# Machine Learning

## CS 165B

Prof. Matthew Turk

Wednesday, May 11, 2016

Today

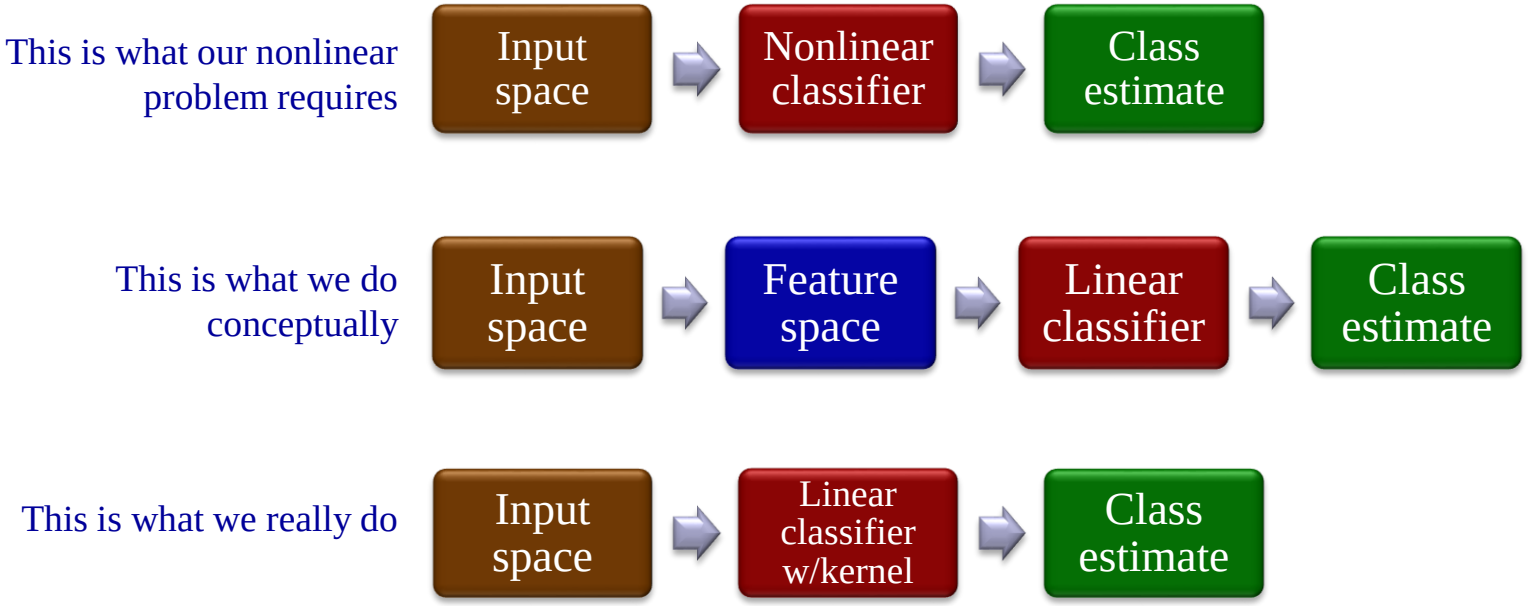
- Nonlinear kernel classifiers
- Distance metrics and clustering (Ch. 8)

# Notes

- HW#4 will be posted on Friday, due next Friday (May 19)

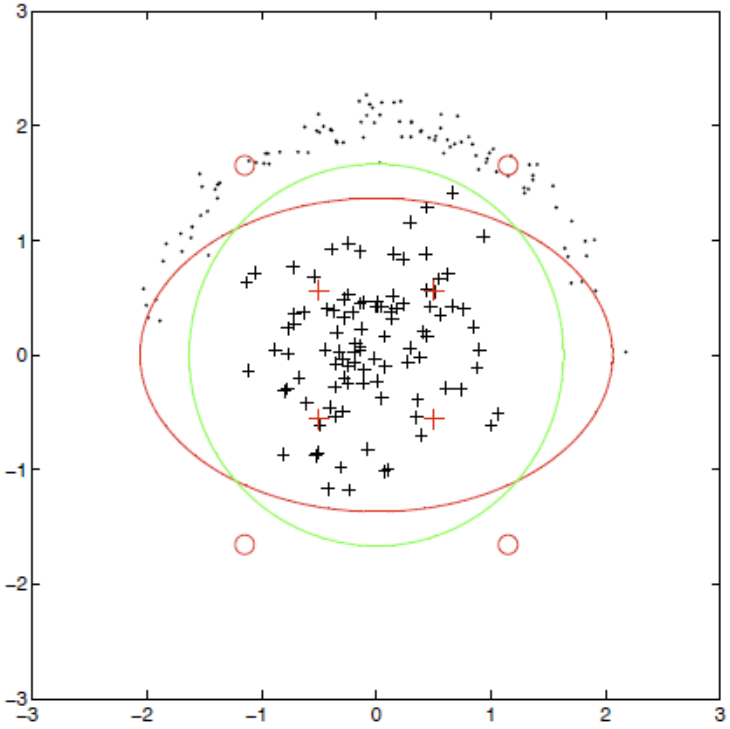
# Nonlinear kernel classifiers

- In many problems, linear decision boundaries just won't do the job
- We can adapt our linear methods to learn (some) nonlinear decision boundaries by transforming the data nonlinearly to a feature space in which is suited for linear classification
  - These are kernel methods – the ***kernel trick***!

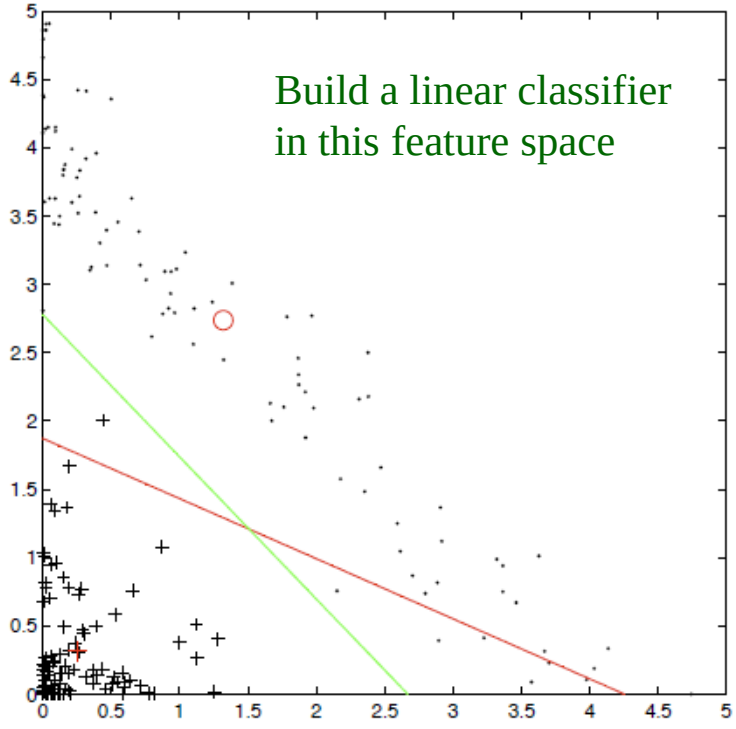


# Example: a quadratic decision boundary

**Input space**



**Feature space** – square the input features

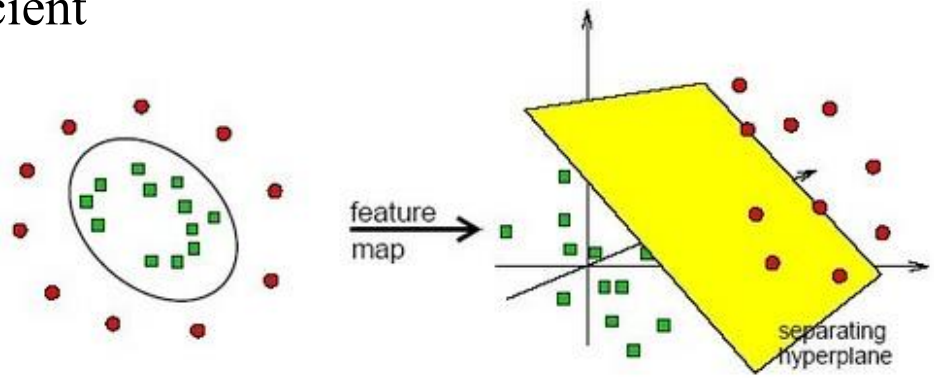



Red – Basic linear classifier decision boundary (centroids = red circles/crosses)
Green – Perceptron decision boundary

Often in kernel methods we don't actually construct the feature space – rather, we perform all operations in the input space

# The kernel trick

- In machine learning, the "kernel trick" is a way of mapping features into another (often higher dimensional) space to make the data linearly separable, *without having to compute the mapping explicitly*.
- The dot product operation in a linear classifier $\boldsymbol{x}_1 \cdot \boldsymbol{x}_2$ is replaced by a kernel function $\kappa(\boldsymbol{x}_1, \boldsymbol{x}_2)$ that computes the dot product of the values $(\boldsymbol{x}_1', \boldsymbol{x}_2')$ in the new (linearly separable) space.
  - Again, without having to compute the mapping from $(\boldsymbol{x}_1, \boldsymbol{x}_2)$ to $(\boldsymbol{x}_1', \boldsymbol{x}_2')$
  - So it's both effective and efficient
- Let's see an example….



# The kernel trick

- In the original feature space, the two classes (o's and x's) are not linearly separable
- So let's map $\boldsymbol{p} = (x_1, x_2)$ to a new space $\boldsymbol{q} = (z_1, z_2, z_3)$ via the transformation $\varphi(\boldsymbol{p})$:

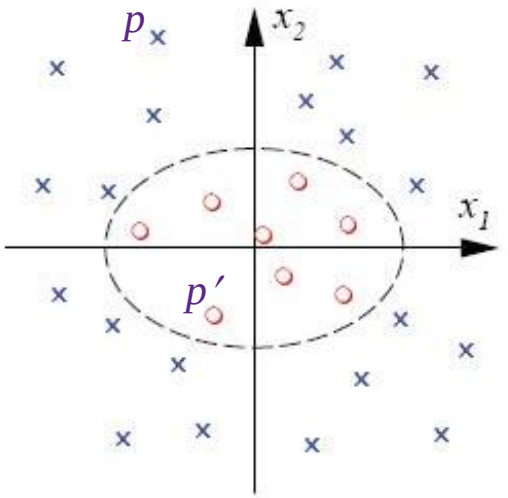
$$z_1 = x_1^2$$
$$z_2 = x_2^2$$
$$z_3 = \sqrt{2}x_1x_2$$

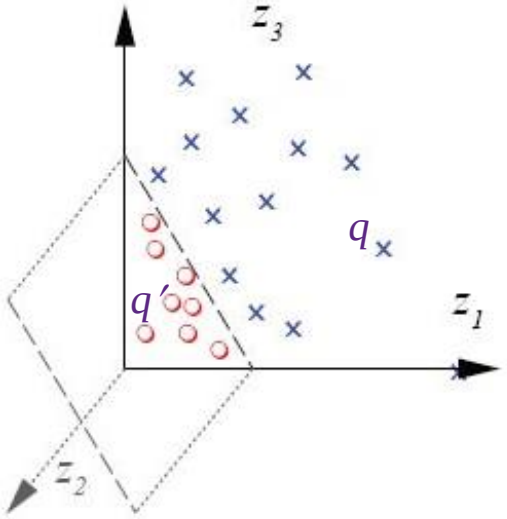
where, it turns out, the o's and x's are linearly separable.

- A dot product in the new space:

$$\begin{aligned}\boldsymbol{q} \cdot \boldsymbol{q}' &= z_1 z_1' + z_2 z_2' + z_3 z_3' \\ &= x_1^2 x_1'^2 + x_2^2 x_2'^2 + \sqrt{2}x_1x_2\sqrt{2}\,x_1' x_2' \\ &= (x_1 x_1' + x_2 x_2')^2 \\ &= (\boldsymbol{p}\cdot\boldsymbol{p}')^2 = \kappa(\boldsymbol{p}_1, \boldsymbol{p}_2)\end{aligned}$$

is merely the square of the original dot product!



# Feature transformation and the kernel trick

- The kernel trick is widely used in machine learning
- Assumption: achieving linear separation is worth the effort
  - There are non-linear classifiers, but linear classification tends to be simple and fast
- Assumption: the dot product is the key computation
  - Yes, for a linear classifier
  - So we just replace the dot product with the kernel function
- How do we find the mapping that will make the data linearly separable?
  - Good question!
  - Insight into the data, trial and error, …
  - Are there principled ways to determine such a transformation?

# The kernel function

- We have linear methods that use the dot product among instances, $\boldsymbol{x}_1{}^T \boldsymbol{x}_2$ (also written $\boldsymbol{x}_1 \cdot \boldsymbol{x}_2$)
  - But if our data is not appropriate for a linear model, we can't use these methods!
- So… we find a transformation $\varphi(\boldsymbol{x})$ of the input space into a feature space that makes the data linearly separable
- Then, for training and subsequent classification, we conceptually transform inputs $\boldsymbol{x}$ into the feature space $\varphi(\boldsymbol{x})$ to learn a linear classifier and for classifying new instances
- But we don't actually have to do this. Instead, we define a kernel function $\kappa(\boldsymbol{x}_1, \boldsymbol{x}_2)$ that performs the dot product in the feature space – i.e., $\kappa(\boldsymbol{x}_1, \boldsymbol{x}_2) = \varphi(\boldsymbol{x}_1) \cdot \varphi(\boldsymbol{x}_2)$
  - $\kappa : \mathbb{R}^N \times \mathbb{R}^N \rightarrow \mathbb{R}$

# Kernel perceptron

Learns a nonlinear decision boundary

**Algorithm** KernelPerceptron($D, \kappa$) – perceptron training algorithm using a kernel.

**Input** : labelled training data $D$ in homogeneous coordinates;
kernel function $\kappa$.

**Output** : coefficients $\alpha_i$ defining non-linear decision boundary.

$\alpha_i \leftarrow 0$ for $1 \leq i \leq |D|$;
*converged*←false;
**while** *converged* = false **do**
    *converged*←true;
    **for** $i = 1$ to $|D|$ **do**
        **if** $y_i \sum_{j=1}^{|D|} \alpha_j y_j \kappa(\mathbf{x}_i, \mathbf{x}_j) \leq 0$ **then**
            $\alpha_i \leftarrow \alpha_i + 1$;
            *converged*←false;
        **end**
    **end**
**end**

replaces $\boldsymbol{x}_i \cdot \boldsymbol{x}_j$

# Kernel perceptron

- The kernel perceptron doesn't learn a linear discriminant $\boldsymbol{w}$
  - It learns the $\alpha_i$ parameters (see the dual form of the learning algorithm)
- Classifying a new instance does not use $\boldsymbol{w}^T \boldsymbol{x} > t$ – instead, it evaluates

$$\sum_{i=1}^{n} \alpha_i y_i \, \kappa(\boldsymbol{x}, \boldsymbol{x}_i) > t$$

- This is $O(n)$, involving all training data with non-zero $\alpha_i$
- This approach will be more efficient with SVMs, since $\alpha_i \neq 0$ only for the support vectors!

# Kernel SVM

- The kernel SVM is the same basic idea as the kernel perceptron – replace the dot product $\boldsymbol{x}_i^T \boldsymbol{x}_j$ with a kernel function $\kappa(\boldsymbol{x}_i, \boldsymbol{x}_j)$ that captures the nonlinear mapping of the input space to the feature space, where the data are linearly separable

- We can then replace the Gram matrix **G** with the kernel matrix **K**, and use entries of **K** in the learning computation

$$\alpha_1^*, \ldots, \alpha_n^* = \underset{\alpha_1, \ldots, \alpha_n}{\arg\max} -\frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \kappa(\mathbf{x}_i, \mathbf{x}_j) + \sum_{i=1}^{n} \alpha_i$$

$$\text{subject to } \alpha_i \geq 0, 1 \leq i \leq n \text{ and } \sum_{i=1}^{n} \alpha_i y_i = 0$$

But only entries for which $\alpha_i > 0$

# Kernel SVM

- After learning the $\alpha_i$ parameters, we can then classify a new instance $\boldsymbol{x}$ using

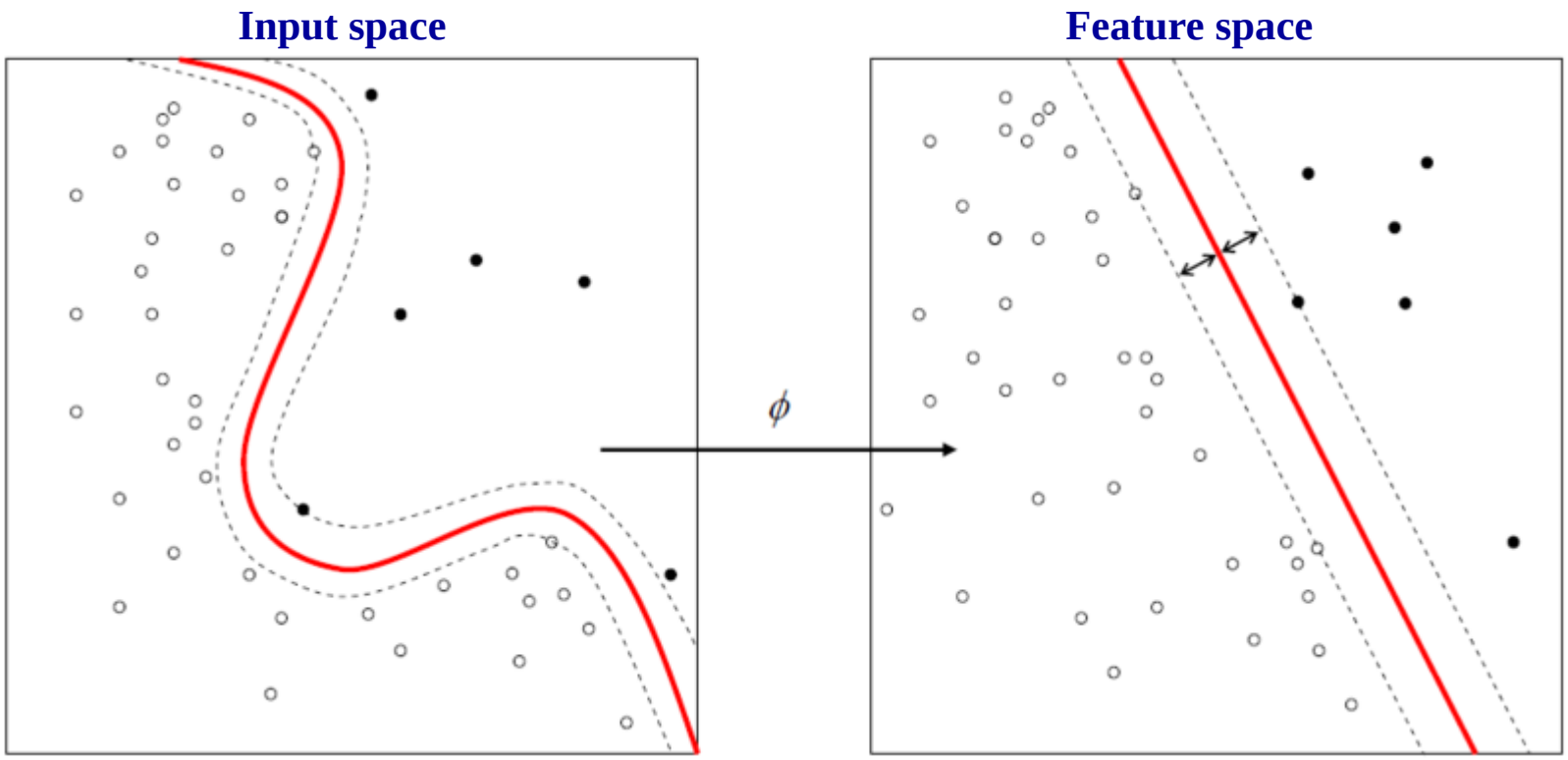
$$\sum_{i=1}^{n} \alpha_i y_i \, \kappa(\boldsymbol{x}, \boldsymbol{x}_i) > t$$

- This sum is only over the support vectors, so it's an efficient computation

- To learn a soft margin kernel SVM, we can include slack variables $\xi_i$ and the complexity parameter $C$

# Kernel SVM

With kernel functions, SVMs can be used as non-linear classifiers



# Some kernel functions

- The linear kernel:

$$\kappa(\boldsymbol{x}_1, \boldsymbol{x}_2) = \boldsymbol{x}_1{}^T \boldsymbol{x}_2$$

- The polynomial kernel:

$$\kappa(\boldsymbol{x}_1, \boldsymbol{x}_2) = (\boldsymbol{x}_1{}^T \boldsymbol{x}_2 + c)^d$$

- The Gaussian kernel

$$\kappa(\boldsymbol{x}_1, \boldsymbol{x}_2) = \exp\left(\frac{-\|\boldsymbol{x}_1 - \boldsymbol{x}_2\|^2}{2\sigma^2}\right)$$

This is also known as a radial basis function (RBF) kernel

It is essentially a measure of similarity between $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$, scaled by $\sigma$

- The larger $\sigma$ is, the more effect a distant point $\boldsymbol{x}_i$ will have

# How to choose a kernel function

- Selecting a kernel function entails:
  - Choosing the function family (polynomial, RBF, etc.)
  - Determining the parameters of the function
    - ($c$, $d$) for polynomial
    - $\sigma$ for RBF
    - Etc.
- Various optimization methods exist for making these choices, using cross-validation (randomly partitioning the experimental data into training and validation parts, repeatedly)
  - *Applying machine learning to machine learning!*
- Knowledge of the problem space can be helpful
  - Collected wisdom: "In cases like *this*, try *that* kernel function…"

# Distance metrics and clustering

Chapter 8 in the textbook

# Distance and clustering

- In many machine learning methods – especially geometric models – the notion of distance is important
- Especially in clustering, where we assume similarity is some function of distance
  - But using what distance measure?
- Clustering is grouping data without prior information (unlabeled data)
- Why cluster?
  - To make apparent the natural groupings/structure in the data (perhaps for further processing)
  - To discover previously unknown relationships
  - To provide generic labels for the data

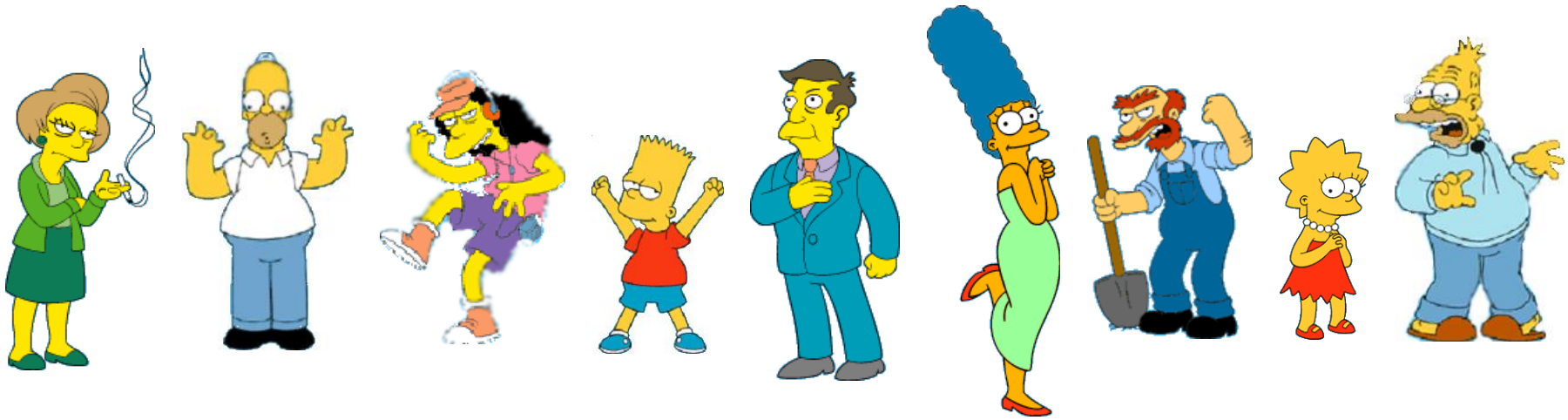
# Clustering

- In clustering, we organize data into classes such that:
  - The within-class (intra-class) similarity is high
    - Lower intra-class variance
  - The between-class (inter-class) similarity is low
    - Higher inter-class variance
  - Objects in the same group (a cluster) are more similar to one another than to objects in other groups (clusters)

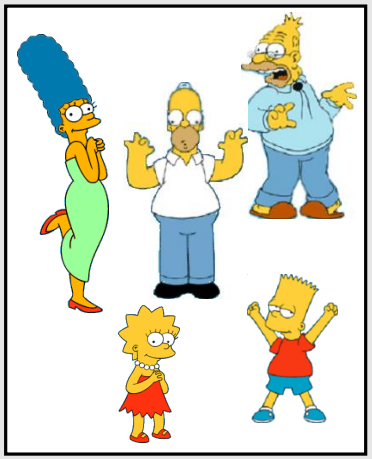
- But similarity and grouping may not be obvious…
- We'd like to define features and distance measures that will capture the intended notion of similarity

Distance $\propto$ dissimilarity

# What's a natural grouping among these objects?



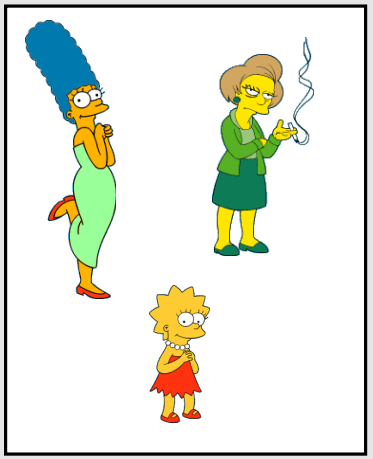
Clustering is subjective!



Simpson Family

School Employees

Others?



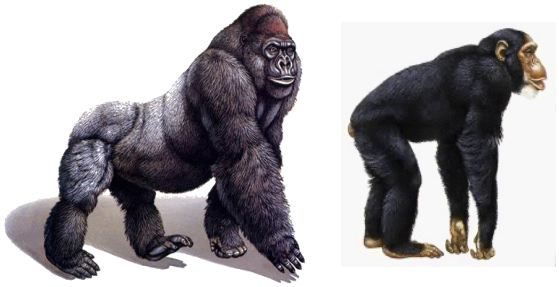
Females

Males

# What is similarity?

# Distance measures

Let $O_1$ and $O_2$ be two objects from the universe of possible objects. The distance (dissimilarity) between $O_1$ and $O_2$ is a real number denoted by $D(O_1,O_2)$





$D(O_1, O_2)$

0.6

Peter Piotr

$D(O_1, O_2)$

3.0

$D(O_1, O_2)$

342.7

# Distance measures

A distance metric $D(x_1, x_2)$ is a function $D : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ such that for any $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z} \in \mathcal{X}$:

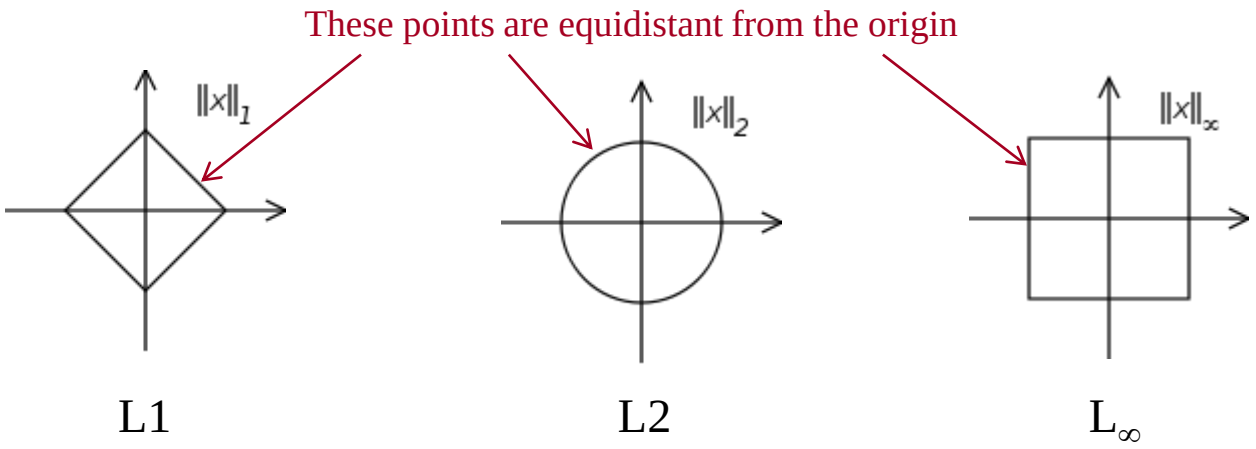
1. $D(\boldsymbol{x}, \boldsymbol{x}) = 0$
2. If $\boldsymbol{x} \neq \boldsymbol{y}$ then $D(\boldsymbol{x}, \boldsymbol{y}) > 0$
3. $D(\boldsymbol{x}, \boldsymbol{y}) = D(\boldsymbol{y}, \boldsymbol{x})$
4. $D(\boldsymbol{x}, \boldsymbol{z}) \leq D(\boldsymbol{x}, \boldsymbol{y}) + D(\boldsymbol{y}, \boldsymbol{z})$

Or we can refer to a norm $D(v)$ of $\boldsymbol{v} = \boldsymbol{x} - \boldsymbol{y}$, such that $D : \mathcal{X} \to \mathbb{R}$ :

1. $D(\mathbf{0}) = 0$
2. If $\boldsymbol{v} \neq \mathbf{0}$ then $D(\boldsymbol{v}) > 0$
3. $D(\boldsymbol{v}) = D(-\boldsymbol{v})$
4. $D(\boldsymbol{a} + \boldsymbol{b}) \leq D(\boldsymbol{a}) + D(\boldsymbol{b})$

# Some common distance measures

- Manhattan (L1) distance: $D(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=1}^{d} |x_i - y_i| = \|\boldsymbol{x} - \boldsymbol{y}\|_1$
  *1-norm, Cityblock/Manhattan distance*
- Euclidian (L2) distance: $D(\boldsymbol{x}, \boldsymbol{y}) = \left( \sum_{i=1}^{d} (x_i - y_i)^2 \right)^{1/2} = \|\boldsymbol{x} - \boldsymbol{y}\|_2$
  *2-norm*
- Minkowski ($L_p$) distance: $D(\boldsymbol{x}, \boldsymbol{y}) = \left( \sum_{i=1}^{d} |x_i - y_i|^p \right)^{1/p} = \|\boldsymbol{x} - \boldsymbol{y}\|_p$
  *p-norm*



# Some common distance metrics (cont.)

- $L_\infty$ distance/norm is known as Chebyshev distance

$$L_\infty(\boldsymbol{x}, \boldsymbol{y}) = \|\boldsymbol{x} - \boldsymbol{y}\|_\infty = \max_i |x_i - y_i|$$

- $L_0$ distance/norm counts the number of non-zero elements

$$L_0(\boldsymbol{x}, \boldsymbol{y}) = \|\boldsymbol{x} - \boldsymbol{y}\|_0 = \text{count}(|x_i - y_i| > 0)$$

  - This is the Hamming distance if $\boldsymbol{x}$ and $\boldsymbol{y}$ are binary vectors

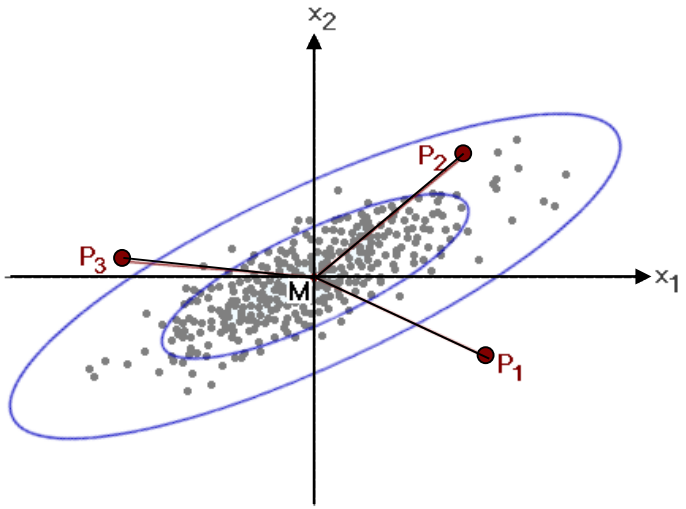
- Mahalanobis distance takes into account the covariance in a data set

$$D_M(\boldsymbol{x}, \boldsymbol{y}) = \sqrt{(\boldsymbol{x} - \boldsymbol{y})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{x} - \boldsymbol{y})}$$

where $\boldsymbol{\Sigma}$ is the covariance matrix

$$\boldsymbol{\Sigma} = \frac{1}{k} \boldsymbol{X}\boldsymbol{X}^T = \frac{1}{k} \boldsymbol{S}$$

Scatter matrix



# Distance-based methods

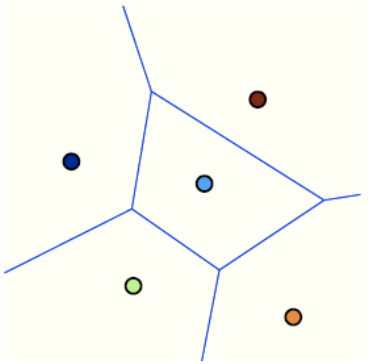
- Methods for classification and clustering based on distances to exemplars or neighbors
  - Exemplar – a prototypical instance
    - E.g., the ideal example instance of Class A
  - Neighbor – a "nearby" instance or exemplar
    - E.g., within some distance radius *d*

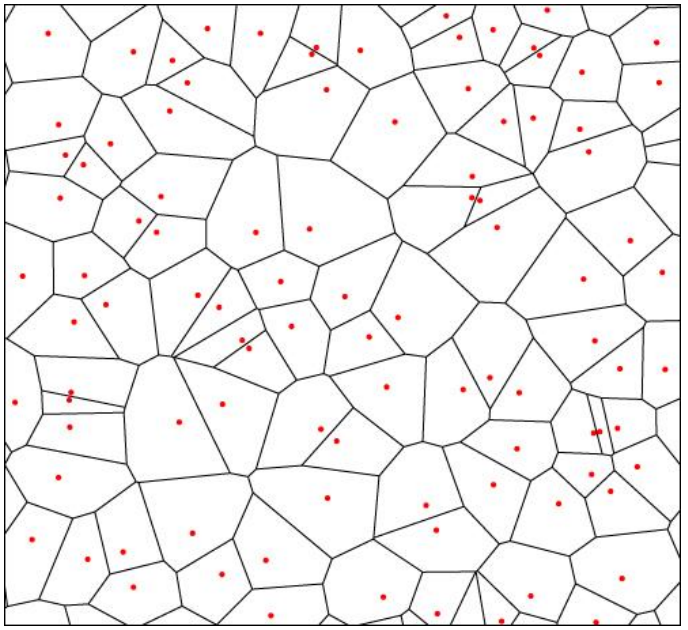
- Our basic (binary) linear classifier follows this procedure:
  1. Construct an exemplar for each class from its mean
  2. Assign a new instance ***x*** to the nearest exemplar using Euclidian distance

- This is a basic nearest neighbor (NN) approach
  - No explicit construction of a decision boundary is required

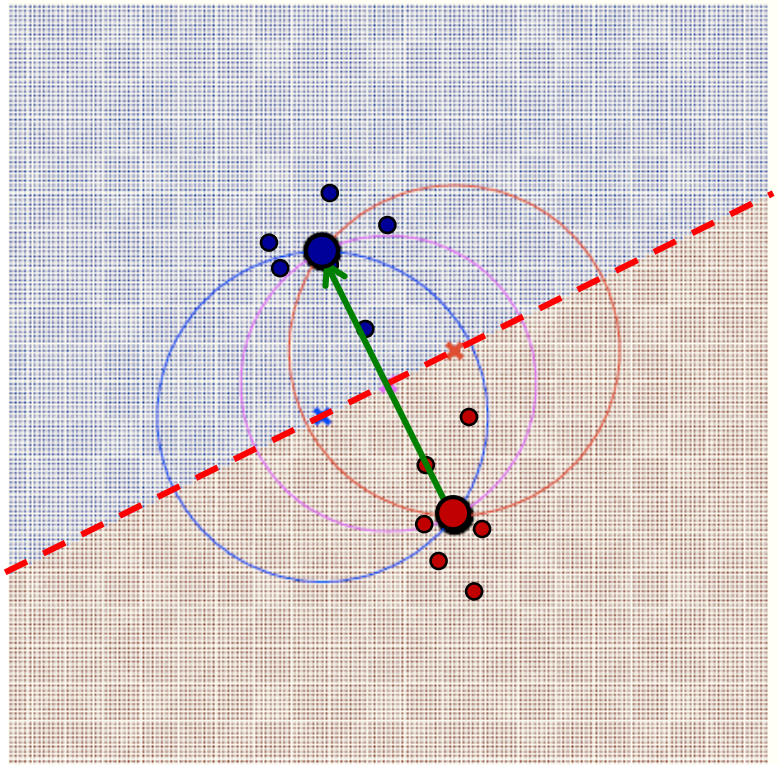
# 1-Nearest neighbor (1NN) classifier

- The simplest nearest neighbor classifier: Assign the new instance ***x*** to the nearest labeled training point (or exemplar)
  - Training = memorizing the training data
  - Each point is an exemplar, or exemplars are computed from the data
  - But it generalizes, unlike the lookup table approach
  - The *implicit* decision boundaries of a 1NN classifier comprise a Voronoi tesselation
    - Leads to piecewise linear decision boundaries

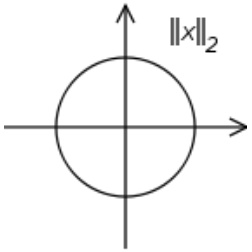


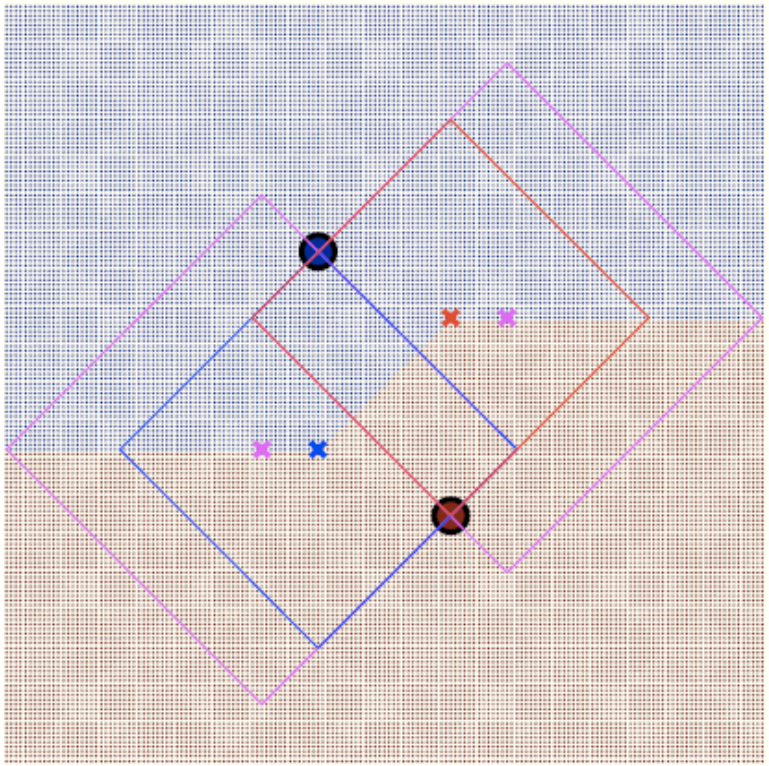


# Implicit 1NN decision boundaries (N=2)

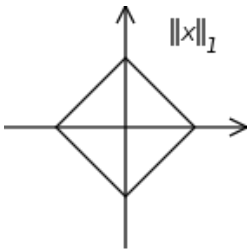


Euclidian (L2)

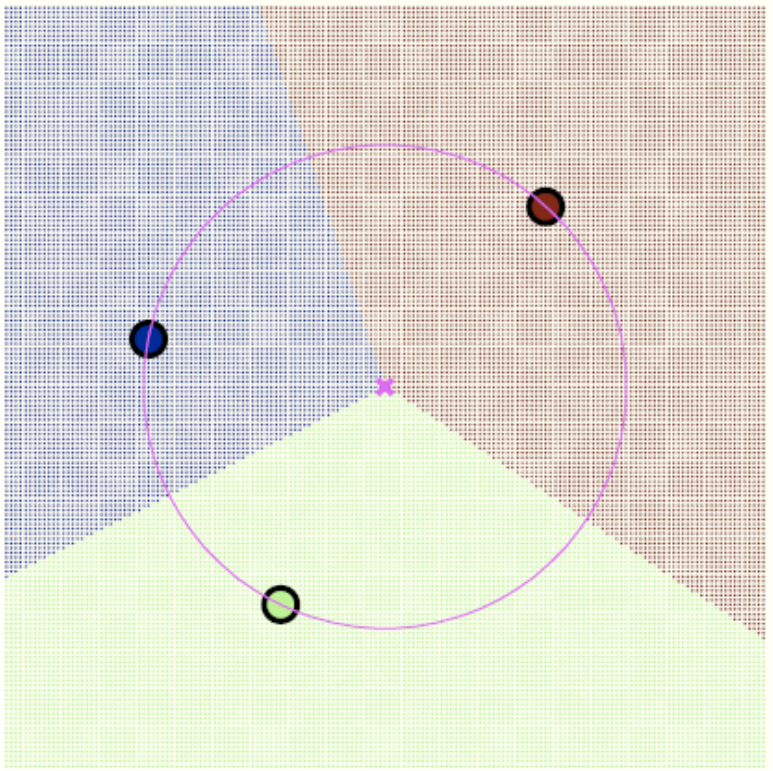


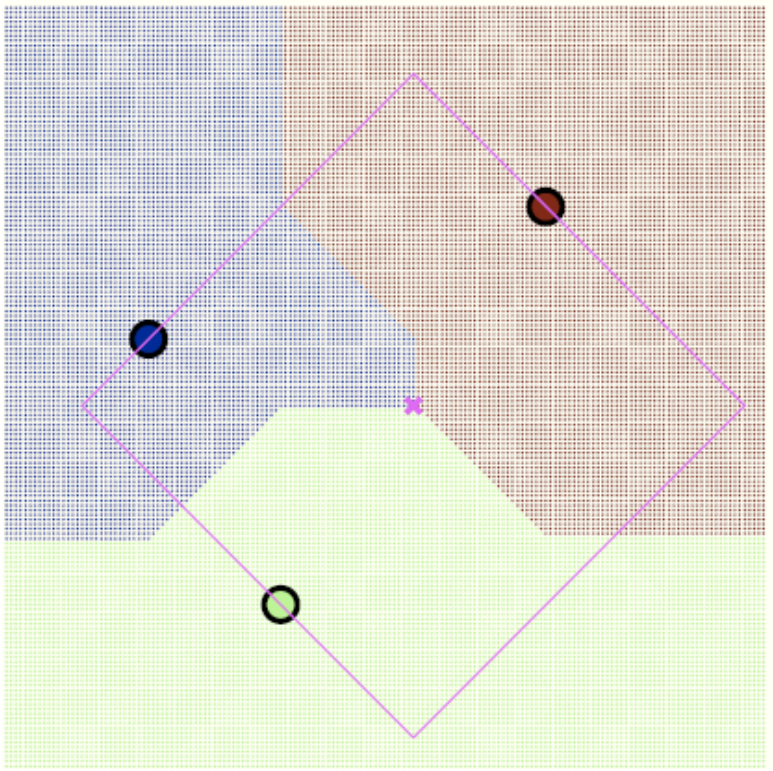
Manhattan (L1)



# Implicit 1NN decision boundaries (N=3)



Euclidian (L2)



Manhattan (L1)

Multi-class version of the basic linear classifier

# *k*-Nearest neighbor (*k*NN) classifiers

- In some cases, the *k-nearest neighbor* method is preferable:
  - Classify a new instance by taking a vote of the $k \geq 1$ nearest exemplars
  - E.g., in a binary classifier, with $k = 7$, for a new input point the 7 nearest neighbors may include 5 positives and 2 negatives, so we choose positive as the classification
- Or, instead of using a fixed *k*, vote among all neighbors within a fixed radius *r*
- Or, combine the two, stopping when (*count* > *k*) or (*dist.* > *r*)
- May also use distance weighting – the closer an exemplar is to the instance, the more its vote counts (e.g., $w_i = \frac{1}{D(\boldsymbol{x}, \boldsymbol{x}_i)}$)
- What about ties?
  - Preference to the 1NN
  - Random choice
  - Etc.