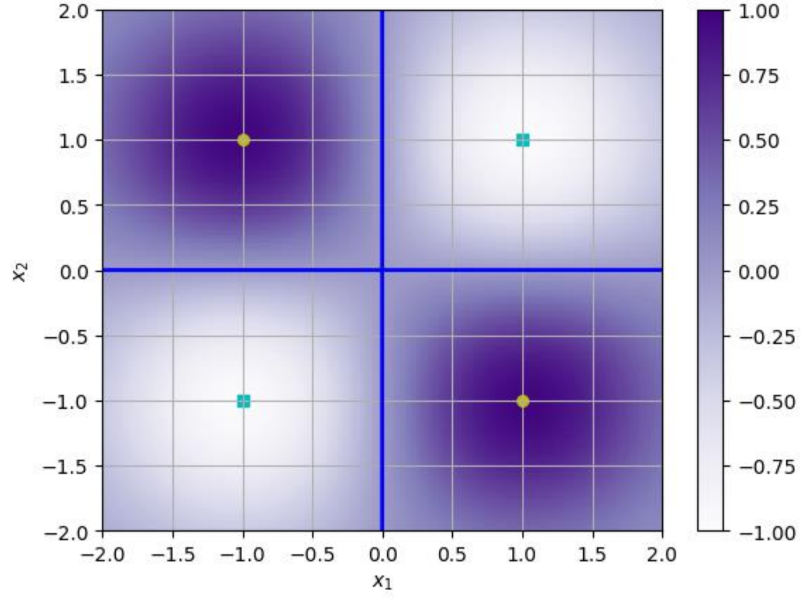
SVM with kernel: $k(\mathbf{a},\mathbf{b}) = \boldsymbol{\phi}(\boldsymbol{a})^T\boldsymbol{\phi}(\boldsymbol{b})$



SVM with pre-defind kernel Matrix



SVM with pre-defined kernel: $k(\mathbf{a},\mathbf{b}) = \exp(-\gamma\|a-b\|^2)$



See: https://github.com/foxtrotmike/svmtutorial/blob/master/svmtutorial.ipynb

Using the SVM

- Read:

- Ben-Hur, Asa, and Jason Weston. 2010. “A User’s Guide to Support Vector Machines.” In *Data Mining Techniques for the Life Sciences*, edited by Oliviero Carugo and Frank Eisenhaber, 223–39. Methods in Molecular Biology 609. Humana Press. http://dx.doi.org/10.1007/978-1-60327-241-4_13
- http://pyml.sourceforge.net/doc/howto.pdf

- Coding tutorial: https://github.com/foxtrotmike/svmtutorial/blob/master/svmtutorial.ipynb

Steps for Feature based Classification

- Prepare the pattern matrix X
- Select the kernel function to use
- Select the parameter of the kernel function and the value of *C*
  - You can use the values suggested by the SVM software, or you can set apart a validation set to determine the values of the parameter
- Execute the training algorithm and obtain the α_i
- Unseen data can be classified using the α_i and the support vectors

Choosing the Kernel Function

- Probably the trickiest part of using SVM.
- The kernel function is important because it creates the kernel matrix, which summarizes all the data
- In practice, a low degree polynomial kernel or RBF kernel with a reasonable width is a good initial try
- Use hyperparameter optimization over a validation set to choose a kernel

Handling data imbalance

- If the data is imbalanced (too much of one class and only a small number of examples from the other)
  - You can set an individual C for each class (called class weighting) or even per-example weighting
  - Can also be used to reflect a priori knowledge

$$\min_{\boldsymbol{\alpha},b} \frac{1}{2}\sum_{i,j=1}^{N} \alpha_i \alpha_j k(\boldsymbol{x}_i, \boldsymbol{x}_j) + \frac{C}{N}\sum_{i=1}^{N} \max\left\{0, 1 - y_i\left(b + \sum_{j=1}^{N} \alpha_j k(\boldsymbol{x}_i, \boldsymbol{x}_j)\right)\right\}$$

$$\min_{\boldsymbol{\alpha},b} \frac{1}{2}\sum_{i,j=1}^{N} \alpha_i \alpha_j k(\boldsymbol{x}_i, \boldsymbol{x}_j) + \sum_{i=1}^{N} c_i \max\left\{0, 1 - y_i\left(b + \sum_{j=1}^{N} \alpha_j k(\boldsymbol{x}_i, \boldsymbol{x}_j)\right)\right\}$$

per-example weighting

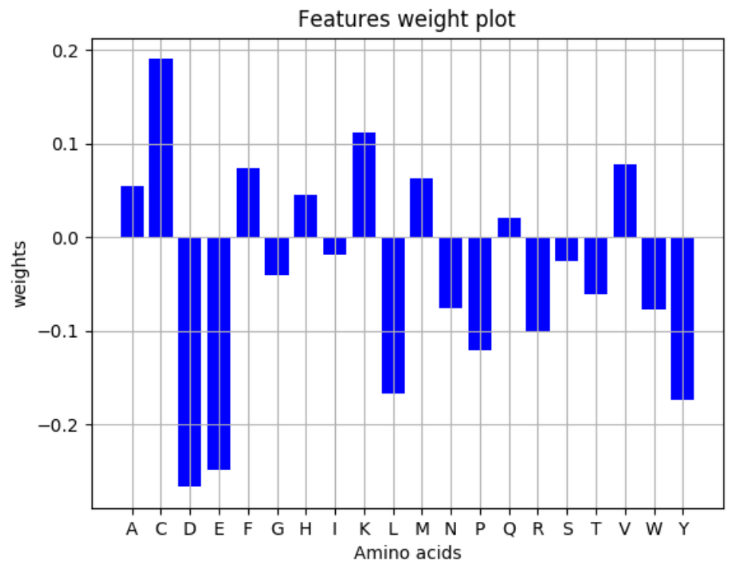
Coding: https://scikit-learn.org/stable/auto_examples/svm/plot_weighted_samples.html

Strengths and Weaknesses of SVM

- **Strengths**
  - Only a few training points (Support Vectors) determine the final boundary
  - Very useful is the amount of training data is small (esp. in biomedical domains)
  - Margin maximization and kernelized
  - Optimization is relatively easy: No local optimal, unlike in neural networks
  - It scales well to high dimensional data
  - Tradeoff between classifier complexity and error can be controlled explicitly (through C)
  - Non-traditional data like strings and trees can be used as input to SVM, instead of feature vectors as the SVM only requires defining a kernel or degree of similarity between examples
  - Completely interpretable and explainable
    - When using linear SVMs, the weight vector gives a clear indication of which features are important (if input data is appropriately scaled): $f(\boldsymbol{x};\boldsymbol{w}) = w_1 x^{(1)} + w_2 x^{(2)} + \cdots + w_d\, x^{(d)} + b$
    - When using non-linear SVMs, the decision can still be explained in terms of the degree of similarity to different training examples: $f(\boldsymbol{x};\boldsymbol{\alpha}) = \alpha_1 k(\boldsymbol{x},\boldsymbol{x}_1) + \alpha_2\alpha_1 k(\boldsymbol{x},\boldsymbol{x}_2) + \cdots + \alpha_N\, \alpha_1 k(\boldsymbol{x},\boldsymbol{x}_N) + b$

- **Weaknesses**
  - Need to choose a “good” kernel function.
  - Can be sensitive to data normalization and standardization
    - See: https://scikit-learn.org/stable/modules/preprocessing.html
  - Large scale data
    - Kernel Approximation Algorithms

“Weighted Sums of Random Kitchen Sinks: Replacing minimization with randomization in learning” by Recht and Rahimi, 2009



What are the underlying characteristics of an antimicrobial peptide? We can infer the relative importance of different amino acids in an antimicrobial peptide using the weights of the SVM.

Gull, Sadaf, Nauman Shamim, and Fayyaz Minhas. “AMAP: Hierarchical Multi-Label Prediction of Biologically Active and Antimicrobial Peptides.” *Computers in Biology and Medicine* 107 (April 1, 2019): 172–81. https://doi.org/10.1016/j.compbiomed.2019.02.018.

Advantages of kernels

- Once we replace the dot product with a kernel function (i.e., perform the kernel trick or ‘kernelize’ the formulation), the SVM formulation no longer requires any features!
- As long as you have a kernel function, everything works
  - Remember a kernel function is simply a mapping from two examples to a scalar
    - Tells us how similar the two examples are to each other

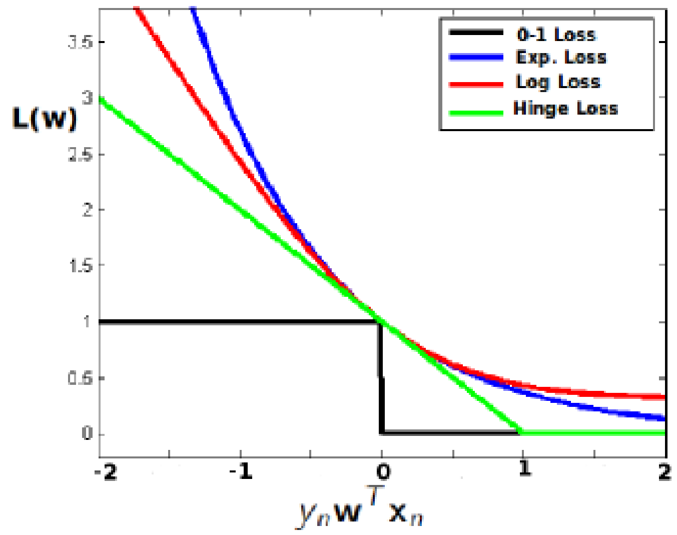
General Principle

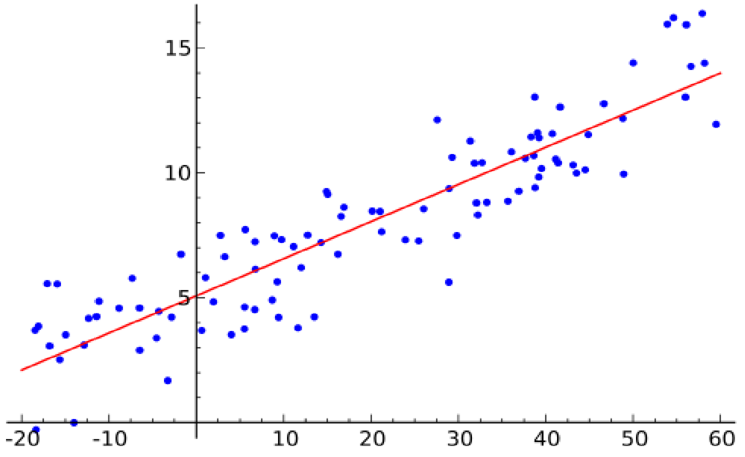
- Each machine learning model should have:
  - Empirical Error Minimization
  - Regularization

- Feature transformations ↔ Kernels ↔ Paper folding

What can we do with SRM?

- The principal of SRM allows us to develop a family of large margin learning machines by changing its components
- Example
- SVM: $\boldsymbol{min}_{\boldsymbol{w},\boldsymbol{b}}\ \frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \sum_{\boldsymbol{i}=\boldsymbol{1}}^{\boldsymbol{N}}[1 - y_i f(\boldsymbol{x_i})]_+$
- Regularized least square regression
  - $\boldsymbol{min}_{\boldsymbol{w},\boldsymbol{b}}\ \frac{\boldsymbol{\lambda}}{\boldsymbol{2}}\|\boldsymbol{w}\|^2 + \sum_{\boldsymbol{i}=\boldsymbol{1}}^{\boldsymbol{N}}\left(y_i - f(\boldsymbol{x_i})\right)^{\boldsymbol{2}}$
- Support Vector Regression
  - $\boldsymbol{min}_{\boldsymbol{w},\boldsymbol{b}}\ \frac{\boldsymbol{\lambda}}{\boldsymbol{2}}\|\boldsymbol{w}\|^{\boldsymbol{2}} + \sum_{\boldsymbol{i}=\boldsymbol{1}}^{\boldsymbol{N}}[|y_i - f(\boldsymbol{x_i})| - \epsilon]_+$
- Feature selection
  - $\boldsymbol{min}_{\boldsymbol{w},\boldsymbol{b}}\ \frac{\lambda}{2}\|\boldsymbol{w}\|_1^2 + \sum_{\boldsymbol{i}=\boldsymbol{1}}^{\boldsymbol{N}}[1 - y_i f(\boldsymbol{x_i})]_+$

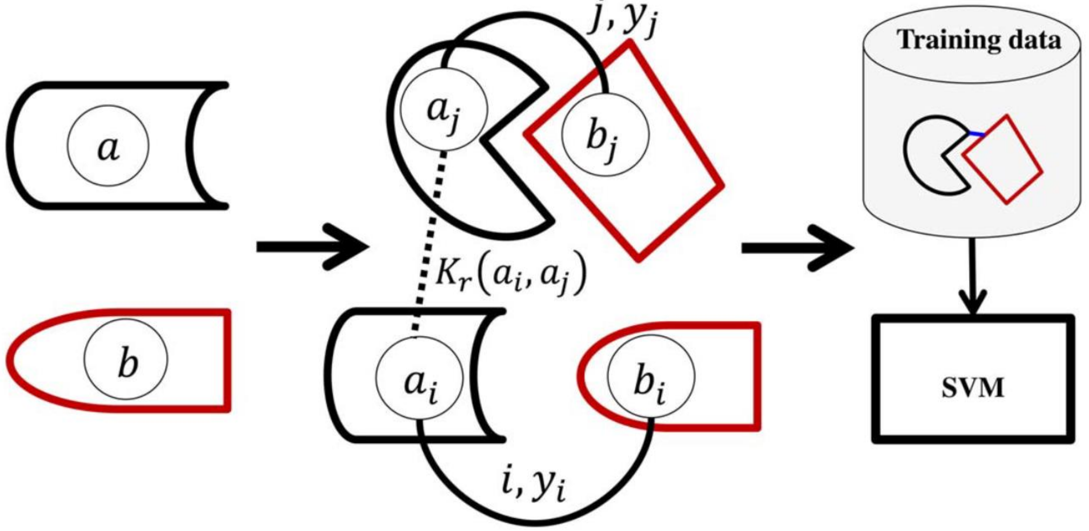


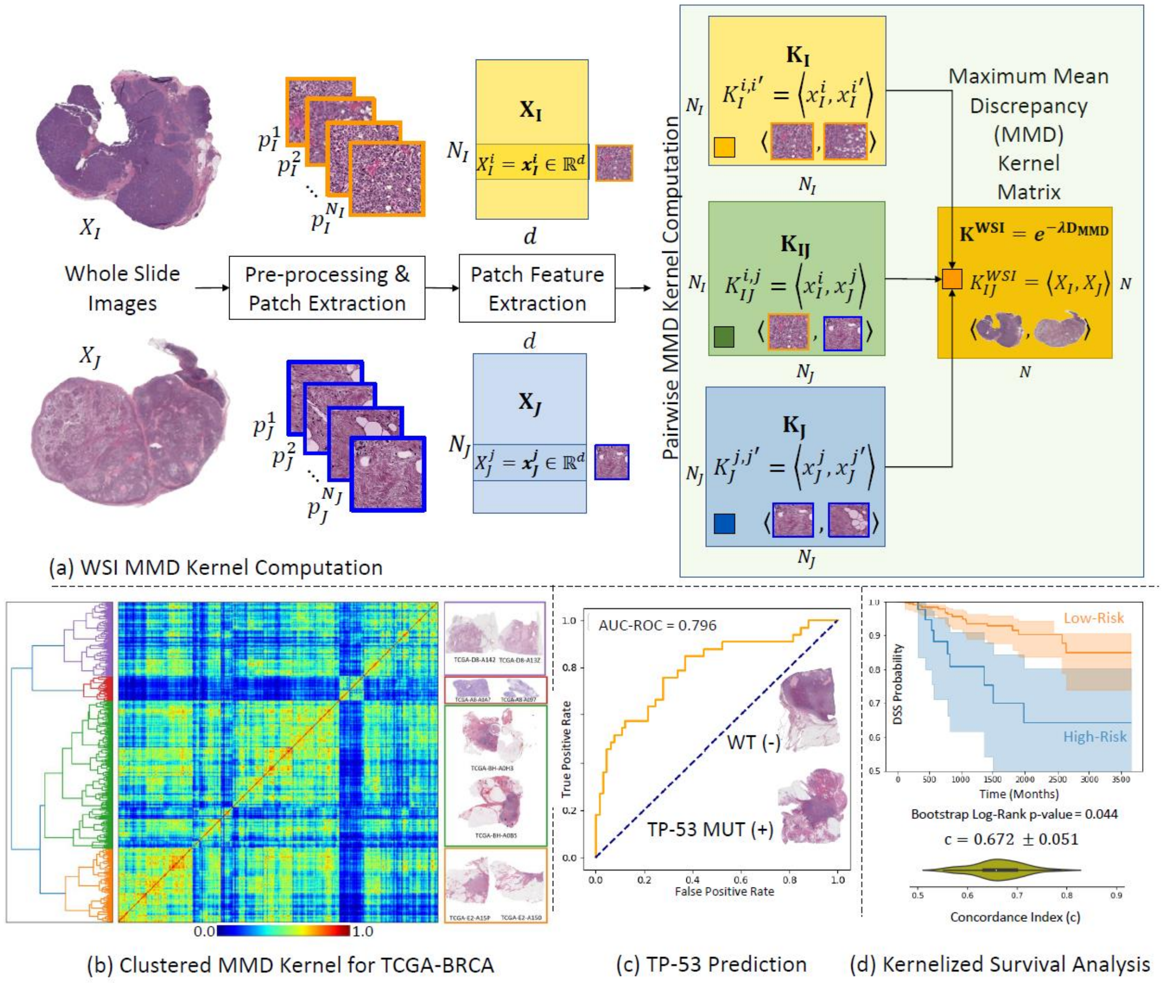


Regularizers

- Controls the complexity error of the classifier
- There are also other regularizers
  - $\|\boldsymbol{w}\|_{\boldsymbol{2}}^{\boldsymbol{2}} = w_1^{\boldsymbol{2}} + w_2^{\boldsymbol{2}} + \cdots + w_d^{\boldsymbol{2}}$
    - Convex, Smooth
  - $\|\boldsymbol{w}\|_{\boldsymbol{1}}^{\boldsymbol{1}} = |\boldsymbol{w_1}| + |\boldsymbol{w_2}| + \cdots + |\boldsymbol{w_d}|$
    - Used for feature reduction
    - "1-norm Support Vector Machine", Zhu et al. (2004)
  - $\|\boldsymbol{w}\|_{\boldsymbol{0}} = \boldsymbol{number\ of\ non - zero\ elements\ in\ w}$
    - Minimization of this norm will lead to feature selection
    - "Use of the Zero-Norm with Linear Models and Kernel Methods", JMLR, Weston et al., (2003)

Application Examples





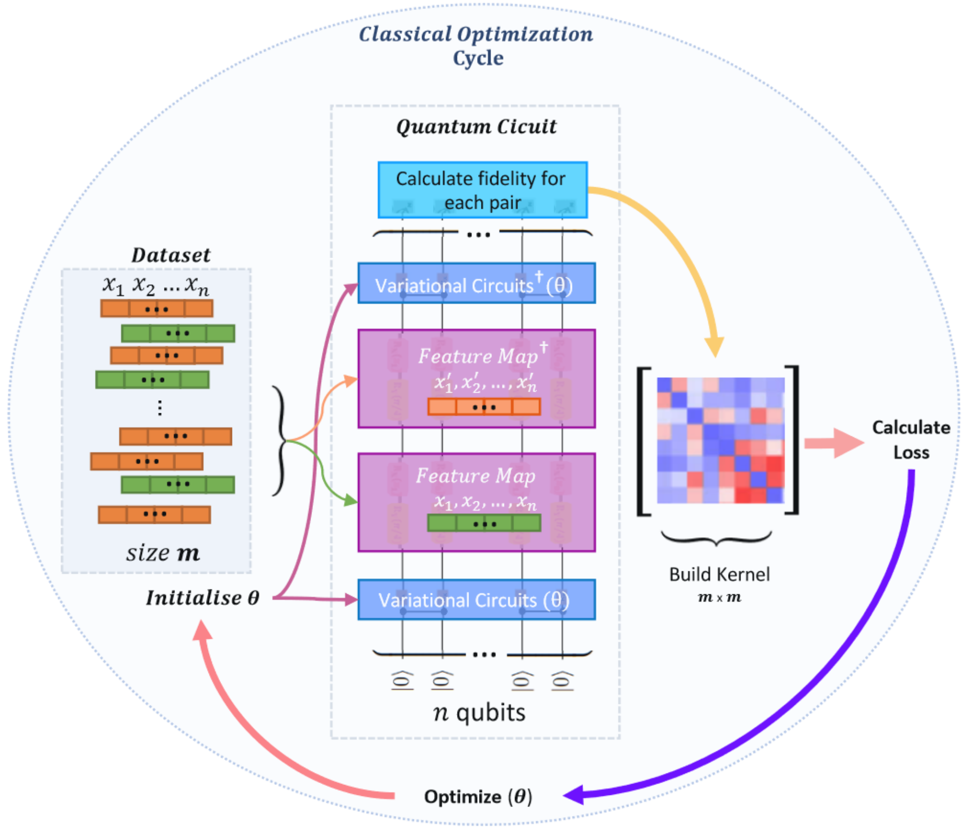
Whole Slide Images
Pre-processing & Patch Extraction
Patch Feature Extraction
Pairwise MMD Kernel Computation
Maximum Mean Discrepancy (MMD) Kernel Matrix
$\mathbf{K^{WSI}} = e^{-\lambda \mathbf{D_{MMD}}}$
$K_{IJ}^{WSI} = \langle X_I, X_J \rangle$
$K_I^{i,i'} = \langle x_I^i, x_I^{i'} \rangle$
$K_{IJ}^{i,j} = \langle x_I^i, x_J^j \rangle$
$K_J^{j,j'} = \langle x_J^j, x_J^{j'} \rangle$
(a) WSI MMD Kernel Computation
AUC-ROC = 0.796
WT (-)
TP-53 MUT (+)
True Positive Rate
False Positive Rate
Low-Risk
High-Risk
DSS Probability
Time (Months)
Bootstrap Log-Rank p-value = 0.044
c = 0.672 ± 0.051
Concordance Index (c)
(b) Clustered MMD Kernel for TCGA-BRCA
(c) TP-53 Prediction
(d) Kernelized Survival Analysis

PAIRpred: Partner-specific prediction of interacting residues from sequence and structure, Fayyaz Minhas, Brian Geiss and Asa Ben-Hur in Proteins: Structure, Function and Bioinformatics, vol. 82, no. 7, pp. 1142-1155, 2014 (Published Online: 2013).

Keller, Piotr, Muhammad Dawood, and Fayyaz ul Amir Afsar Minhas. "Maximum Mean Discrepancy Kernels for Predictive and Prognostic Modeling of Whole Slide Images." in proc. IEEE - ISBI 2023 International Symposium on Biomedical Imaging (ISBI), Columbia, April 2023. https://doi.org/10.48550/arXiv.2301.09624.

Why (still) study SVMs?

- Cover important concepts
- Future: Quantum Support Vector Machines
  - https://qiskit.org/documentation/stable/0.24/tutorials/machine_learning/01_qsvm_classification.html



Sahin, M. Emre, Benjamin C. B. Symons, Pushpak Pati, Fayyaz Minhas, Declan Millar, Maria Gabrani, Jan Lukas Robertus, and Stefano Mensa. "Efficient Parameter Optimisation for Quantum Kernel Alignment: A Sub-Sampling Approach in Variational Training." arXiv, January 5, 2024. https://doi.org/10.48550/arXiv.2401.02879.

End of Lecture

We want to make a machine that will be proud of us.

- Danny Hillis