

# Vector Space Classification

Rocchio, KNN, and SVM

---

CHIEN CHIN CHEN

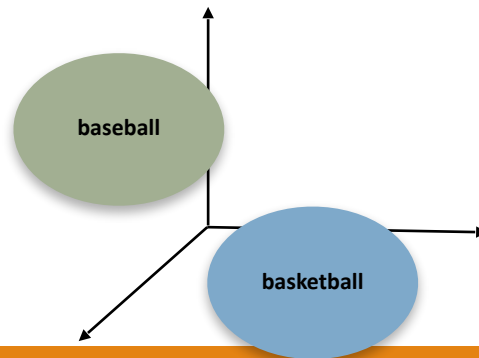
# Preface (1/4)

---

In text classification, classes usually can be distinguished by word patterns.

- Documents in the class **basketball** tend to heavily mention terms like shoot, rebound, and dunk.
- Documents in the class **baseball** tend to have a lot of bat, pitch, and hit.

In terms of **vector space model**, documents of the two classes **probably** form distinct regions in a high-dimension vector space.



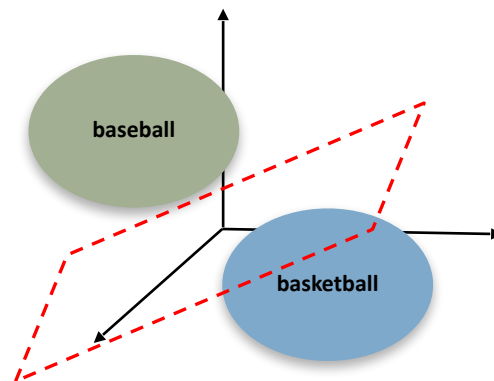
# Preface (2/4)

---

## ***Contiguity hypothesis:***

- Documents in the same class form a contiguous region.
- Regions of different classes do not overlap.

Vector space classification try to identify boundaries between the regions to classify new documents.



# Preface (3/4)

---

But if the document representation chosen is unfavorable, the contiguity hypothesis will not hold ...

- Then successful vector space classification may not be possible.
- Representation includes type of weighting, normalization... etc.

**Weighted** representation is always preferred

- In particular **tf-idf representation**.
- **Multi-hot** (binary) representation is not welcome because it ignores weights of terms.

# Preface (4/4)

---

Decisions of many vector space classifier are based on a notion of **distance**.

- E.g., the nearest neighbors in kNN classification.

Here, we mainly use **Euclidean distance** as the underlying distance measure.

$$|\underline{x} - \underline{y}| = \sqrt{\sum_{i=1}^M (x_i - y_i)^2}$$

For **length-normalized** vectors, there is a direct correspondence between **cosine similarity** and **Euclidean distance**.

$$\begin{aligned} |\underline{x} - \underline{y}| &= \sqrt{\sum_{i=1}^M (x_i - y_i)^2} \\ &= \sqrt{\sum_{i=1}^M (x_i^2 + y_i^2 - 2x_i y_i)} \\ &= \sqrt{\sum_{i=1}^M x_i^2 + \sum_{i=1}^M y_i^2 - 2 \sum_{i=1}^M x_i y_i} \\ &= \sqrt{2 - 2\text{cosine}(\underline{x}, \underline{y})} \end{aligned}$$

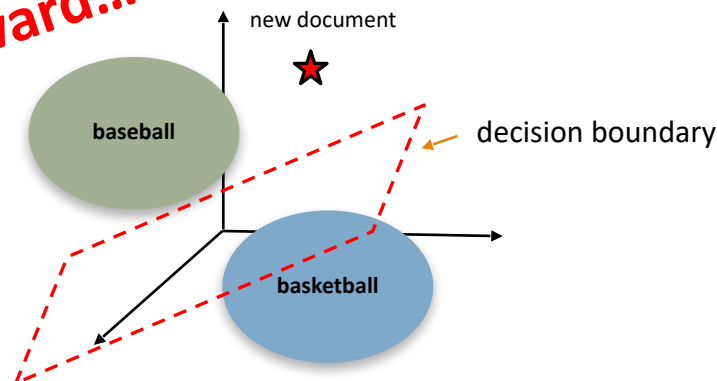
some methods also use **1 – cosine(x,y)** as the distance measure

# Rocchio Classification (1/9)

The main work in vector space classification is to define the boundaries between classes.

- The boundaries are also called ***decision boundaries***.
- To classify a new document, we then determine the region it occurs in, and assign it the class of that region.

**So Straightforward...**



**BUT ... HOW TO DETERMINE THE BOUNDARY??**

# Rocchio Classification (2/9)

---

**Rocchio classification** uses **centroid** to define the boundaries.

- Also called **prototype**.

The centroid of a class  $c$  is computed as the vector average (center of mass) of its members:

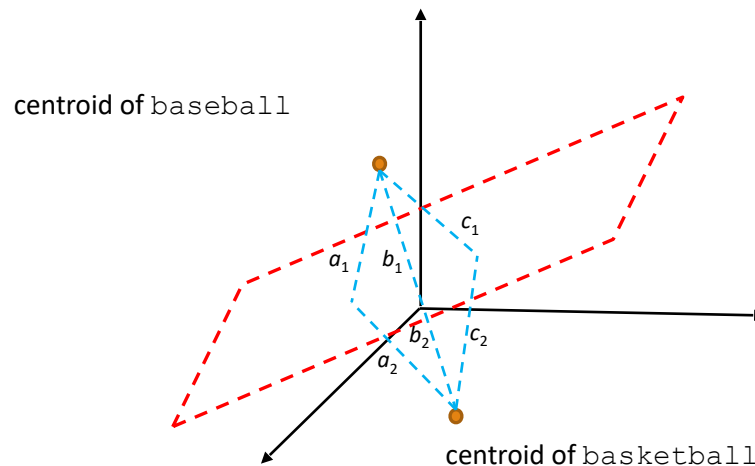
$$\underline{\mu}(c) = \frac{1}{|D_c|} \sum_{d \in D_c} \underline{v}(d)$$

- $D_c$  is the set of training documents whose class is  $c$ .
- $\underline{v}(d)$  is the vector of document  $d$ .

# Rocchio Classification (3/9)

Then, the boundary between two classes is the set of points with **equal distance** from the two centroids.

- $a_1=a_2$ ,  $b_1=b_2$ , and  $c_1=c_2$  in the figure.





# Rocchio Classification (4/9)

---

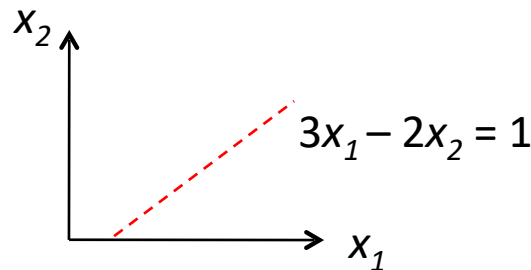
Decision Boundary:

- In 2-dimensional space, the set of points is always a **line**.
- In 3-dimensional space, the set of points is always a **plane**.
- In  $M$ -dimensional space, the set of points is a ***hyperplane***.

Lines, planes, and hyperplanes can be defined as:

$$\underline{w}^T \underline{x} = b \text{ or } \underline{w} \cdot \underline{x} = b$$

- For example, lines in 2-dimensional space can be represented as  $w_1x_1 + w_2x_2 = b$ .



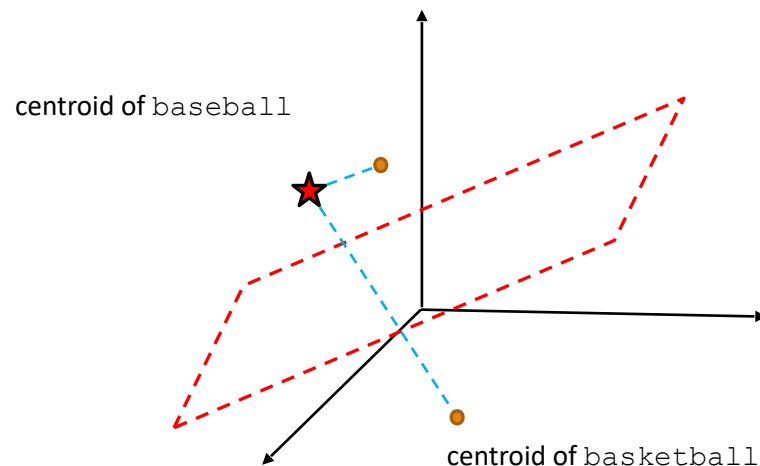
# Rocchio Classification (5/9)

---

In Rocchio classification, we do not have to identify the boundaries between classes.

Equivalently, to classify a point (document vector), we determine the centroid  $\mu(c)$  that the point is closest to and assign it to  $c$ .

- With ties broken randomly.



# Rocchio Classification (6/9)

---

Rocchio training:

```
TrainRocchio( $C, D$ )  
  for each  $c_j$  in  $C$   
    do  
       $D_j = \{d : \langle d, c_j \rangle \text{ in } D\}$   
       $\underline{u}_j = \Sigma_{d \text{ in } D_j} \underline{v}(d) / |D_j|$   
  return  $\{\underline{u}_1, \dots, \underline{u}_j\}$ 
```

Rocchio testing:

```
ApplyRocchio( $\{\underline{u}_1, \dots, \underline{u}_j\}, \underline{d}_{new}$ )  
  return  $\operatorname{argmin}_j |\underline{u}_j - \underline{d}_{new}|$ 
```

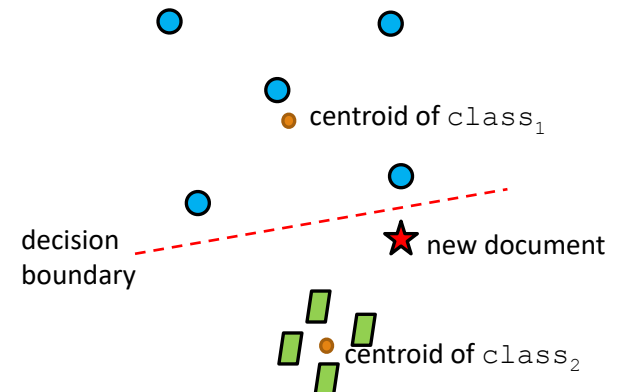
# Rocchio Classification (7/9)

Rocchio is so simple, but its classification results are generally inferior.

This is because the classification method **assumes** that the classes should be approximate spheres with similar radii.

Problems with similar radii:

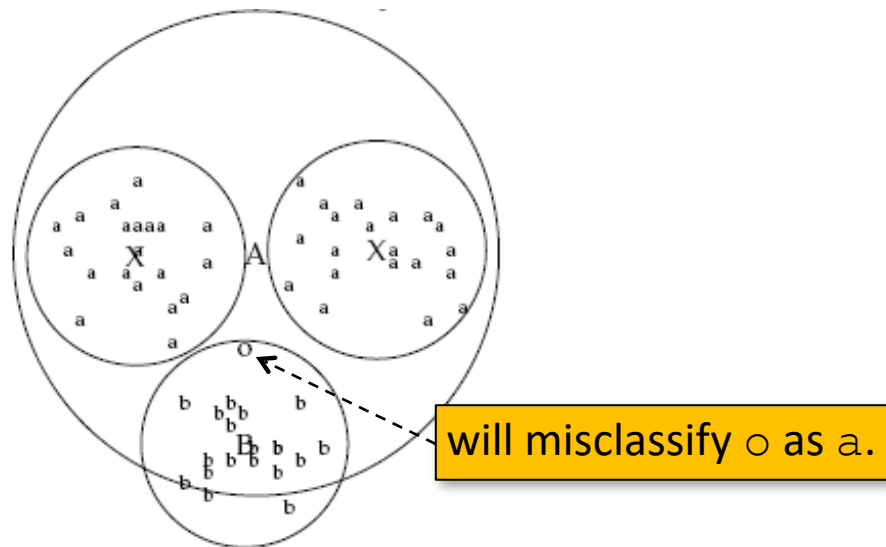
- The point is a better fit for `class1`.
  - `class1` is more scattered than `class2`.
- But Rocchio assigns it to `class2` because
  - It ignores the details of the distribution of point in a class .
  - And only uses distance from the centroid for classification.



# Rocchio Classification (8/9)

Problems with sphere:

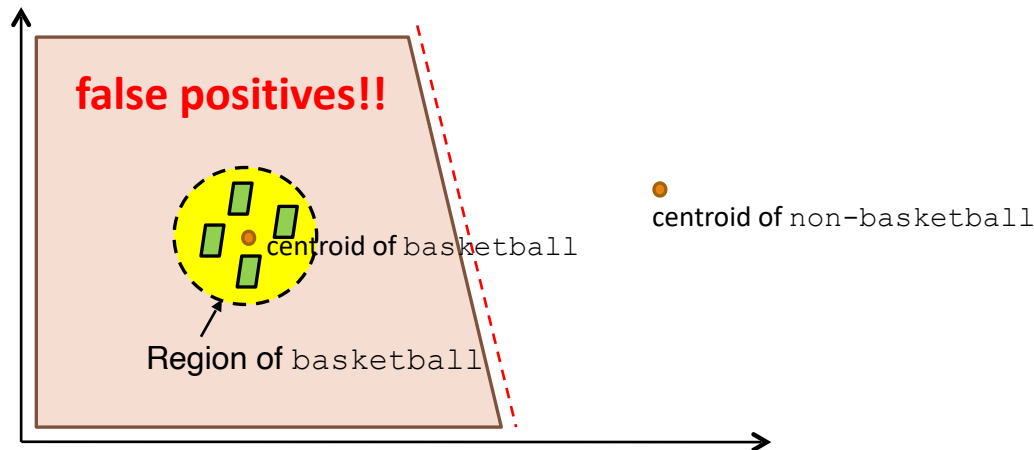
- Rocchio often misclassifies *multimodal class*.
- Multimodal class – a class has more than one cluster.



# Rocchio Classification (9/9)

Two-class classification are rarely distributed like spheres with similar radii.

- For example, class `basketball` occupies a small region of the space; class `not-basketball` are widely scattered.
- Assuming equal radii will result in a large number of false positives.



# $k$ Nearest Neighbor (1/6)

$kNN$  assigns documents to the majority class of their  $k$  closest neighbors, with ties broken randomly.

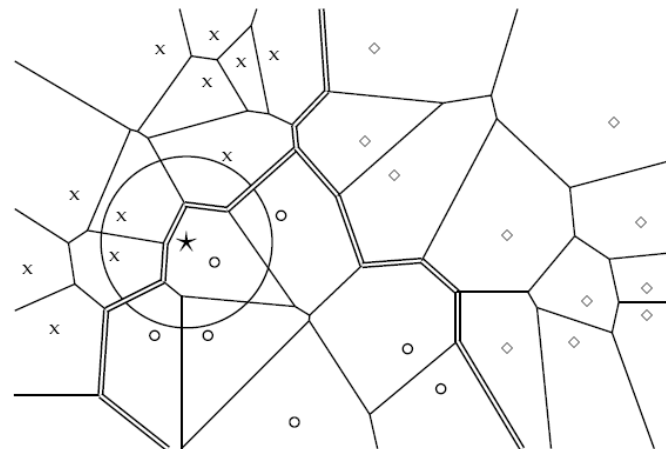
- So, it determines the decision boundary locally.

For 1NN, we assign each document to the class of its closest neighbor.

- Decision boundaries are **concatenated segments** of the *Voronoi tessellation*.

a set of objects (documents)  
decomposes space into cells.

each object's cell consists of  
all points that are closer to the  
object than other objects



each cell is a  
convex polygon.

# $k$ Nearest Neighbor (2/6)

---

Obviously, 1NN is not very robust, and kNN for  $k > 1$  is more robust.

- The parameter  $k$  is often chosen based on experience or knowledge about the classification problem at hand.
- It is desirable for  $k$  to be **odd** to make ties less likely.
- E.g.,  $k = 3$  and  $k = 5$  are common choices.
- Or to select the  $k$  that gives best results on a held-out portion of the training set (i.e., validation set).

## Probabilistic version of kNN:

- The probability  $P(c|d)$  is the proportion of the  $k$  nearest neighbors in the class.
- E.g., for  $k = 3$ ,  $P(\text{circle class}|\text{star}) = 1/3$ .



# $k$ Nearest Neighbor (3/6)

---

We can also **weight the “votes”** of the  $k$  nearest neighbors by their cosine similarity. Then assign the document to the class with the highest score.

$$\text{score}(c, d) = \sum_{d' \in S_k} I_c(d') \text{cosine}(d', d)$$

- $S_k$  is the set of  $d$ 's  $k$  nearest neighbors.
- $I_c(d') = 1$  if  $d'$  is in class  $c$  and 0 otherwise.

Weighting by similarities is often more accurate than simple voting.

- If two classes have the same number of votes, the class with the more similar neighbors wins.

# $k$ Nearest Neighbor (4/6)

---

kNN training:

```
Train-kNN(C, D)  
   $D' \leftarrow \text{Preprocess}(D)$   
   $k \leftarrow \text{Select-}k(C, D')$   
  return  $D', k$ 
```

- Different from most other classification algorithms, training a kNN classifier simply consists of determining  $k$  and preprocessing documents.
- $k$  is usually pre-defined; kNN requires **no training at all**.
- Preprocessing is necessary for every text mining system.

# *k* Nearest Neighbor (5/6)

---

- kNN testing:

```
Apply-kNN(C, D', k, d)  
   $S_k \leftarrow \text{ComputNearestNeighbors}(\underline{D'}, k, \underline{d})$   
  for each  $c_j$  in C  
  do  
     $p_j \leftarrow |S_k \cap c_j| / k$   
  return  $\text{argmax}_j p_j$ 
```

- **LESS EFFICIENT!!**
- Test time is **linear in the size of the training set**.
  - We need to compute the distance (or similarity) of each training document from the test document.
  - Many packages provide options to accelerate the nearest neighbor search

# $k$ Nearest Neighbor (6/6)

---

It is usually desirable to have as much training data as possible in learning tasks.

- However, large training sets come with a severe efficiency penalty in kNN.

kNN **does not perform** any estimation of parameters (e.g., centroids of Rocchio or priors and conditional probabilities of NB).

It simply **memorizes all training examples** and then compares the test document to them.

- We also call this type of learning algorithm *memory-based learning* or *instance-based learning*.

# Coding Example (1/4)

---

## Load data from the Excel file

```
In [1]: import xlrd

In [2]: workbook = xlrd.open_workbook('./blog-gender-dataset.xlsx')

In [3]: booksheet = workbook.sheet_by_name('data')

In [4]: texts = []
        labels = []

In [5]: for i in range(booksheet.nrows):
        labels.append(booksheet.cell(i,1).value)
        texts.append(booksheet.cell(i,0).value)
```

## Convert text into TFIDF vectors; split into training/testing sets

```
In [6]: from sklearn.feature_extraction.text import TfidfVectorizer

In [7]: TFIDF_vectorizer = TfidfVectorizer(min_df=1)

In [8]: TFIDF_vectors = TFIDF_vectorizer.fit_transform(texts)

In [9]: TFIDF_vectors.shape

Out[9]: (3227, 52456)

In [10]: x_train = TFIDF_vectors[0:2500]
        x_test = TFIDF_vectors[2500:]
        y_train = labels[0:2500]
        y_test = labels[2500:]
```

# Coding Example (2/4)

Construct a KNN model and fit the training data

```
In [11]: from sklearn.neighbors import KNeighborsClassifier
```

```
KNN_model = KNeighborsClassifier(n_neighbors = 5)
```

neighbor size

```
In [12]: KNN_model.fit(x_train,y_train)
```

```
Out[12]: KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',  
metric_params=None, n_jobs=None, n_neighbors=5, p=2,  
weights='uniform')
```

Set **weights='distance'** to  
weight neighbors according  
with their distances to the  
testing data

# Coding Example (3/4)

---

Make predictions on the testing data

```
In [13]: predicted_results = []  
         expected_results = []
```

```
In [14]: expected_results.extend(y_test)  
         predicted_results.extend(KNN_model.predict(x_test))  
         print(predicted_results)
```

```
['F', 'F', 'M', 'F', 'M', 'F', 'F', 'F', 'M', 'M', 'M', 'F', 'M', 'F', 'M', 'F', 'F', 'M',  
'M', 'M', 'M', 'M', 'F', 'M', 'F', 'F', 'M', 'M', 'F', 'M', 'M', 'F', 'F', 'M', 'M',  
'F', 'M', 'M', 'F', 'M', 'M', 'M', 'F', 'M', 'F', 'M', 'M', 'F', 'M', 'M', 'F', 'F', 'M',  
'F', 'F', 'F', 'F', 'M', 'M', 'F', 'M', 'M', 'F', 'M', 'M', 'F', 'M', 'F', 'M', 'F', 'M',  
'F', 'F', 'F', 'M', 'M', 'M', 'F', 'F', 'F', 'F', 'M', 'M', 'F', 'F', 'F', 'F', 'M', 'M',  
'F', 'M', 'M', 'M', 'F', 'F', 'F', 'M', 'M', 'F', 'M', 'M', 'F', 'M', 'M', 'M', 'M', 'F',  
'F', 'F', 'M', 'M', 'M', 'F', 'F', 'M', 'F', 'M', 'F', 'M', 'M', 'F', 'F', 'F', 'M', 'M',
```

# Coding Example (4/4)

---

Compute the classification score

```
In [15]: from sklearn import metrics  
print(metrics.classification_report(expected_results, predicted_results))
```

	precision	recall	f1-score	support
F	0.62	0.59	0.60	370
M	0.59	0.62	0.61	357
accuracy			0.60	727
macro avg	0.60	0.60	0.60	727
weighted avg	0.60	0.60	0.60	727

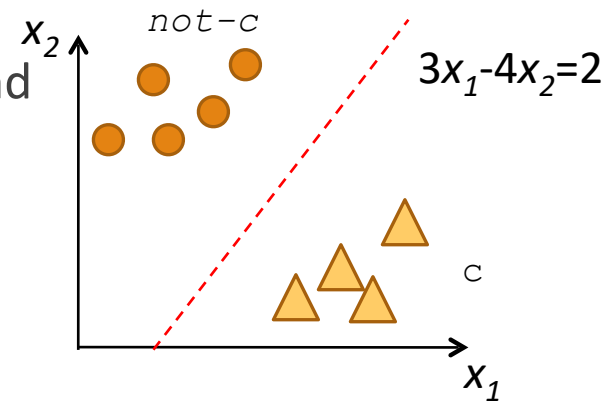


# Linear/Nonlinear Classifiers (1/6)

**Linear classifier** (two-class) decides class membership by comparing a linear combination of the features to a threshold.

In two dimensions, a linear classifier is a line!!

- The form of a line:  $w_1x_1 + w_2x_2 = b$ .
- The classifier assigns a document  $(x_1, x_2)$  to class  $c$  if  $w_1x_1 + w_2x_2 > b$  and to class  $not-c$  if  $w_1x_1 + w_2x_2 \leq b$ .

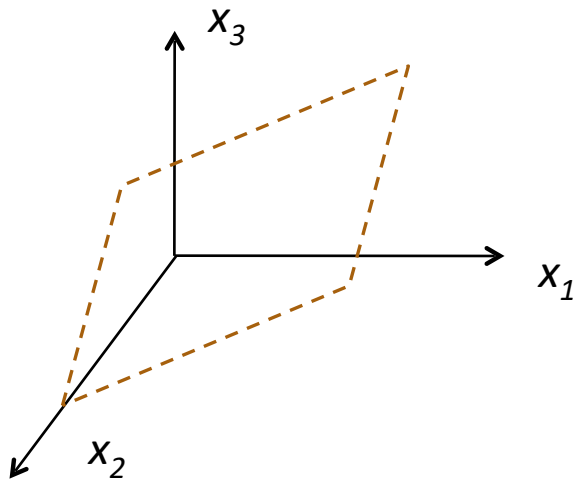


# Linear/Nonlinear Classifiers (2/6)

---

In  $M$ -dimensional space, a linear combination of the features, together with a threshold, define a hyperplane.

- E.g., in 3-dimensional space,  $w_1x_1 + w_2x_2 + w_3x_3 = b$  defines a plane.

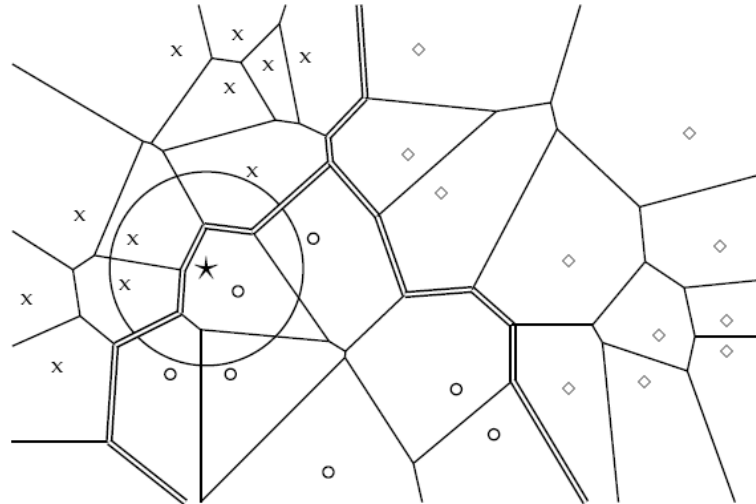


- We call the hyperplane (the linear combination) for classification decision a ***decision hyperplane***.
- $\underline{w}^T \underline{x} > b \rightarrow$  assign to class  $c$ , otherwise to  $not-c$ .

# Linear/Nonlinear Classifiers (3/6)

kNN linear or not?

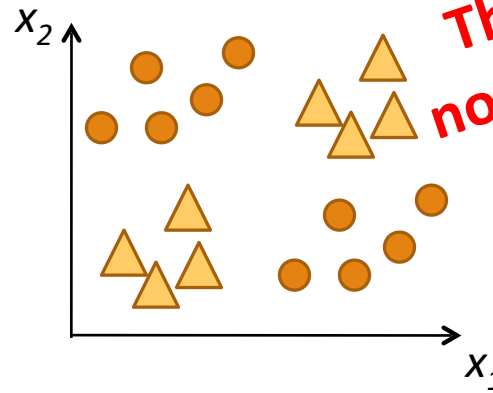
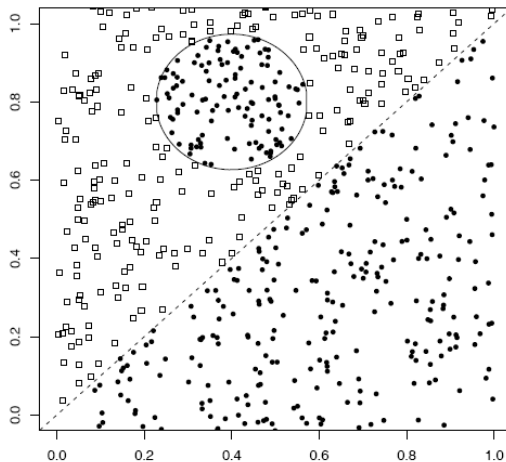
- KNN is **NOT** a linear classifier.
- The decision boundary of kNN consists of **locally linear segments**,
- In general the boundary has a complex shape that is not equivalent to a line in 2D or a hyperplane in  $M$ -dimensional space.



# Linear/Nonlinear Classifiers (4/6)

Linear/nonlinear -- **Which one is better?**

Let's consider the following cases:



**These two cases are not linearly separable!!**

We are **not** saying that non-linear methods are always superior to linear methods.

- **Bias-Variance trade-off:** no optimal learning method.
- In fact, many effective text classification methods are linear (e.g., SVM).

# Linear/Nonlinear Classifiers (5/6)

Let's assume the classification problem is linearly separable.

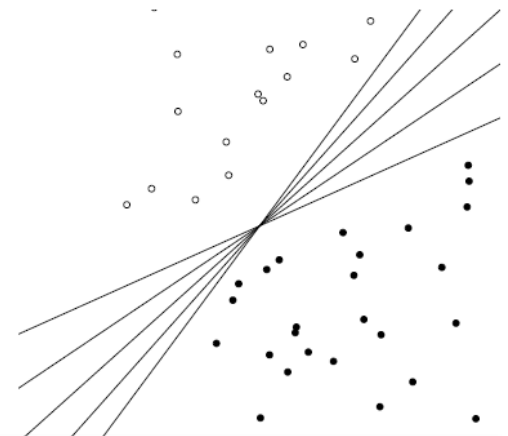
- Then, the training process of a linear classifier is to ...
  - Identify a separating hyperplane between the two classes.
  - Or ... to determine appropriate **normal vector  $\underline{w}$**  and  **$b$**  for separating the classes.



**So TRIVIAL, right?~~**

**NOT AT ALL!!**

Theoretically, there is an **infinite**  
**number of linear separators.**



**Which one is the best?**

# Linear/Nonlinear Classifiers (6/6)

---

Each linear classification method has its own way to identify appropriate separators.

- For instance, Rocchio determines the separators in terms of the set of points with **equal distance** from the centroids.

Some methods convert the  $(\underline{w}, b)$  searching as a optimization problem.

- **Support Vector Machines**

# Support Vector Machines (1/14)

---

林智仁

教授



譯自英文 - 林志仁 (Chih-Jen Lin) 是國立台灣大學計算機科學特聘教授，並且是機器學習，優化和數據挖掘方面的領先研究人員。他以開放源代碼庫LIBSVM (支持向量機的實現) 而聞名。

[維基百科 \(英文\)](#)

[查看原文說明](#) ▼

## LIBSVM: A library for support vector machines

CC Chang, [CJ Lin](#) - ACM transactions on intelligent systems and ..., 2011 - dl.acm.org

LIBSVM is a library for Support Vector Machines (SVMs). We have been actively developing this package since the year 2000. The goal is to help users to easily apply SVM to their applications. LIBSVM has gained wide popularity in machine learning and many other ...

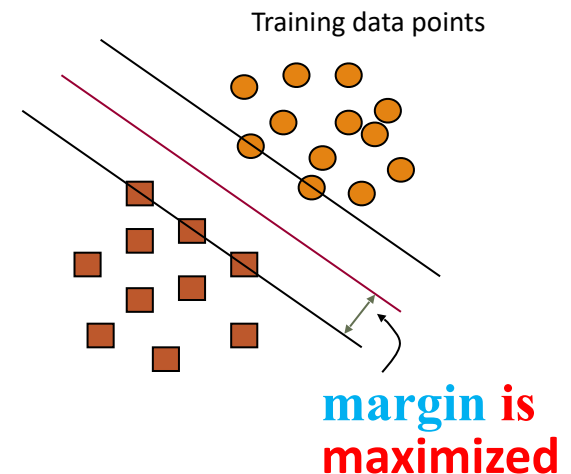
☆ 被引用 42782 次 相關文章 全部共 27 個版本

# Support Vector Machines (2/14)

**Support Vector Machine** (SVM) is a kind of *large-margin classifier*.

- *Margin*: the distance from a decision boundary to the closest data point.
- Its goal is to find a *decision boundary* (hyperplane) between two classes that is **maximally far from any point in the training data**.
- Points near the boundary represent very uncertain classification decision.
- A classifier (decision boundary) with a large margin makes little uncertain classification decisions.

Intuitively, a decision boundary drawn in the middle of the void between points of the two classes seems better.



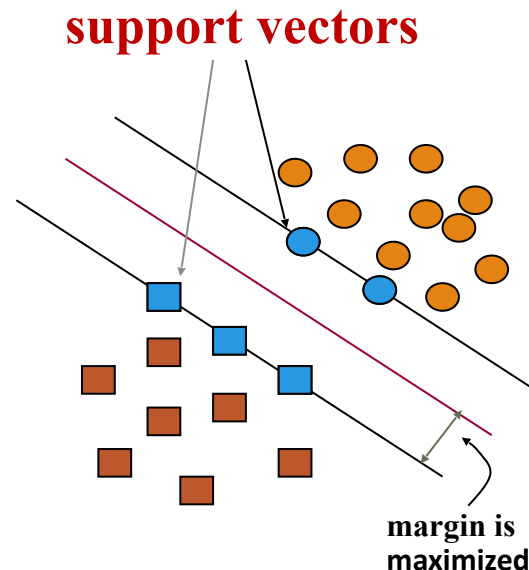


# Support Vector Machines (3/14)

The decision boundary is specified by **a subset of the data** which defines the position of the separator.

- these points are referred to as the **support vectors**.

**Other data points play no part in determining the decision boundary!!**



# Support Vector Machines (4/14)

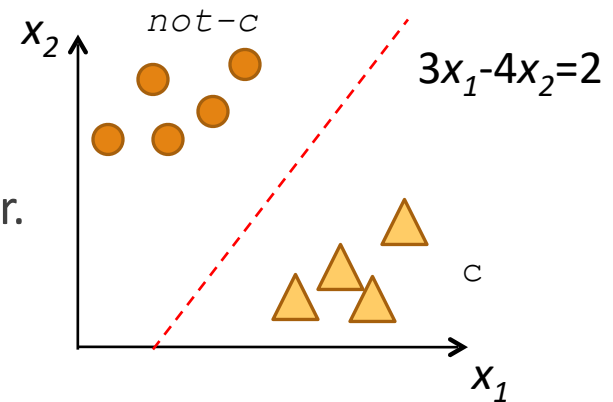
For SVM, the two data classes are named +1 and -1.

Mathematically, a decision hyperplane can be defined by  $\mathbf{b}$  and  $\mathbf{w}$ .  
Then, the (linear) classifier is:

$$f(\underline{x}) = \text{sign}(\underbrace{\underline{w}^T \underline{x}}_{\text{inner product of } \underline{w} \text{ and } \underline{x}} + b)$$

the signum function which returns **the sign of a number**

**Negative** indicates one class, **positive** the other.



# Support Vector Machines (5/14)

---

How do we acquire the decision boundary (i.e.,  $\underline{w}$  and  $b$ ) from training data set  $D = \{(\underline{x}_i, y_i)\}$  ?

- $\underline{x}_i$ : the feature vector of a training instance  $i$ ; e.g., a **tf-idf vector**
- $y_i$ : the corresponding class, +1 or -1.

For a decision hyperplane, we define the **functional margin** of the  $i$ th training instance as:

$$y_i(\underline{w}^T \underline{x}_i + b)$$

The functional margin of a training dataset with respect to a hyperplane is **twice** the functional margin of a point in the dataset with **minimal function margin**.

# Support Vector Machines (6/14)

---

Intuitively, we can acquire  $\underline{w}$  and  $b$  by maximizing the last **minimal function margin**

However ... the function margin **is unconstrained!!**

- We can always make the margin as big as we wish by scaling up  $\underline{w}$  and  $b$ .
  - For instance,  $\underline{w} \rightarrow 5\underline{w}$  and  $b \rightarrow 5$ , then the margin is five times as large.
- Impossible to obtain  $\langle \underline{w}, b \rangle$  to maximize the margin!!

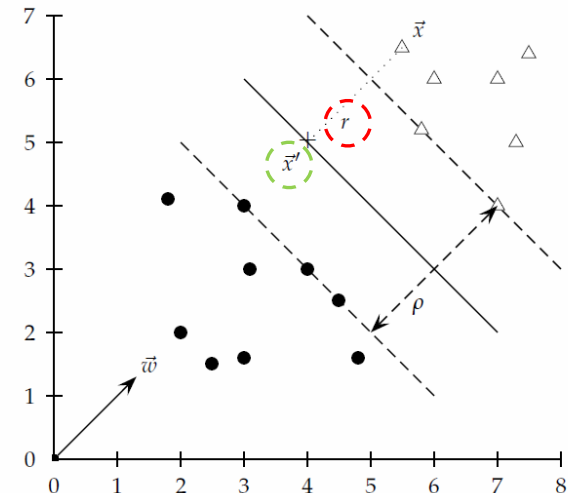
To place some constraint, let's look at the actual ***geometric distance*** (Euclidean distance).

# Support Vector Machines (7/14)

Let  $r$  be the distance we are looking for.

The shortest distance between a point and a hyperplane is perpendicular to the plane.

- That is, parallel to  $\underline{w}$ .
- A unit vector in this direction is  $\underline{w} / |\underline{w}|$ .



Then, the point on the hyperplane closest to  $\underline{x}$  (denoted as  $\underline{x}'$ )

$$\underline{x}' = \underline{x} - \rho \frac{\underline{w}}{|\underline{w}|}$$

# Support Vector Machines (8/14)

---

Since  $\underline{x}'$  lies on the decision boundary, it satisfies

$$\underline{w}^T \underline{x}' + b = 0$$

or

$$\begin{aligned} \underline{w}^T (\underline{x} - yr \frac{\underline{w}}{|\underline{w}|}) + b &= 0 \\ \underline{w}^T \underline{x} - yr \frac{\underline{w}^T \underline{w}}{|\underline{w}|} + b &= 0 \\ r &= y \frac{\underline{w}^T \underline{x} + b}{\left( \frac{\underline{w}^T \underline{w}}{|\underline{w}|} \right)} \end{aligned}$$

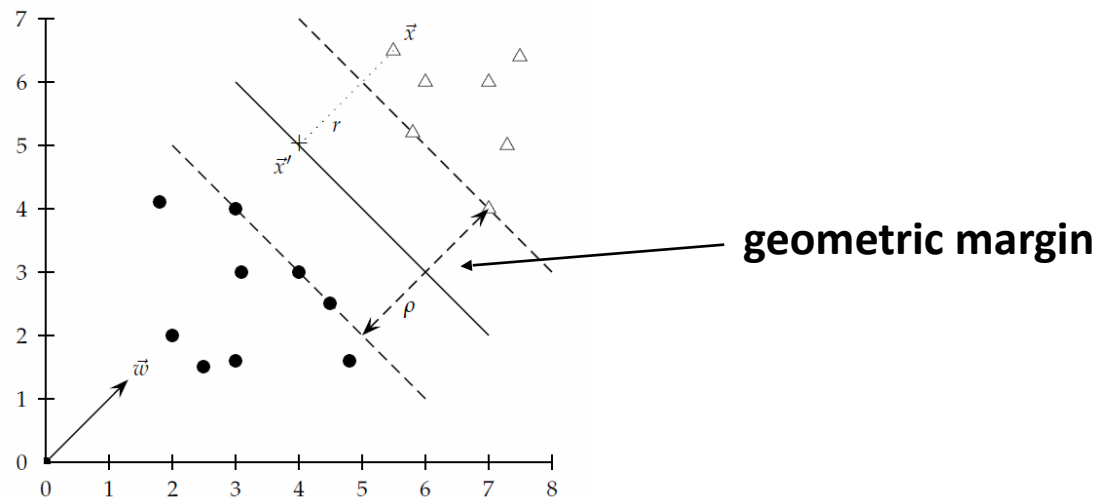
Since  $|\underline{w}| = (\underline{w}^T \underline{w})^{1/2}$

$$\begin{aligned} r &= y \frac{\underline{w}^T \underline{x} + b}{\left( \frac{\underline{w}^T \underline{w}}{(\underline{w}^T \underline{w})^{1/2}} \right)} \\ &= y \frac{\underline{w}^T \underline{x} + b}{(\underline{w}^T \underline{w})^{1/2}} \\ &= y \frac{\underline{w}^T \underline{x} + b}{|\underline{w}|} \end{aligned}$$

# Support Vector Machines (9/14)

Definition: the **geometric margin** of a classifier is the **maximum width** of the band that can be drawn separating the support vectors of the two classes.

- Or ... It is twice the minimum value over data points for  $r$ .



# Support Vector Machines (10/14)

---

The geometric margin is invariant to scaling of  $\langle \underline{w}, b \rangle$ !!

- If  $\underline{w} \rightarrow 5\underline{w}$  and  $b \rightarrow 5b$ , then the geometric margin is the same.

This means that ... when search for  $\langle \underline{w}, b \rangle$ , **we can impose any scaling constraint on  $\langle \underline{w}, b \rangle$  without affecting the geometric margin.**

- Or ... we can scale the function margin  $y_i(\underline{w}^T \underline{x}_i + b)$  without affecting the geometric margin.

**For convenience in solving SVMs**, we require that:

$$y_i(\underline{w}^T \underline{x}_i + b) \geq 1$$

- The functional margin of all training instances is at least 1.
- **Equal 1 for support vectors!!**



# Support Vector Machines (11/14)

---

Since each instance's distance from the hyperplane is

$$r_i = y_i \frac{\underline{w}^T \underline{x}_i + b}{|\underline{w}|}$$

and for support vectors  $y_i(\underline{w}^T \underline{x}_i + b) = 1$ ; The geometric margin thus is

$$\rho = 2/|\underline{w}|$$

The goal of SVMs is to maximize the geometric margin!!

- That is, we want to find  $\underline{w}$  and  $b$  such that:
  - $\rho = 2 / |\underline{w}|$  **is maximized!!**  $\longrightarrow$  or ...  $\frac{1}{2} \underline{w}^T \underline{w}$  **is minimized**
  - For all  $(x_i, y_i)$  in  $D$ ,  $y_i(\underline{w}^T \underline{x}_i + b) \geq 1$

# Support Vector Machines (12/14)

---

Not going to the detail of mathematics, but to better understand **kernel trick** and **kern function**, we show the dual form of the original optimization problem:

- Find  $\alpha_1, \dots, \alpha_N$  such that

$$\sum \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \underline{x}_i^T \underline{x}_j \text{ is maximized}$$

and

$$\sum \alpha_i y_i = 0$$

$$\alpha_i [y_i (\underline{w}^T \underline{x}_i + b) - 1] = 0$$

$$\alpha_i \geq 0 \text{ for all } 1 \leq i \leq N$$

The dual problem can be solved using numerical techniques (a topic beyond the scope of this course).

# Support Vector Machines (13/14)

---

Once the  $\alpha_i$  are found ...

$$\underline{w} = \sum \alpha_i y_i \underline{x}_i$$

- In the solution, most of the  $\alpha_i$  are zero.
- Each non-zero indicates that the corresponding  $\underline{x}_i$  is a support vector.

As  $y_i(\underline{w}^T \underline{x}_i + b) = 1$  for support vectors ...

$$b = y_k - \underline{w}^T \underline{x}_k \text{ for any } \underline{x}_k \text{ such that } \alpha_k \neq 0.$$

# Support Vector Machines (14/14)

Once we obtain  $\underline{w}$  and  $b$ , the classification function is:

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

Or

$$f(x) = \text{sign}\left(\sum_i \alpha_i y_i \underline{x}_i^T \underline{x} + b\right)$$

**WHY talk about this???**  
Because the **dot product** is the core of  
**kernel function**

This form involves **dot products** between  $\underline{x}$  (the instance to be classified) and  $\underline{x}_i$ 's (the training instances).

And ... **only support vectors** (the instances whose  $\alpha_i > 0$ ) **are involved in the classification decision!!**

# Soft Margin Classification (1/2)

Sometimes, the (training) data are not linearly separable!!

An approach to deal with the problem is to introduce **slack variables  $\zeta_i$** .

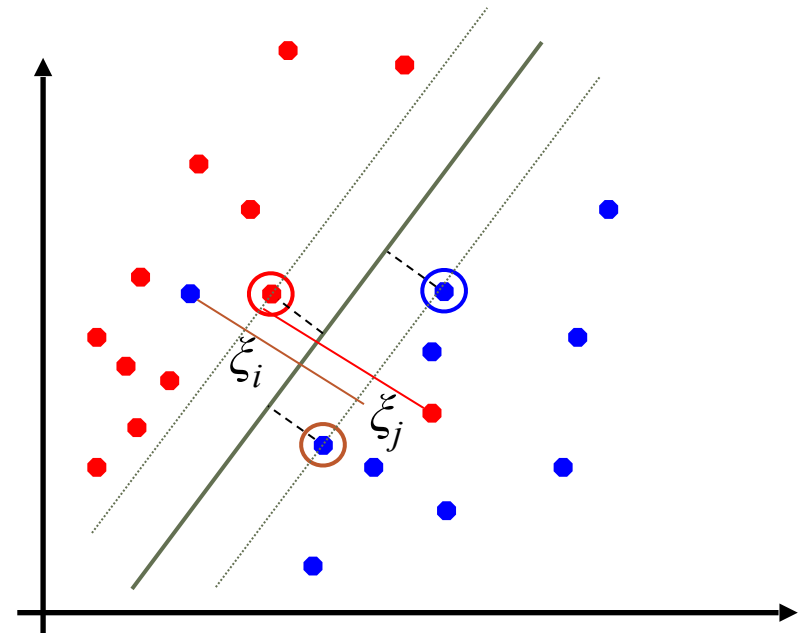
- To allow misclassification of difficult or noisy examples.

Then, find  $\underline{w}$  and  $b$  such that

$$\frac{1}{2} \underline{w}^T \underline{w} + C \sum_i \zeta_i \text{ is minimized}$$

$$\text{for all } \{(x_i, y_i)\}, y_i(\underline{w}^T \underline{x}_i + b) \geq 1 - \zeta_i$$

and  $\zeta_i \geq 0$



# Soft Margin Classification (2/2)

---

The margin can be less than 1 for a point  $x_i$  by having  $\zeta_i > 0$ !!

- But one pays a penalty of  $C\zeta_i$  in the minimization.

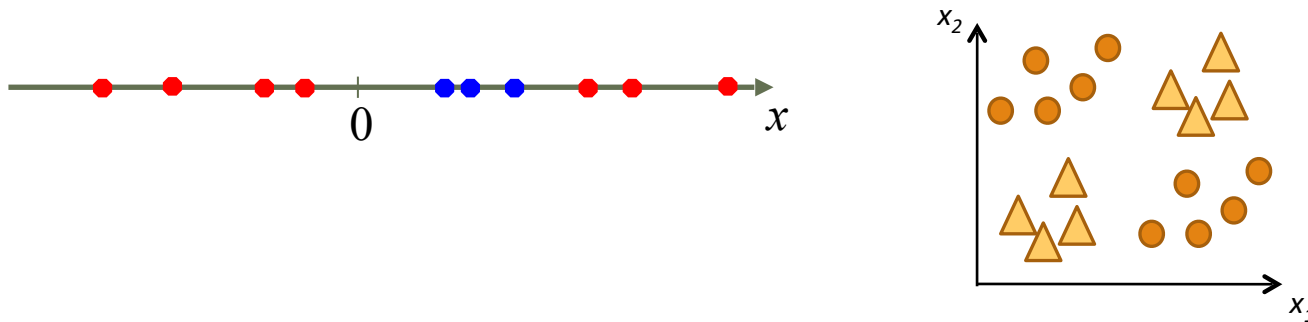
$C$  is a **regularization term**.

- A user-specified parameter.
  - Can be chosen based on a validation set.
- A large  $C$  penalizes learning errors more.
  - To decrease the margin.
- Conversely, a small  $C$  tends to relax the margin.

# Kernel SVM (1/7)

---

Sometimes, problem are too hard to linearly separate.

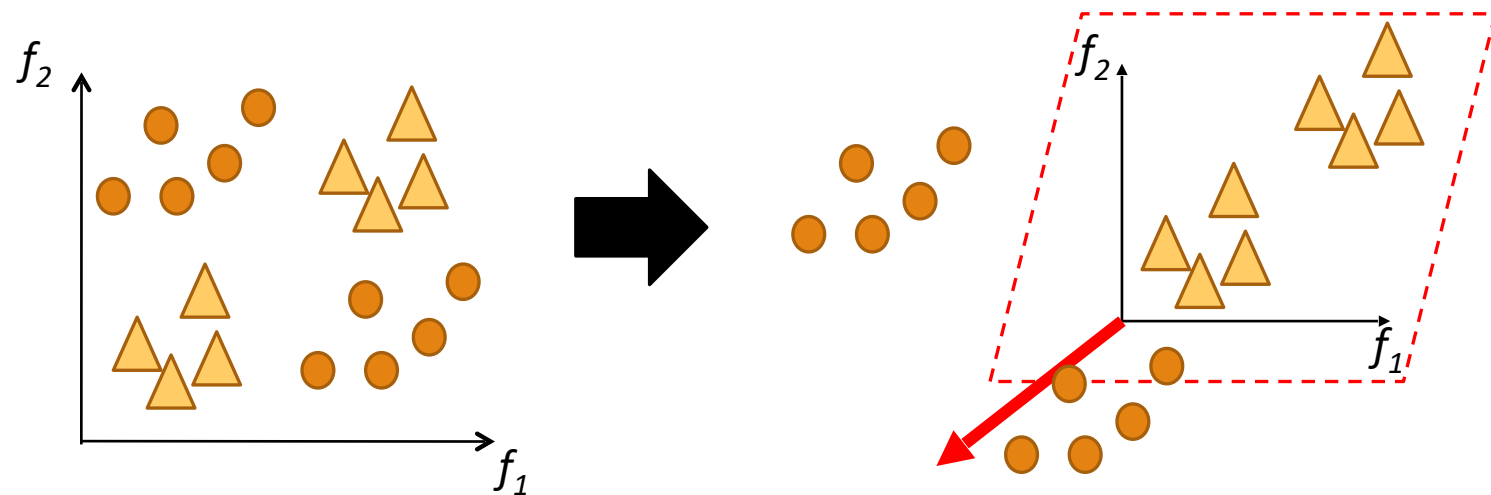


What if we can **map** the data on to a higher dimensional space.

- Then use a linear classifier in the higher dimensional space to separate the data.

# Kernel SVM (2/7)

---



**New dimension:** a certain kind of dimension related to  $f_1, f_2$ , or both.



# Kernel SVM (3/7)

Lets assume that there is a suitable **mapping function**  $\Phi$ .

- Then, we map every data point into a higher dimensional space via a transformation  $\Phi: \underline{x} \rightarrow \Phi(\underline{x})$ .
- And, the linear decision boundary in the transformed space has the following form:

$$\underline{w}^T \Phi(\underline{x}) + b = 0$$

Again, the learning task for a SVM can be formalized as follows:

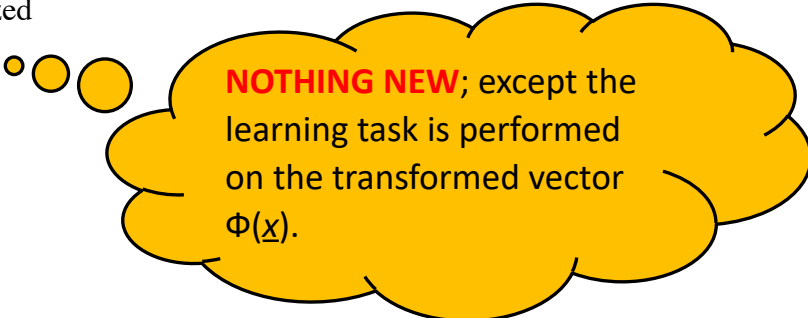
- Find  $\alpha_1, \dots, \alpha_N$  such that

$$\sum \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \Phi(\underline{x}_i)^T \Phi(\underline{x}_j) \text{ is maximized}$$

and

$$\sum \alpha_i y_i = 0$$

$$\alpha_i [y_i (\underline{w}^T \Phi(\underline{x}_i) + b) - 1] = 0$$
$$\alpha_i \geq 0 \text{ for all } 1 \leq i \leq N$$



**NOTHING NEW**; except the learning task is performed on the transformed vector  $\Phi(\underline{x})$ .

# Kernel SVM (4/7)

---

Similarly, once we obtain  $\alpha_1, \dots, \alpha_N$ , classification function is:

$$f(\underline{x}) = \text{sign}\left(\sum_i \alpha_i y_i \Phi(\underline{x}_i)^T \Phi(\underline{x}) + b\right)$$

The concept is neat and simple!! **BUT**

- **Mapping vectors into a higher dimension space and computing their dot product can be computationally expensive!!**

# Kernel SVM (5/7)

---

**Kernel functions** provide an easy and efficient way of doing the mapping and dot production, which is referred to as “**kernel trick**”.

- No need to transfer data into a high-dimensional space.
- The dot product (in high-dimensional space) is computed in terms of the original data points.

Let  $K(.)$  be a kernel function that  $K(\underline{x}_i, \underline{x}_j) = \Phi(\underline{x}_i)^T \Phi(\underline{x}_j)$ , then

$$\begin{aligned} f(\underline{x}) &= \text{sign}\left(\sum_i \alpha_i y_i \Phi(\underline{x}_i)^T \Phi(\underline{x}) + b\right) \\ &= \text{sign}\left(\sum_i \alpha_i y_i K(\underline{x}_i, \underline{x}) + b\right) \end{aligned}$$

# Kernel SVM (6/7)

---

Example:

- 2-dimensional vectors  $\underline{u} = (u_1 \ u_2)$ ,  $\underline{v} = (v_1 \ v_2)$ , let  $K(\underline{u}, \underline{v}) = (1 + \underline{u}^T \underline{v})^2$ .
- We show that  $K$  is a kernel function, i.e., that  $K(\underline{u}, \underline{v}) = \Phi(\underline{u})^T \Phi(\underline{v})$  for that mapping function

$$\Phi(\underline{u}) = (1 \ u_1^2 \ \sqrt{2}u_1u_2 \ u_2^2 \ \sqrt{2}u_1 \ \sqrt{2}u_2)$$

$$\begin{aligned} K(\underline{u}, \underline{v}) &= (1 + \underline{u}^T \underline{v})^2 \\ &= 1 + 2\underline{u}^T \underline{v} + (\underline{u}^T \underline{v})^2 \\ &= 1 + 2(u_1v_1 + u_2v_2) + (u_1v_1 + u_2v_2)^2 \\ &= 1 + u_1^2v_1^2 + 2u_1v_1u_2v_2 + u_2^2v_2^2 + 2u_1v_1 + 2u_2v_2 \\ &= (1 \ u_1^2 \ \sqrt{2}u_1u_2 \ u_2^2 \ \sqrt{2}u_1 \ \sqrt{2}u_2)^T (1 \ v_1^2 \ \sqrt{2}v_1v_2 \ v_2^2 \ \sqrt{2}v_1 \ \sqrt{2}v_2) \\ &= \Phi(\underline{u})^T \Phi(\underline{v}) \end{aligned}$$

# Kernel SVM (7/7)

---

Two well-known kernel functions: **polynomial kernels** and **radial basis functions** (RBF).

**Polynomial kernels** are of the form  $K(\underline{u}, \underline{v}) = (\gamma \underline{u}^T \underline{v} + \text{coef})^d$ .

- E.g.,  $\gamma = 1$ ,  $\text{coef} = 1$ , and  $d = 2$

**Radial basis function** is a Gaussian distribution:

$$K(\underline{u}, \underline{v}) = e^{-\gamma |\underline{u} - \underline{v}|^2}$$

- It is equivalent to mapping the data into an **infinite** dimensional space.
- Default kernel of `sklearn.svm.SVC`

# Coding Example

---

Cut to the chase: SVM model construction

- `sklearn.svm.SVC` is based on **libsvm**!!

```
In [11]: from sklearn.svm import SVC
```

```
#SVM_model = SVC(kernel='linear', C=1.0)  
#SVM_model = SVC(kernel='rbf', gamma='scale', C=1.0)  
SVM_model = SVC(kernel='poly', degree=2, coef0=1, C=1.0)
```

```
In [12]: SVM_model.fit(x_train,y_train)
```

```
Out[12]: SVC(C=1.0, break_ties=False, cache_size=200, class_weight=None, coef0=1,  
            decision_function_shape='ovr', degree=2, gamma='scale', kernel='poly',  
            max_iter=-1, probability=False, random_state=None, shrinking=True,  
            tol=0.001, verbose=False)
```