Machine Learning

CS 165B

Prof. Matthew Turk

Tuesday, May 31, 2016

Final exam review

and

ESCIs!

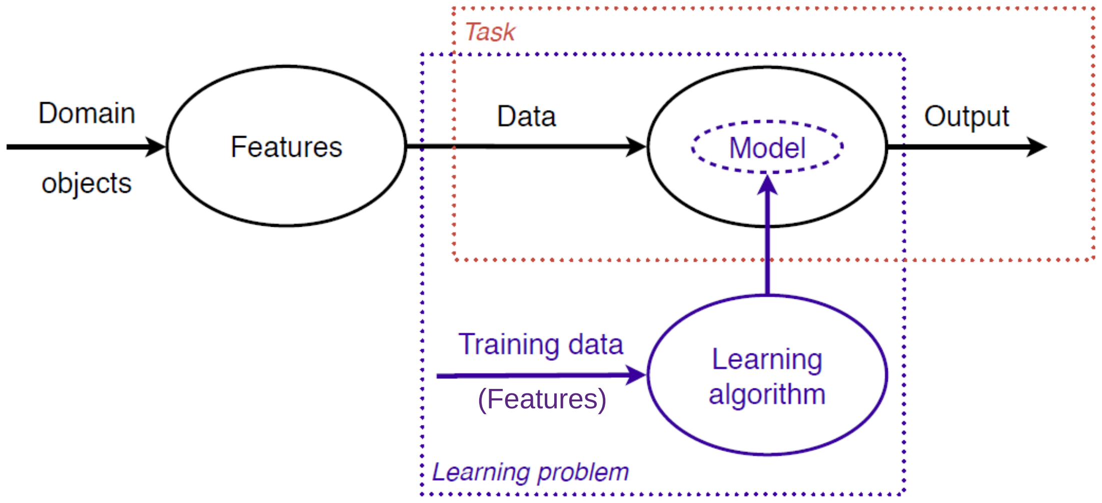
Notes

- Final exam (Wednesday, June 8, 8-11am)
  - Similar to the midterm in style – important to understand the concepts we've covered and how to apply them
  - Covers all material of the course
    - More weight on the material since the midterm
  - Practice exam will *probably* be provided
    - Good practice: midterm, homeworks
  - Closed book/notes
    - Exception: You may bring one 8.5"x11" sheet of paper with your notes (both sides)
    - I'll also provide some information, formulas, etc. (now posted and included with the final exam)

Final Exam Review

Machine learning

Machine learning is about using the right features to build the right models that achieve the right tasks



Features – how we describe our data objects

Model – a mapping from data points to outputs:

Output = *f* (*Data*)

This is what machine learning produces!

Task – an abstract representation of the problem we want to solve

Some machine learning problems

Training data includes desired outputs (input/output pairs)

Training data does not include desired outputs

Outputs

	Supervised learning	Unsupervised learning
Discrete	Classification / labeling Binary / multi-class	Clustering
Continuous	Regression Real-valued prediction	Dimensionality reduction

Also semi-supervised learning, reinforcement learning, etc.

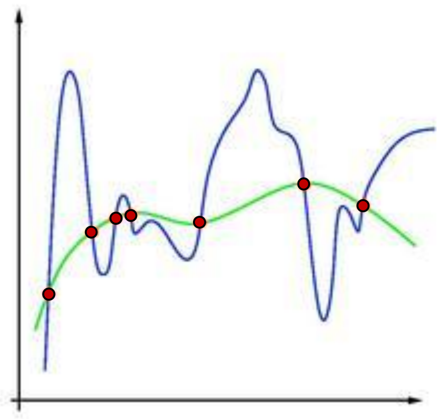
Training data includes *some* desired outputs

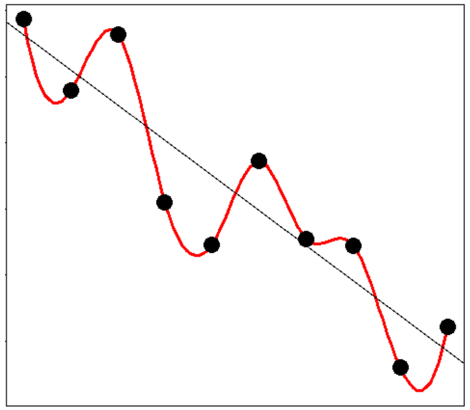
Learning through trial-and-error interactions with the environment

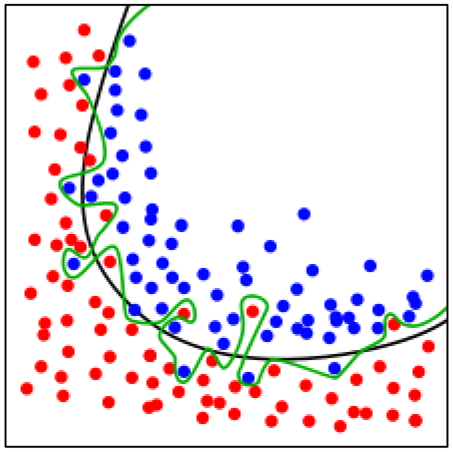
Overfitting

- *Overfitting* and *generalization* are important concepts in machine learning

- Overfitting: Learning that results in good performance on the training data but poor performance on the real task
  - Example: Memorization or lookup table
  - Example: Fitting a model to the data that has more parameters then needed







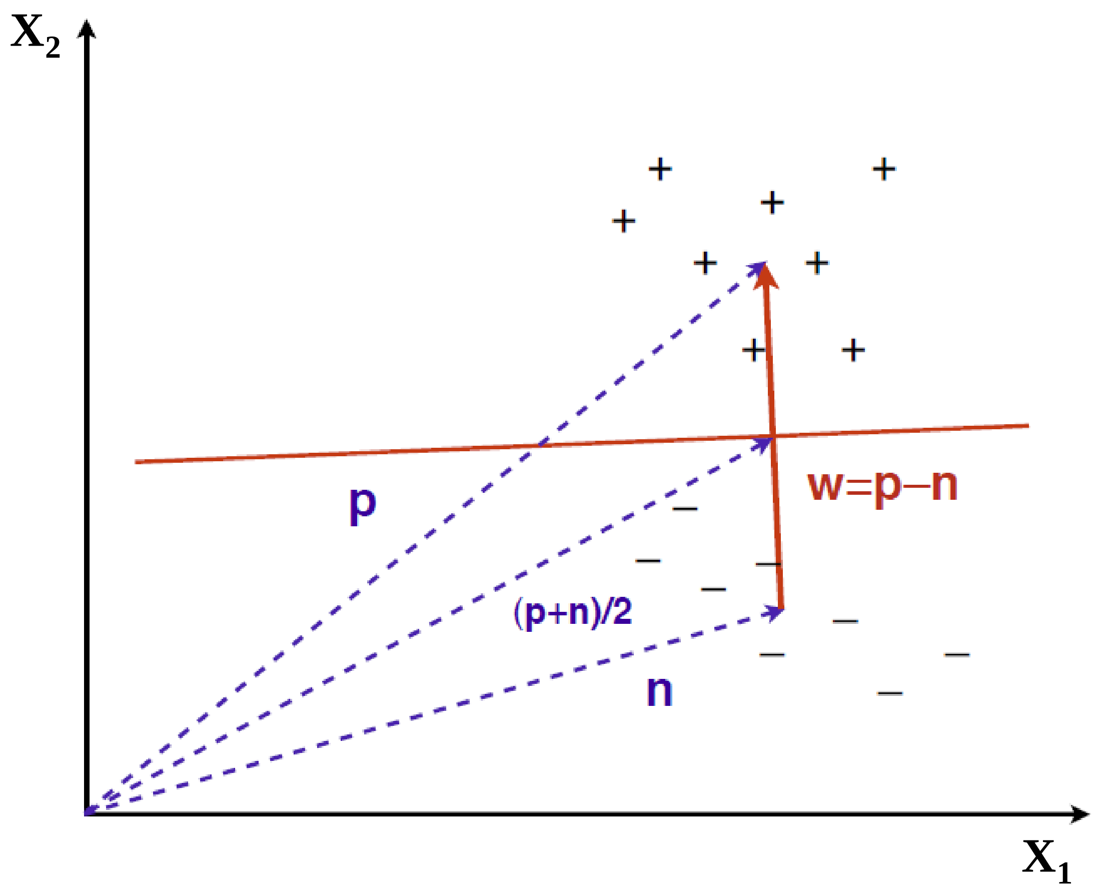
Generalization

- We want machine learning solutions that ***generalize*** to the range of inputs/data that will be seen – not just solutions that work well on the training data
- The real aim of machine learning is to do well on test data that is not known during learning
- Choosing values for the parameters that minimize the error on the training data is not necessarily the best policy.

- We want the learning machine to model the true regularities in the data and to ignore the noise in the data
  - But the learning machine does not know which regularities are real and which are accidental quirks of the particular set of training examples we happen to have! So we have to help….

Basic linear classifier

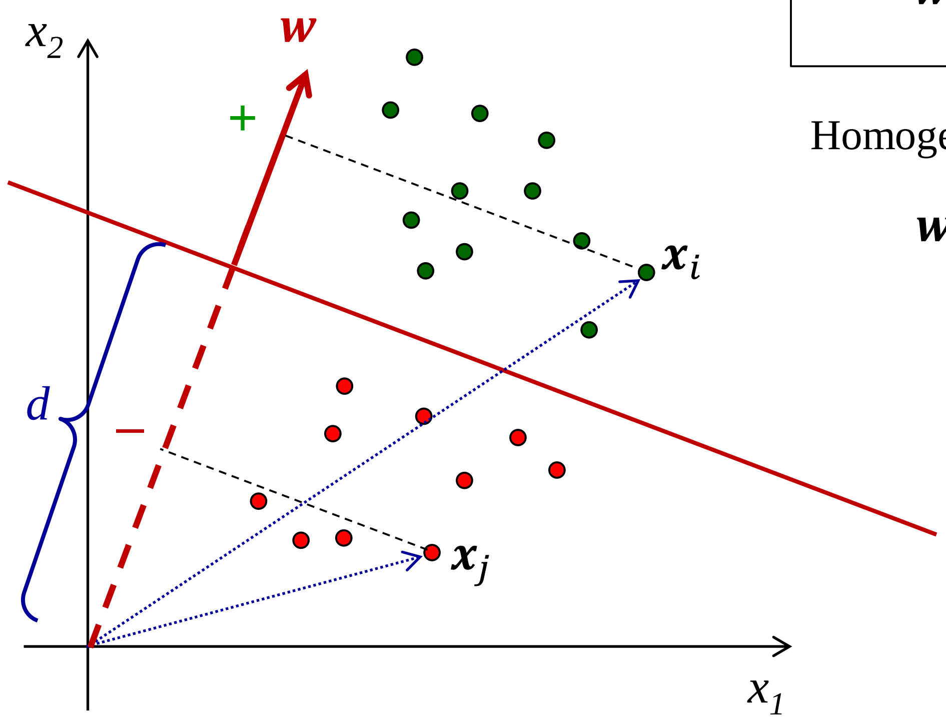
Constructs a linear decision boundary halfway between the positive and negative centers of mass of the two classes



$$f(x) = 1 \;\; if \;\; \mathbf{x} \cdot \mathbf{w} > t$$
$$0 \;\; otherwise$$

How to compute t ?

Classifier geometry – *w* and *t*



Non-homogeneous:

$$\boldsymbol{w}^T\boldsymbol{x} - t = \begin{bmatrix} w_1 & w_2 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} - t > 0$$

Homogeneous:

$$\boldsymbol{w}^T\boldsymbol{x} = \begin{bmatrix} w_1 & w_2 & -t \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix} > 0$$

Is $\boldsymbol{w}$ a unit vector?

Doesn't have to be

What's the relationship between $\boldsymbol{w}$ and t ?

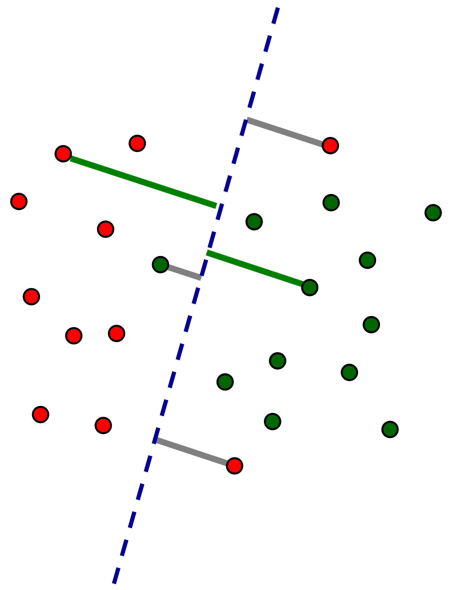
$(w, t) \equiv (kw, kt)$

$$\begin{bmatrix} w_1 & w_2 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} - t = 0 \qquad \begin{bmatrix} 2w_1 & 2w_2 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} - 2t = 0$$

These describe the same line

Classifier margin

- The margin (z) of a <u>sample</u> is its distance from the classification boundary
  - Positive if it's correctly classified
  - Negative if it's incorrectly classified



Perceptron margin for point $\boldsymbol{x}$:

$$\boldsymbol{z}(\boldsymbol{x}) = \frac{y(\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x} - t)}{||\boldsymbol{w}||} = \frac{m}{||\boldsymbol{w}||}$$

Non-homogeneous representation

Note: m is not the margin; it's the result of plugging $\boldsymbol{x}_i$ into $y(\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x} - t)$

Dimensionality reduction

- Let’s say we build a ML system to predict someone’s occupation. The 12 features given to us are:
  - First name, last name, SSN, father’s occupation, mother’s occupation, highest educational degree, last year’s salary, federal taxes paid last year, home address, model of car, miles driven last year, miles flown last year
- Some of these features may be useless, with no relevant information
- There may be redundancies – correlations among features
- Can we transform this 12-dimensional classification problem to a lower-dimensional problem?
  - Perhaps easier, computationally simpler…
- Yes, through dimensionality reduction

Key terminology

False positive rate (FPR) $= \frac{FP}{N} = \alpha$

False negative rate (FNR) $= \frac{FN}{P} = \beta$

True positive rate (TPR) $= \frac{TP}{P}$ = Sensitivity = Recall = 1 - β

True negative rate (TNR) $= \frac{TN}{N}$ = Specificity = 1 - α

Accuracy $= \frac{TP+TN}{P+N} = \left(\frac{P}{P+N}\right) TPR + \left(\frac{N}{P+N}\right) TNR$

Error rate $= \frac{FP+FN}{P+N}$

Precision $= \frac{TP}{\hat{P}}$

Accuracy + error rate = 1

Average recall $= \frac{TPR+TNR}{2}$

Actual class C

Predicted class $\hat{C}$

	1	0	
1	TP	FP	*Estimated positive* $\hat{P}$
0	FN	TN	*Estimated negative* $\hat{N}$
	Positives P	*Negatives* N	TOTAL

Regression – another predictive ML task

- In the classification tasks we've been discussing, the label space was a discrete set of classes
  - Classification, scoring, ranking, probability estimation
- Regression learns a function (the regressor) that is a mapping $\hat{f} : \mathcal{X} \to \mathbb{R}$ from examples – $f(x_i)$
  - I.e., the target variable (output) is real-valued
- Assumption: the examples will be noisy, so watch out for overfitting – need to capture the general trend or shape of the function, not exactly match every data point
  - E.g., if fitting an N-degree polynomial to the training data (thus N+1 parameters to estimate), choose one as low degree as possible
- The number of data points should be much greater than the number of parameters to be estimated!
  - How much data is needed? An open question in ML….

Linear least-squares regression

- Regression learns a function (the regressor) that is a mapping $\hat{f} : \mathcal{X} \to \mathbb{R}$; it's learned from examples, $(x_i, f(x_i))$
  - I.e., the target variable (output $\hat{f}(x)$) is real-valued
- Linear regression – the function is linear
  - Fit a line/plane/hyperplane to the data
- The difference between f and $\hat{f}$ is known as the residual ϵ

$$\epsilon_i = f(x_i) - \hat{f}(x_i)$$

- The least squares method minimizes the sum of the squared residuals – i.e., find $\hat{f}$ that minimizes $\sum_i {\epsilon_i}^2$ on the training data
- Univariate or multivariate regression
  - Univariate – one input variable
  - Multivariate – multiple input variables

Concept learning

- In concept learning, we want to learn a Boolean function over a set of attributes+values
  - I.e., derive a Boolean function from training examples
    - Positive and negative examples of the concept
      - Positive: Temperature = high ∧ Coughing = yes ∧ Spots = yes
      - Negative: Temperature = medium ∧ Coughing = no ∧ Spots = yes
  - This is our hypothesis
- The target concept ***c*** is the true concept
  - We want the hypothesis to be a good estimate of the true concept
  - Thus we wish to find ***h*** (or $\hat{c}$) such that ***h* = *c*** (or $\hat{c}$ = ***c***)
- The hypothesis is a Boolean function over the features
  - E.g., some combinations of {Temperature, Coughing, Spots} are <u>in</u> the concept, and others are <u>not in</u> the concept

The hypothesis space

- Using a set of features, what concepts can possibly be learned?
- The space of all possible concepts is called the hypothesis space
  - What is the hypothesis space for a given problem?
- First, how many possible instances are there for a given set of features?
  - In set theory, the Cartesian product of all the features
  - $F_1 \times F_2 \times \ldots \times F_N$
  - All combinations of feature values
  - UCSB courses: Quarter (4), Dept (40), courselevel (2), topic (500)
  - $4 \times 40 \times 2 \times 500 = 160{,}000$ possible instances
    - E.g., (spring, CS, ugrad, ML), (fall, Music, grad, StringTheory), …

The perceptron

- The perceptron model is an iterative linear classifier that will achieve perfect separation on linearly separable data
- A perceptron iterates over the training data, updating $\boldsymbol{w}$ every time it encounters an incorrectly classified example
  - How to move the boundary for a misclassified example?
  - How much to move it?
- Update rule (homogeneous training data $\boldsymbol{x}_i \in \mathbb{R}^{k+1}$):

  $\boldsymbol{w}' = \boldsymbol{w} + \eta y_i \boldsymbol{x}_i$ where η is the learning rate, $0 < \eta \leq 1$
- Iterate through the training examples (each pass over the data is called an epoch) until all examples in an epoch are correctly classified
- Guaranteed to converge if the training data is linearly separable – but won’t converge otherwise

The perceptron training algorithm

When will this algorithm halt?

$D = \{ (\mathbf{x}_i, y_i) \}$

Algorithm Perceptron(D, η) – train a perceptron for linear classification.

```
Input  : labelled training data $D$ in homogeneous coordinates; learning rate $\eta$.
Output : weight vector $\mathbf{w}$ defining classifier $\hat{y} = \text{sign}(\mathbf{w} \cdot \mathbf{x})$.
$\mathbf{w} \leftarrow \mathbf{0}$ ;                  // Other initialisations of the weight vector are possible
$converged \leftarrow$ false;
while $converged =$ false do
    $converged \leftarrow$ true;
    for $i = 1$ to $|D|$ do
        if $\boxed{y_i \mathbf{w} \cdot \mathbf{x}_i \leq 0}$          // i.e., $\hat{y}_i \neq y_i$     Misclassified
        then
            $\boxed{\mathbf{w} \leftarrow \mathbf{w} + \eta y_i \mathbf{x}_i;}$
            $converged \leftarrow$ false;        // We changed $\mathbf{w}$ so haven't converged yet
        end
    end
end
```

If a positive example is misclassified, add it to $\boldsymbol{w}$
If a negative example is misclassified, subtract it from $\boldsymbol{w}$

All components of homogeneous $\boldsymbol{w}$ are updated (including $\boldsymbol{w}_{k+1} = -t$)

Perceptron duality

- Every time a training example $\boldsymbol{x}_i$ is misclassified, the amount $\eta y_i \boldsymbol{x}_i$ is added to the weight vector $\boldsymbol{w}$
- After training is completed, each example $\boldsymbol{x}_i$ has been misclassified α_i times
- Thus the weight vector can be written as

$$\boldsymbol{w} = \eta \sum_{i} \alpha_i y_i \boldsymbol{x}_i$$

Assuming the initial value of $\boldsymbol{w}$ was initialize to $\boldsymbol{0}$

  So the weight vector is a linear combination of the training instances

- So, alternatively, we can view perceptron learning as learning the α_i coefficients and then, when finished, constructing $\boldsymbol{w}$
  - This perspective comes up again (soon) in support vector machines

Perceptron training in dual form

Algorithm DualPerceptron(D) – perceptron training in dual form.

```
Input   : labelled training data $D$ in homogeneous coordinates.
Output  : coefficients $\alpha_i$ defining weight vector $\mathbf{w} = \sum_{i=1}^{|D|} \alpha_i y_i \mathbf{x}_i$.
$\alpha_i \leftarrow 0$ for $1 \le i \le |D|$;
$converged \leftarrow$ false;
while $converged =$ false do
    $converged \leftarrow$ true;
    for $i = 1$ to $|D|$ do
        if $y_i \sum_{j=1}^{|D|} \alpha_j y_j \boxed{\mathbf{x}_i \cdot \mathbf{x}_j} \le 0$ then
            $\alpha_i \leftarrow \alpha_i + 1$;
            $converged \leftarrow$ false;
        end
    end
end
```

Misclassified (from regular Perceptron algorithm)

$\boxed{\text{if } y_i \mathbf{w} \cdot \mathbf{x}_i \le 0 \quad // \text{ i.e., } \hat{y}_i \ne y_i}$

$$\text{if } \; y_i \overset{\boldsymbol{w}}{\boxed{\sum_{j=1}^{|D|} \alpha_j y_j \boldsymbol{x}_j}} \cdot \boldsymbol{x}_i \le 0$$

Dot products of training examples

Contained in the Gram matrix $\mathbf{G} = \mathbf{X}^T\mathbf{X}$

$\mathbf{G}_{ij} = \boldsymbol{x}_i^T \boldsymbol{x}_j$

The Gram matrix is typically computed in advance for computational efficiency

This notation assumes $\mathbf{X}$ is a matrix in which each <u>column</u> is a vector $\boldsymbol{x}_i$

Support vector machine (SVM)

- A support vector machine (SVM) is a linear classifier whose decision boundary is a linear combination of the support vectors
- In an SVM, we find classifier parameters $(\boldsymbol{w}, t)$ to maximize the margin
- Since $m = y(\boldsymbol{w}^T\boldsymbol{x} - t)$ and we wish to maximize the margin $\frac{m}{||\boldsymbol{w}||}$, we can instead fix $m = 1$ and minimize $||w||$
  - Provided that none of the training points fall inside the margin
- This leads to a constrained optimization problem:

$$\mathbf{w}^*, t^* = \underset{\mathbf{w},t}{\arg\min} \frac{1}{2}||\mathbf{w}||^2 \qquad \text{subject to } y_i(\mathbf{w}\cdot\mathbf{x}_i - t) \geq 1, 1 \leq i \leq n$$

  - Then, after some magical quadratic optimization (p. 212-214)….

Soft margin SVM

ξ_i = “xi”

- We introduce a slack variable ξ_i for each training example to account for margin errors
  - Points that are inside the margin
  - Points that are on the wrong side of the decision boundary

$$\boldsymbol{w}^T \boldsymbol{x}_i - t \geq 1 - \xi_i \qquad \xi_i > 0 \rightarrow \boldsymbol{x}_i \text{ is not a support vector}$$

- Results in the soft margin optimization problem:

$$\mathbf{w}^*, t^*, \xi_i^* = \underset{\mathbf{w}, t, \xi_i}{\arg\min} \left[\frac{1}{2} ||\mathbf{w}||^2 + C \sum_{i=1}^{n} \xi_i \right]$$

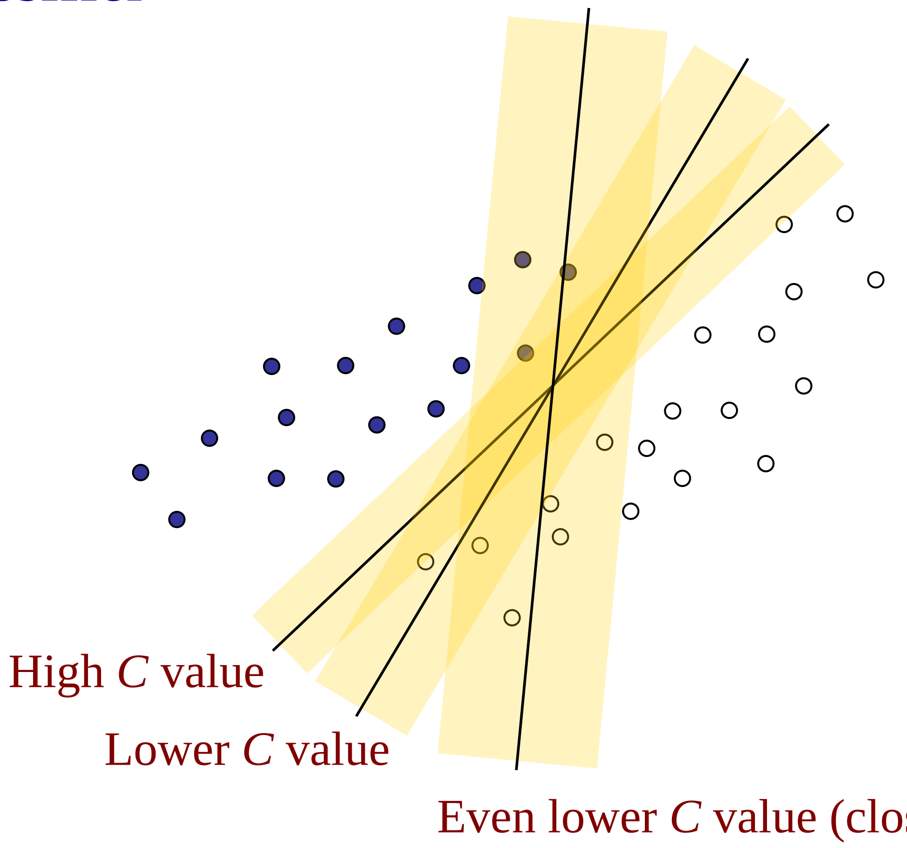
$$\text{subject to } y_i(\mathbf{w} \cdot \mathbf{x}_i - t) \geq 1 - \xi_i \text{ and } \xi_i \geq 0, 1 \leq i \leq n$$

The complexity parameter C is a user-defined parameter that allows for a tradeoff between maximizing the margin (lower C) and minimizing the margin errors (higher C)

- *Note that when C = 0, this gives no penalty to outliers – which makes it equivalent to our basic linear classifier!*

Soft margin SVM

A minimal-complexity (low *C*) soft margin classifier summarizes the classes by their class means in a way very similar to the basic linear classifier



Perceptron and SVM binary classifiers – summary

- In both perceptron and SVM learning, the linear decision boundary is a linear combination of the training data points
  - In the perceptron, just the ones that get misclassified in the iterative training
  - In the SVM, just the (few) support vectors

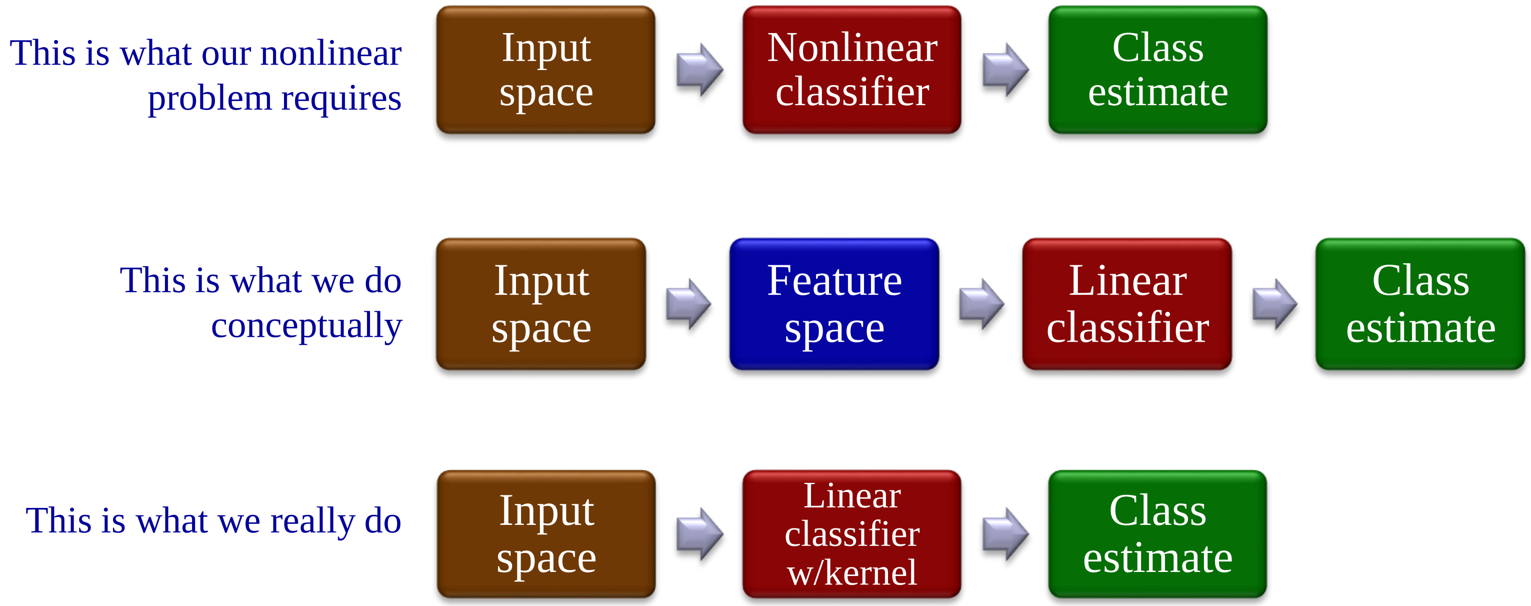
- Both learning methods have a dual form in which the dot product of training data points $\boldsymbol{x}_i^T \boldsymbol{x}_j$ is part of the main computation
  - All values of $\boldsymbol{x}_i^T \boldsymbol{x}_j$ are contained in the Gram matrix

$$\boldsymbol{G} = \boldsymbol{X}^T \boldsymbol{X} = [\boldsymbol{x}_1 \quad \boldsymbol{x}_2 \quad \dots \quad \boldsymbol{x}_k]^T [\boldsymbol{x}_1 \quad \boldsymbol{x}_2 \quad \dots \quad \boldsymbol{x}_k]$$

so it's often efficient to compute the Gram matrix in advance and index into it, rather than computing the dot products over and over again

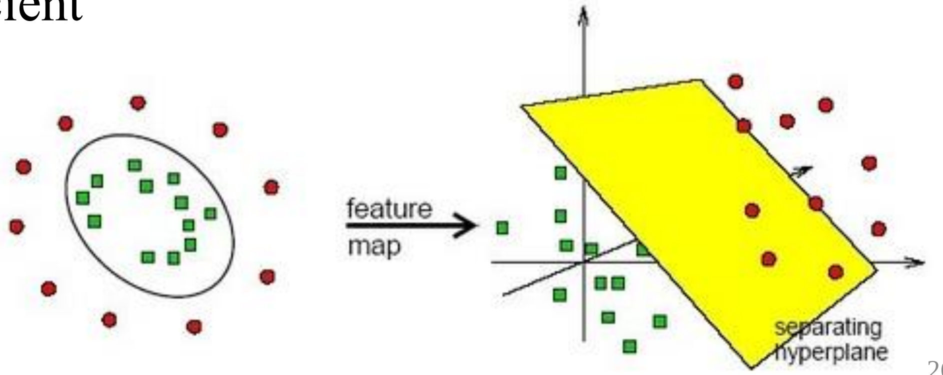
Nonlinear kernel classifiers

- 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***!


This is what our nonlinear problem requires
Input space
Nonlinear classifier
Class estimate
This is what we do conceptually
Input space
Feature space
Linear classifier
Class estimate
This is what we really do
Input space
Linear classifier w/kernel
Class estimate

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….



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?

Kernel perceptron

Learns a nonlinear decision boundary

Algorithm KernelPerceptron(D, κ) – perceptron training algorithm using a kernel.

```
Input   : labelled training data $D$ in homogeneous coordinates;
          $\boxed{\text{kernel function } \kappa.}$
Output  : coefficients $\alpha_i$ defining non-linear decision boundary.
$\alpha_i \leftarrow 0$ for $1 \le i \le |D|$;
$converged \leftarrow$ false;
while $converged =$ false do
    $converged \leftarrow$ true;
    for $i = 1$ to $|D|$ do
        if $y_i \sum_{j=1}^{|D|} \alpha_j y_j \boxed{\kappa(\mathbf{x}_i, \mathbf{x}_j)} \le 0$ then
            $\alpha_i \leftarrow \alpha_i + 1$;
            $converged \leftarrow$ false;
        end
    end
end
```

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

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 α_i parameters, we can then classify a new instance $\boldsymbol{x}$ using

$$\sum_{i=1}^{n} \alpha_i y_i \, \kappa(\boldsymbol{x}, \boldsymbol{x}_i) > 0$$

- 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 ξ_i and the complexity parameter C

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 σ

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

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)

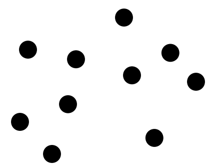
- But similarity and grouping may not be obvious…
- We'd like to define features and distance measures that will capture the intended notion of similarity

Distance $\propto$ dissimilarity

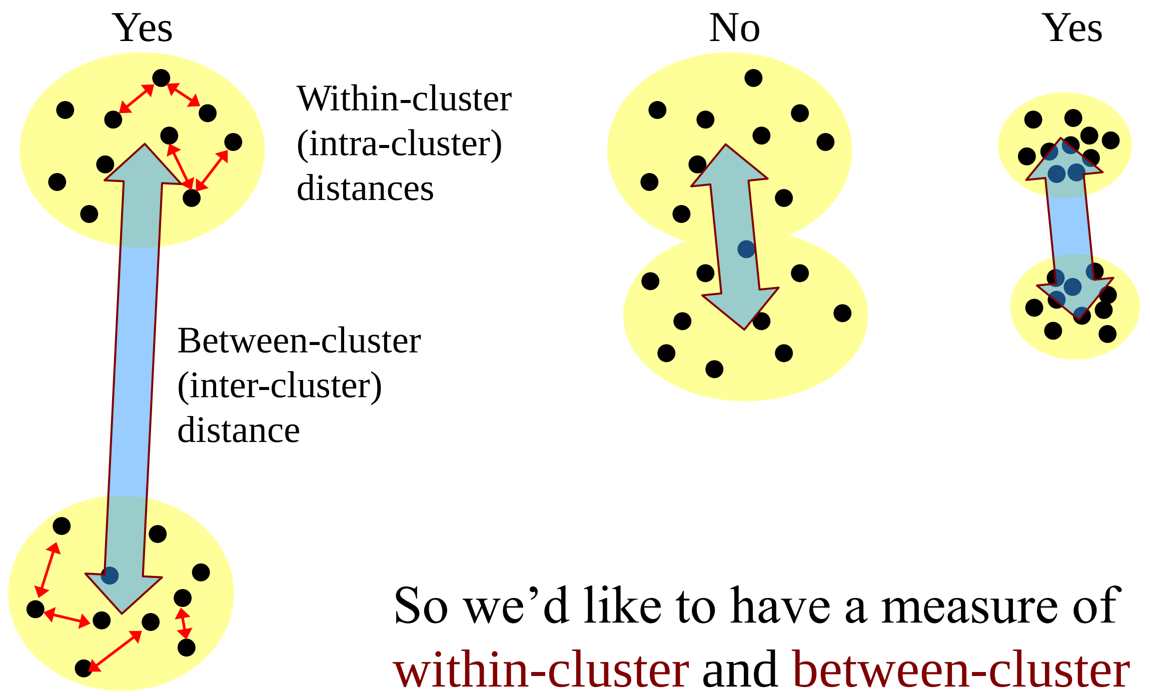
Clustering

- The goal of clustering is to find clusters (groupings) that are compact with respect to the distance metric
- What do we mean by *compactness*?

Is this cluster compact?



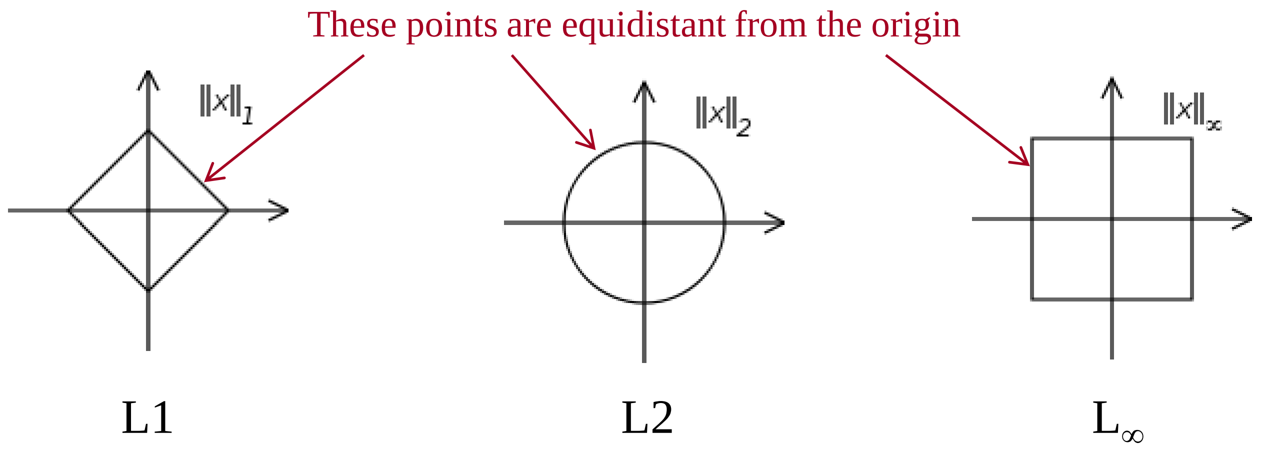
It depends….



So we'd like to have a measure of within-cluster and between-cluster distribution or ***scatter***

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_∞ 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

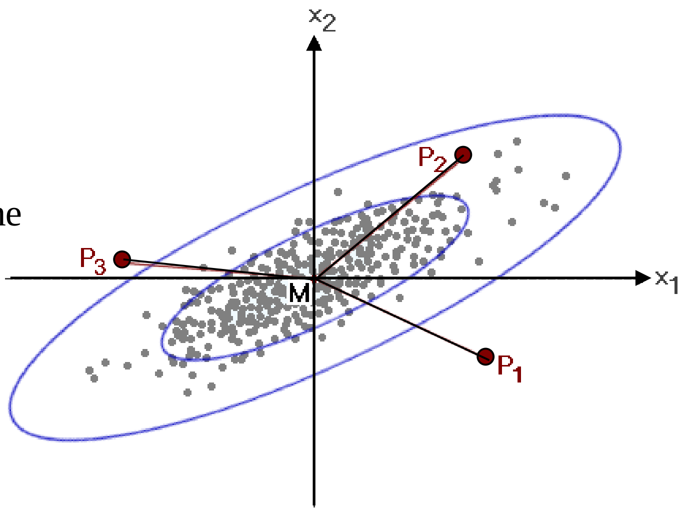
$$D_M(\boldsymbol{x}, \boldsymbol{y}) = \sqrt{(\boldsymbol{x} - \boldsymbol{y})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{x} - \boldsymbol{y})}$$

1. The distance between $\boldsymbol{x}$ and the origin $\boldsymbol{y}$, or
2. The distance between two variables that have the same distribution (and thus the same covariance matrix)

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

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

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 $\boldsymbol{x}$ to the nearest exemplar using Euclidian distance

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

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.

K-means clustering

- The general K-means clustering problem is NP-complete, so there is no efficient solution to find the optimal clustering (data partition)
- A widely-used heuristic algorithm for clustering is also known as the K-means algorithm, but it is not optimal
  - It will converge to a solution, but there is no guarantee that the solution is the best one (the global minimum of scatter)
  - But it works quite well in most cases!
- Typically, the K-means algorithm would be run several times (with a random starting point) and then the best solution is selected
  - I.e., the solution with the smallest within-cluster scatter

K-means algorithm

K is an input parameter

Algorithm KMeans(D, K) – K-means clustering using Euclidean distance Dis_2.

```
Input  : data $D \subseteq \mathbb{R}^d$; number of clusters $K \in \mathbb{N}$.
Output : $K$ cluster means $\mu_1, \ldots, \mu_K \in \mathbb{R}^d$.
randomly initialise $K$ vectors $\mu_1, \ldots, \mu_K \in \mathbb{R}^d$;
repeat
    assign each $\mathbf{x} \in D$ to $\arg\min_j \mathrm{Dis}_2(\mathbf{x}, \mu_j)$;    ⟵ 1-Nearest neighbor assignment
    for $j = 1$ to $K$ do
        $D_j \leftarrow \{\mathbf{x} \in D | \mathbf{x}$ assigned to cluster $j\}$;    ⟵ Partition defined by assignment
        $\mu_j = \frac{1}{|D_j|} \sum_{\mathbf{x} \in D_j} \mathbf{x}$;    ⟵ Re-compute the cluster mean
    end
until no change in $\mu_1, \ldots, \mu_K$;
return $\mu_1, \ldots, \mu_K$;
```

K-medoids algorithm

Algorithm KMedoids(D, K, Dis) – K-medoids clustering using arbitrary distance metric Dis.

```
Input   : data $D \subseteq \mathscr{X}$; number of clusters $K \in \mathbb{N}$;
          distance metric $\mathrm{Dis} : \mathscr{X} \times \mathscr{X} \to \mathbb{R}$.
Output  : $K$ medoids $\mu_1, \dots, \mu_K \in D$, representing a predictive clustering of $\mathscr{X}$.
randomly pick $K$ data points $\mu_1, \dots, \mu_K \in D$;
repeat
    assign each $\mathbf{x} \in D$ to $\arg\min_j \mathrm{Dis}(\mathbf{x}, \mu_j)$;
    for $j = 1$ to $K$ do
        $D_j \leftarrow \{\mathbf{x} \in D | \mathbf{x}$ assigned to cluster $j\}$;
        $\mu_j = \arg\min_{\mathbf{x} \in D_j} \sum_{\mathbf{x}' \in D_j} \mathrm{Dis}(\mathbf{x}, \mathbf{x}')$;   $\longleftarrow$ Re-compute the cluster medoid
    end
until no change in $\mu_1, \dots, \mu_K$;
return $\mu_1, \dots, \mu_K$;
```

Kernel K-means clustering

```
Algorithm Kernel-KMeans($D, K$) – $K$-means clustering using kernelised distance $\mathrm{Dis}_\kappa$.

Input   : data $D \subseteq \mathscr{X}$; number of clusters $K \in \mathbb{N}$.
Output  : $K$-fold partition $D_1 \uplus \ldots \uplus D_K = D$.
randomly initialise $K$ clusters $D_1, \ldots, D_K$;
repeat
    assign each $\mathbf{x} \in D$ to $\arg\min_j \frac{1}{|D_j|} \sum_{\mathbf{y} \in D_j} \mathrm{Dis}_\kappa(\mathbf{x}, \mathbf{y})$;
    for $j = 1$ to $K$ do
        $D_j \leftarrow \{\mathbf{x} \in D | \mathbf{x}$ assigned to cluster $j\}$;
    end
until no change in $D_1, \ldots, D_K$;
return $D_1, \ldots, D_K$;
```

$\longleftarrow$ *Re-assign each point to a partition according to the minimum average (kernel) distance*

As before, replace the dot product with a kernel function ***κ***

$$\mathrm{Dis}_2(\mathbf{x}, \mathbf{y}) = ||\mathbf{x} - \mathbf{y}||_2 = \sqrt{(\mathbf{x} - \mathbf{y}) \cdot (\mathbf{x} - \mathbf{y})} = \sqrt{\mathbf{x} \cdot \mathbf{x} - 2\mathbf{x} \cdot \mathbf{y} + \mathbf{y} \cdot \mathbf{y}}$$

$$\mathrm{Dis}_\kappa(\mathbf{x}, \mathbf{y}) = \sqrt{\kappa(\mathbf{x}, \mathbf{x}) - 2\kappa(\mathbf{x}, \mathbf{y}) + \kappa(\mathbf{y}, \mathbf{y})}$$

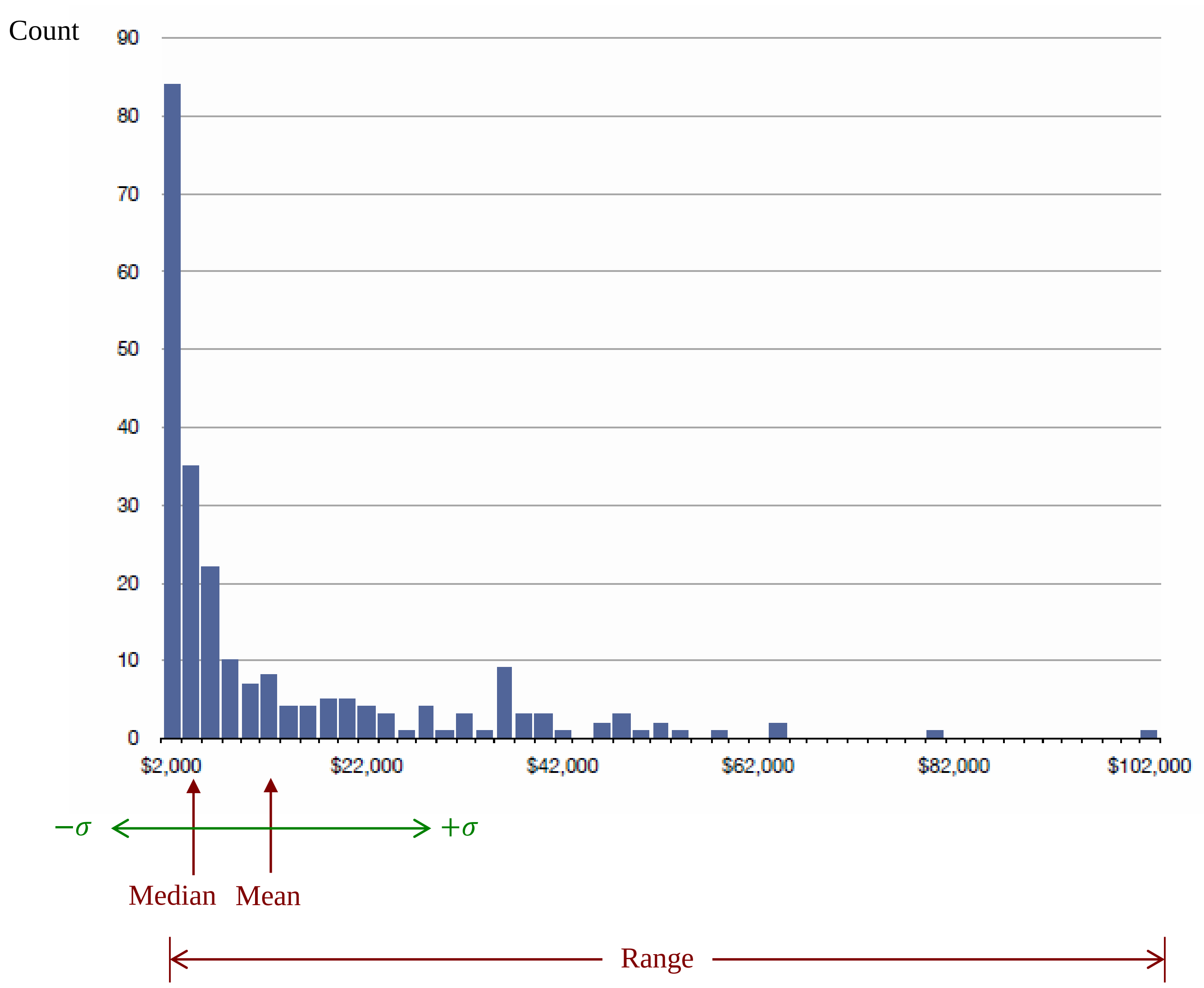
Main feature types

- Quantitative features
  - Measured on a meaningful numeric scale
  - Domain is often real values
  - E.g.: weight, height, age, angle, match to template, …
- Ordinal features
  - The relevant information is the ordering, not the scale
  - Domain is an ordered set
  - E.g., rank, street addresses, preference, ratings, …
- Categorical features
  - No scale or order information
  - Domain is an unordered set
  - E.g., colors, names, parts of speech, binary attributes, …

Histogram

Shows data as a frequency plot, with data collected in *bins*

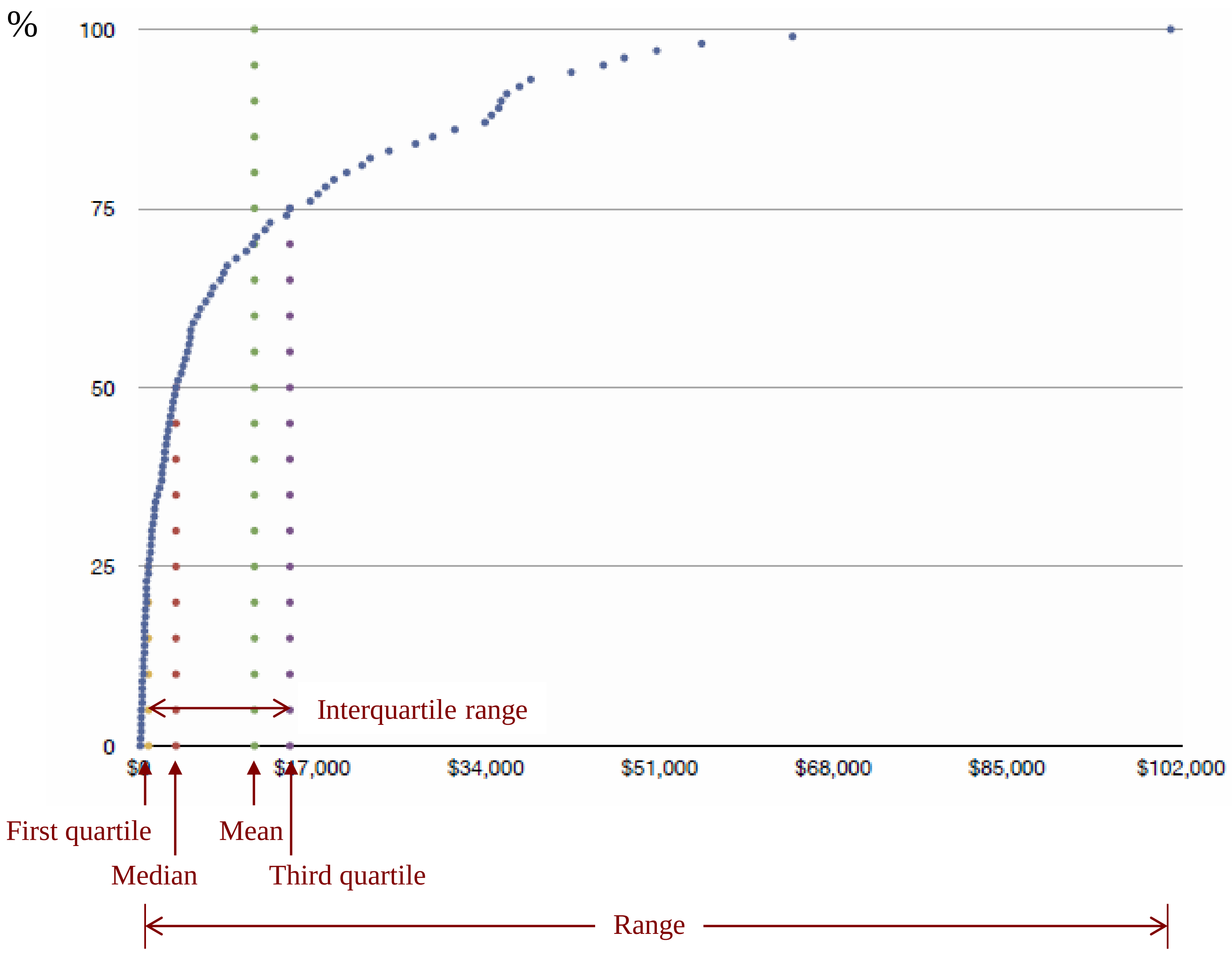
A histogram of GDP per capita for 231 countries (bin size = $2000):



Percentile plot

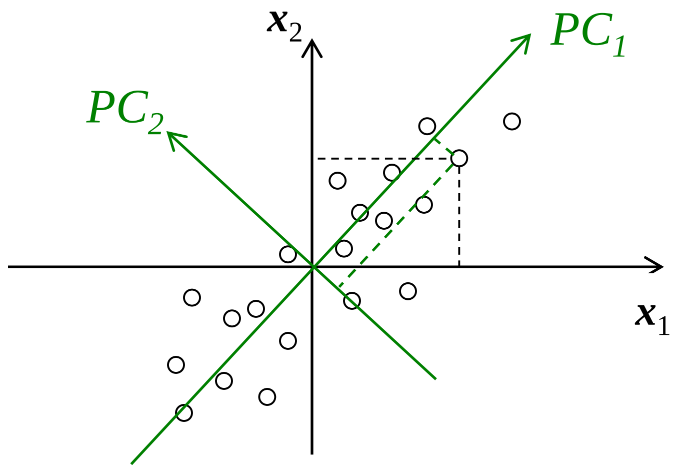
Shows cumulative fraction of data – what % fall below a given value

A percentile plot of GDP per capita for 231 countries:



PCA and eigendecomposition

- One of the most widely used feature construction/selection techniques is Principal Component Analysis (PCA)
  - PCA constructs new features that are linear combinations of given features
- Computed eigenvectors and eigenvalues hold useful information
- Often used for dimensionality reduction, finding the intrinsic linear structure in the data



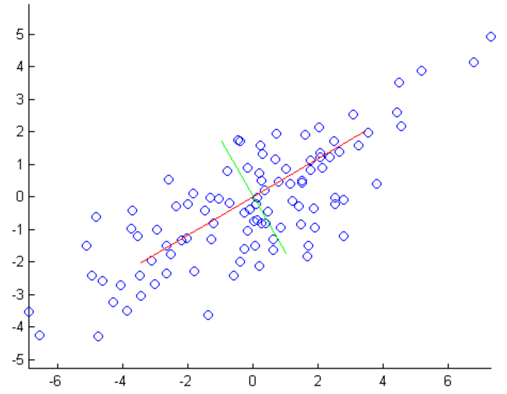
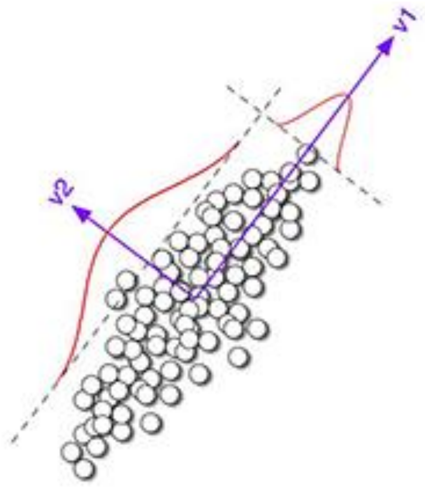
Given features 1 and 2 (x_1, x_2)
Computed features 1 and 2 (green axes)

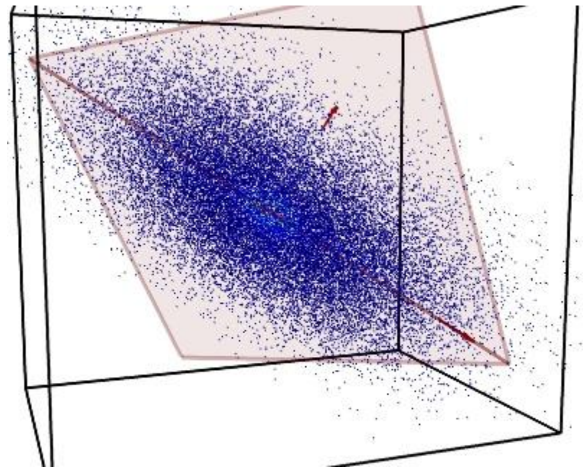
$$\mathrm{PC}_1 = \underset{y}{\mathrm{argmax}} (y^T X)(X^T y)$$

(maximize variance of points projected onto unit vector y)

PCA and eigendecomposition

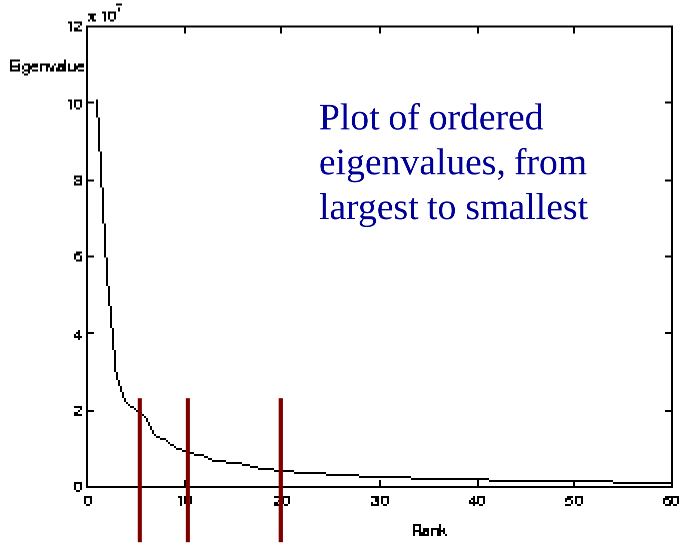
- The first principal component is the direction of maximum (1D) variance in the data
- The second principal component is the direction, perpendicular to the 1st PC, of maximum variance in the data
  - Etc. for additional PCs
- For N-dimensional data, there are N principal components
  - But perhaps only *k* of them are useful (k < N) – dimensionality reduction!
- PCA typically assumes zero-mean data
  - First subtract the mean from each data point; centroid is thus (0, 0)



PCA for dimensionality reduction

- So the eigenvalues can give clues to the inherent dimensionality of the data, or at least provide a way to more efficiently approximate high-dimensional with lower-dimensional feature vectors
- For example:



60-dimensional data (60 eigenvectors and eigenvalues)

Many of the eigenvalues are small, meaning that their associated eigenvectors don’t contribute much to the representation of the data

We can choose a cutoff – say, only use the first 20 eigenvectors (or 10, or 5)

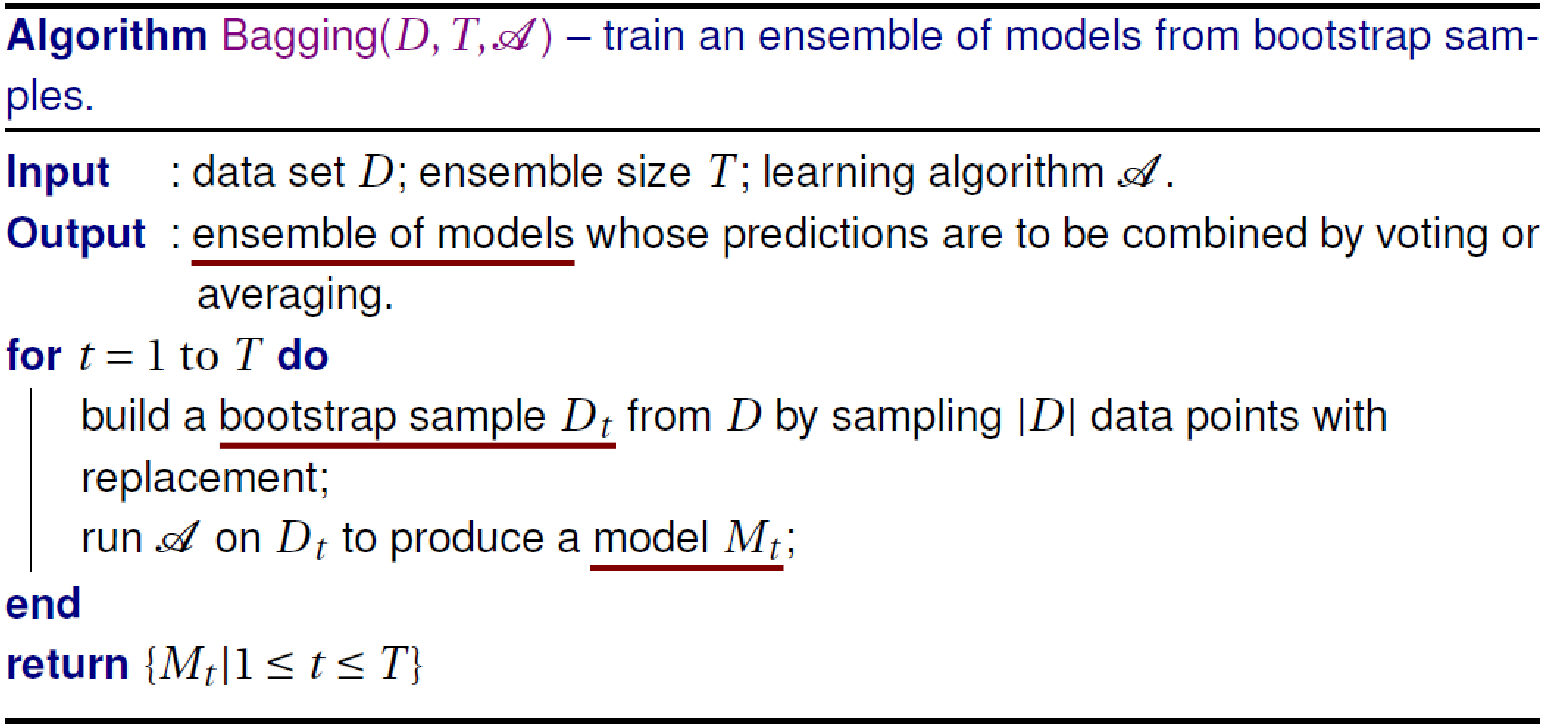
Model ensembles

- We’ve seen how combining features can be beneficial
- We can also combine models to increase performance
  - Combinations of models are known as model ensembles
  - Potential for better performance at the cost of increased complexity
- General approach to model ensembles:
  - Construct multiple different models from adapted versions of the training data (e.g., reweighted or resampled)
  - Combine the predictions of these models in some way (averaging, voting, weighted voting, etc.)
- Two of the best-known ensemble methods are bagging and boosting
- These are “meta” methods – i.e., they are independent of the particular learning method (linear classifier, SVM, etc.)

Bagging

- Bagging = "bootstrap aggregation"
  - Create T models on different random samples of the training data set
  - Each sample is a set of training data called a bootstrap sample
    - Could be any size – often |D| is used, the size of the training set
- Bootstrap samples: Sample the data set with replacement (i.e., a sample can be chosen more than once in a bootstrap sample)
  - For j = 1 to |D|, choose with uniform probability from training data points $D = \{ d_1, d_2, \ldots, d_{|D|} \}$
  - Gather these into the bootstrap sample D_i
- Use $\{ D_1, D_2, \ldots, D_T \}$ to train models $\{ M_1, M_2, \ldots, M_T \}$, then combine the predictions of the T models
- Differences between the T bootstrap samples create diversity among the models in the ensemble

Bagging



Algorithm Bagging($D, T, \mathscr{A}$) – train an ensemble of models from bootstrap samples.

```
Input  : data set $D$; ensemble size $T$; learning algorithm $\mathscr{A}$.
Output : ensemble of models whose predictions are to be combined by voting or
         averaging.
for $t = 1$ to $T$ do
    build a bootstrap sample $D_t$ from $D$ by sampling $|D|$ data points with
    replacement;
    run $\mathscr{A}$ on $D_t$ to produce a model $M_t$;
end
return $\{M_t | 1 \leq t \leq T\}$
```

Bagging with tree models

Algorithm RandomForest(D, T, d) – train an ensemble of tree models from bootstrap samples and random subspaces.

```
Input  : data set $D$; ensemble size $T$; subspace dimension $d$.
Output : ensemble of tree models whose predictions are to be combined by
         voting or averaging.
for $t = 1$ to $T$ do
    build a bootstrap sample $D_t$ from $D$ by sampling $|D|$ data points with
    replacement;
    select $d$ features at random and reduce dimensionality of $D_t$ accordingly;
    train a tree model $M_t$ on $D_t$ without pruning;
end
return $\{M_t | 1 \leq t \leq T\}$
```

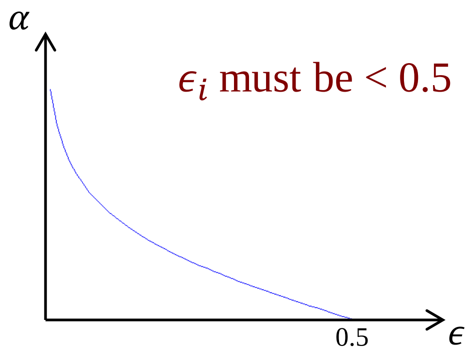
Boosting

- Boosting is similar to bagging, but it uses a more sophisticated technique to create diverse training sets
- The focus is on adding classifiers that do better on the misclassifications from earlier (Boolean) classifiers
  - By giving them a higher weight
- As with bagging, boosting attempts to create a *strong classifier* (*learner*) from a set of *weak classifiers* (*learners*)
  - The resulting ensemble model is less susceptible to overfitting
- Adaboost (“adaptive boosting”)
  - As long as the performance of each classifier is better than chance ($\epsilon < 0.5$), it is guaranteed to converge to a **better** classifier
- Can be extended to multi-class classification

Boosting

- Procedure:
  - Train a classifier; assign it a confidence factor α_i based on the error rate ϵ_i

$$\alpha_i = \frac{1}{2} \ln \frac{1-\epsilon_i}{\epsilon_i}$$



  - Give misclassified instances a higher weight
    - Assign half of the total weight to the misclassified examples

Misclassified point

$$w' = \frac{w}{2\epsilon_i}$$

Correctly classified point

$$w' = \frac{w}{2(1-\epsilon_i)}$$

Weights will sum to 0.5+0.5=1

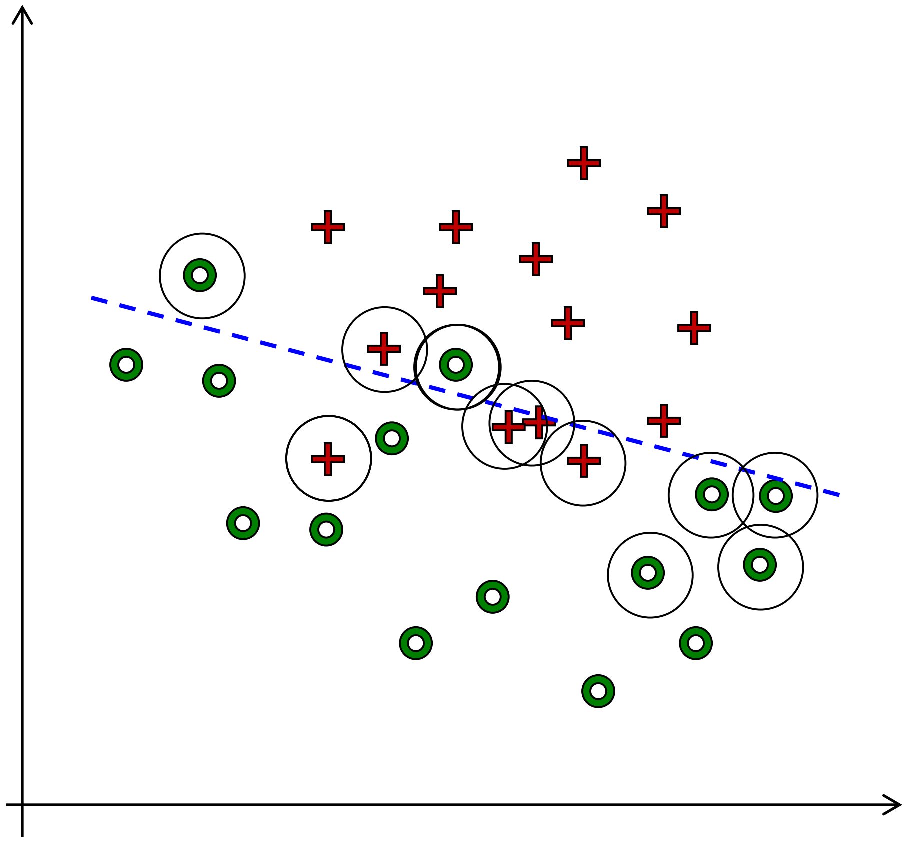
  - Repeat for *T* classifiers
  - The ensemble predictor is a weighted average of the models (rather than majority vote)

$$M(x) = \sum_{t=1}^{T} \alpha_t M_t(x)$$

  - Threshold for binary output

Boosting example

29 points
Initial weights w = 1/29 = 0.034



Round 1:
2 + 4 = 6 errors

$$\epsilon_1 = \frac{6}{29} = 0.21$$

$$\alpha_1 = \frac{1}{2}\ln\frac{1-\epsilon_1}{\epsilon_1} = 0.67$$

$w' = \{\ 2.42w,\ 0.63w\ \}$

Round 2:
2 + 5 = 7 errors

$$\epsilon_2 = \frac{7}{29} = 0.24$$

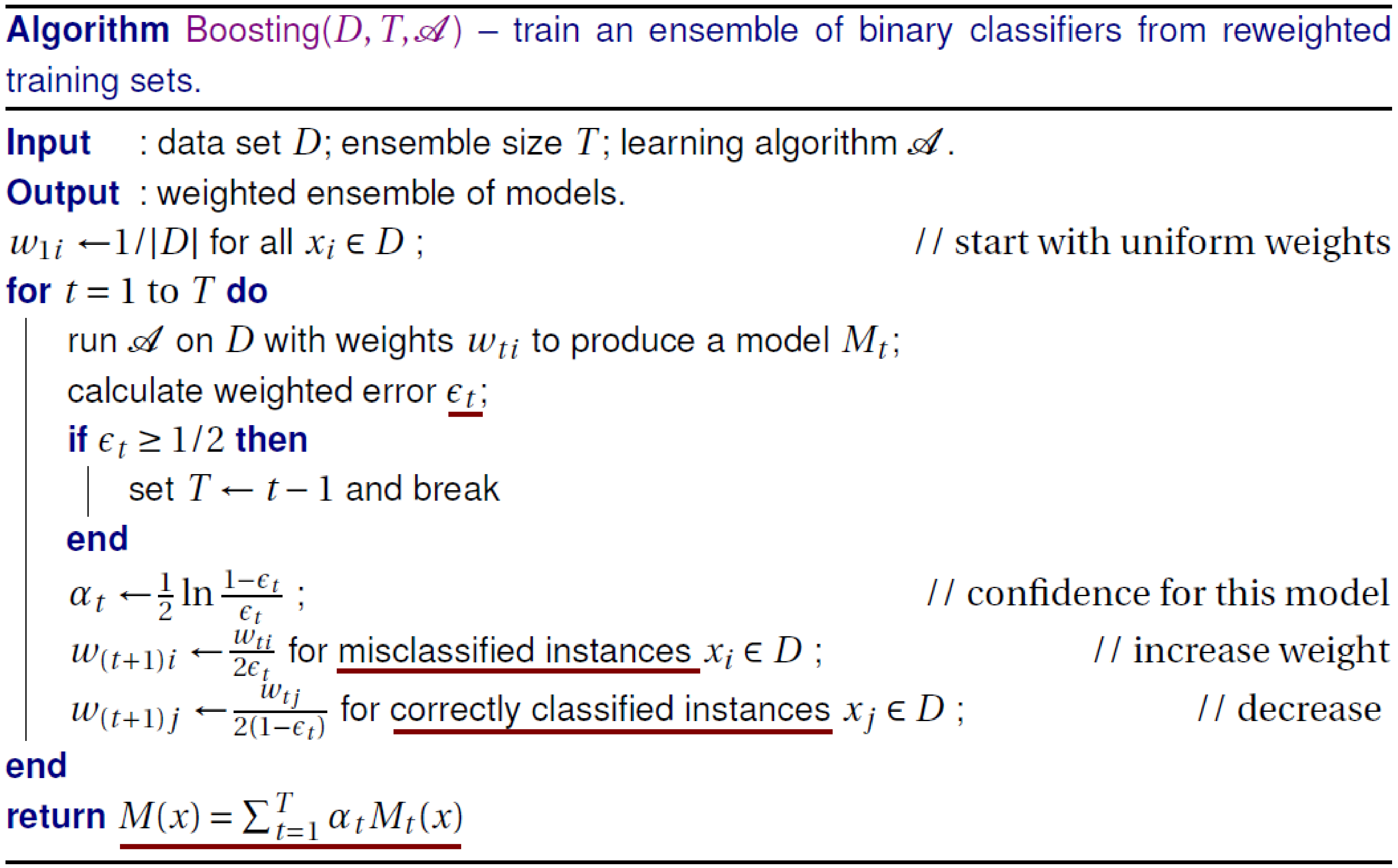
$$\alpha_2 = \frac{1}{2}\ln\frac{1-\epsilon_2}{\epsilon_2} = 0.58$$

$w' = \{\ 2.07w,\ 0.66w\ \}$

Continue for T iterations or until $\epsilon \geq 0.5$

Boosting

Computational complexity: $O(TDK)$
(ensemble size x data size x dimensionality)



Algorithm Boosting$(D, T, \mathscr{A})$ – train an ensemble of binary classifiers from reweighted training sets.

```
Input  : data set $D$; ensemble size $T$; learning algorithm $\mathscr{A}$.
Output : weighted ensemble of models.
$w_{1i} \leftarrow 1/|D|$ for all $x_i \in D$ ;                // start with uniform weights
for $t = 1$ to $T$ do
    run $\mathscr{A}$ on $D$ with weights $w_{ti}$ to produce a model $M_t$;
    calculate weighted error $\epsilon_t$;
    if $\epsilon_t \geq 1/2$ then
        set $T \leftarrow t - 1$ and break
    end
    $\alpha_t \leftarrow \frac{1}{2} \ln \frac{1-\epsilon_t}{\epsilon_t}$ ;          // confidence for this model
    $w_{(t+1)i} \leftarrow \frac{w_{ti}}{2\epsilon_t}$ for misclassified instances $x_i \in D$ ;      // increase weight
    $w_{(t+1)j} \leftarrow \frac{w_{tj}}{2(1-\epsilon_t)}$ for correctly classified instances $x_j \in D$ ;   // decrease
end
return $M(x) = \sum_{t=1}^{T} \alpha_t M_t(x)$
```

How to measure performance

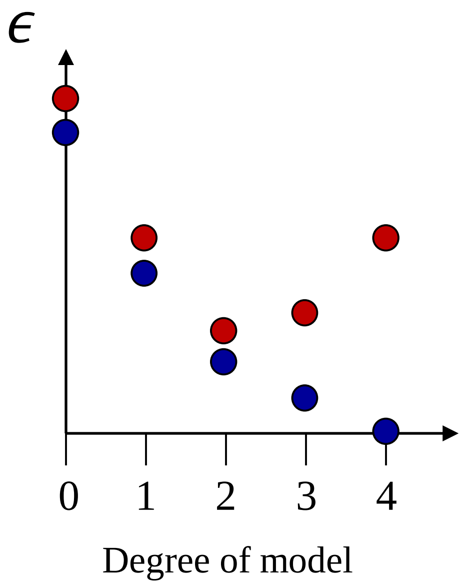
- So we decide on a performance measure (e.g., accuracy) and we evaluate it on test data, resulting in *acc*
  - How confident are we that our classifier really has an accuracy of *acc*?
  - I.e., if we repeated it on (different sets of) representative data an infinite number of times, what would the variance be?
    - We don't know, but we can estimate it by assuming certain things about the true distribution of the variable *acc*
- If we can estimate the performance measure *k* times, we reduce the sample variance by a factor of $^{1}/_{\sqrt{k}}$
- This is typically done in the experimental process of *cross-validation*

Cross-validation

- The cross-validation process for experimental evaluation:
  - Randomly partition the experimental data into *k* parts ("folds")
  - Train the model on *k*–1 folds
  - Evaluate it on the remaining test fold
  - Repeat for a total of *k* times, evaluating on each test fold
  - Average the performance measure (e.g., accuracy), over the *k* trials
- This is *k*-fold cross-validation
  - Typically, $k = 10$, but other values are also used
    - $k = 2$ is quite common
  - Rule of thumb: folds should contain at last 30 instances
    - Implying $\geq$300 instances in the data set for $k = 10$
  - If $k = n$ (the number of data points), this is leave-one-out cross-validation (a.k.a. the jackknife)

Typical training/validation/testing example

- For a 1D regression problem, let's train different models (different regression functions):
  - 0-degree polynomial (a constant), 1-degree polynomial (a line), 2nd-degree polynomial, 3rd-degree polynomial, 4th-degree polynomial
- Errors on the training data:



Which is the best model?

Answer: We don't know!

Use the validation data to determine:

The second-degree polynomial has the lowest error on this validation data

Now run cross-validation on the test data to estimate the performance of your model

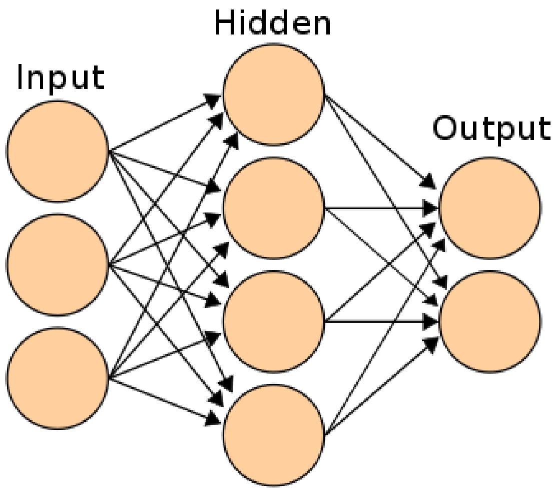
Produce the final model by using all your data

Ship it!

Artificial neural networks

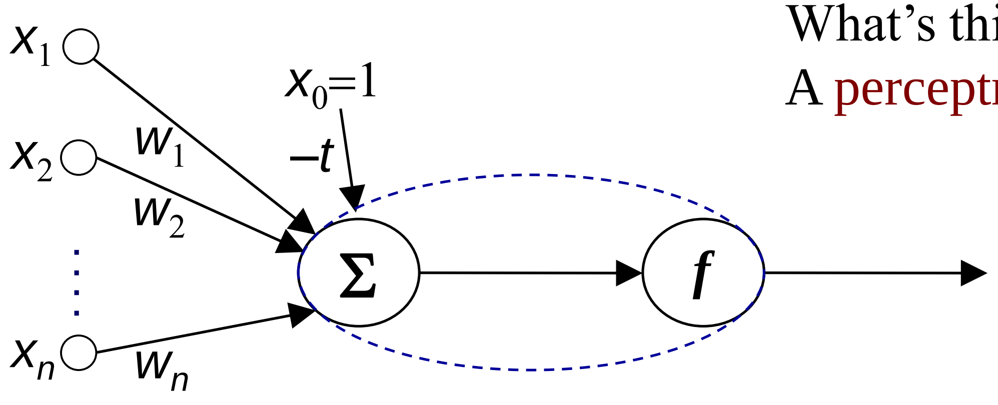
- Inspired by our understanding of the brain, an ANN is a set of simple processing elements (nodes or neurons) connected together to form a network that shares some properties with connected networks of neurons in the brain
- Neural networks typically have these characteristics:
  - Node connections with adaptive (learnable) weights
  - Can approximate nonlinear functions of the inputs
  - Highly parallel (conceptually – may be implemented serially)

A simple 2-layer neural network



The NN learning problem: from training data, learn the weights

A simple neural network



$$o = f\left(\sum_{i=0}^{n} w_i x_i\right)$$

What's this neural network called?
A perceptron! (I.e., a linear classifier)

Inputs	Weights	Sum of	(Threshold)
x	***w***	weighted inputs	function

$-t$: threshold value or bias

$$\left(\sum_{i=0}^{n} w_i x_i\right) = \left(\sum_{i=1}^{n} w_i x_i\right) - t = w^T x - t$$

Homogeneous　　Non-homogeneous

f : activation function – may be a thresholding unit (binary output):

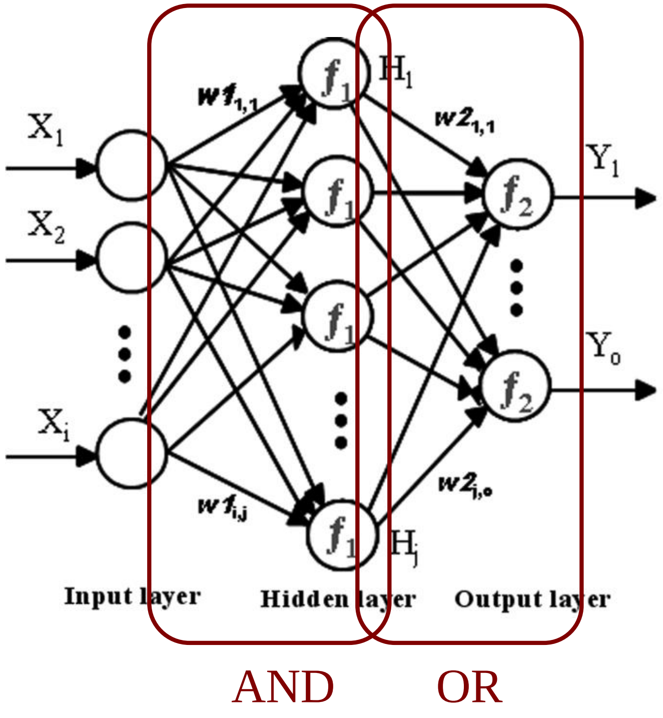
$$f(x) = \begin{cases} 1 & x > 0 \\ -1 & \text{otherwise} \end{cases}$$

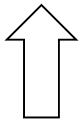
Implementing general Boolean functions

- Solution:
  - A network of perceptrons
  - Any Boolean function representable as disjunctive normal form (DNF)
    - 2 layers
    - Disjunction (layer 2) of conjunctions (layer 1)
- Example of XOR in DNF

  x_1 XOR $x_2 = (x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$

- Practical problem of representing high-dimensional functions





Feedforward network
(no cycles in the network)

As opposed to a **recurrent** network

Typical neural network learning

- The target function can be discrete-valued, real-valued, or a vector of several real- or discrete-valued attributes
- Training data: attribute-value pairs $(\boldsymbol{x}_i, y_i)$
  - E.g., for ALVINN, $\boldsymbol{x}_i$ is the input (30x32) image, y_i is the steering direction
- The training data may contain errors (i.e., noisy)
- Long training time, fast execution (evaluation) time
  - E.g., real-time steering response for ALVINN
- In training, use gradient descent to search the hypothesis space of possible weight vectors to find the $\boldsymbol{w}$ that best fits the training examples

Good final exam questions?

- The contingency table
  - Calculate FPR, TPR, accuracy, average recall, etc.
- Gram and scatter matrix
- Bias and variance of an ML model
- Multivariate linear regression eqn: $\boldsymbol{y} = \boldsymbol{X}\boldsymbol{w} + \boldsymbol{\epsilon}$
- Outliers
- Classifiers – assume linear separability in order to work?
- Original vs. dual perceptron models
- Classifier margin
- Soft-margin classifier
- Use of Kernel functions
- Clustering and intra/inter-class variance
- Distance metrics
- K-means clustering algorithm
- Bagging, boosting
- Cross-validation
- Basic NN questions

Questions?