Classification (1)

COMP9417 Machine Learning and Data Mining

Term 2, 2021

Acknowledgements

Material derived from slides for the book
"Elements of Statistical Learning (2nd Ed.)" by T. Hastie,
R. Tibshirani & J. Friedman. Springer (2009)
http://statweb.stanford.edu/~tibs/ElemStatLearn/

Material derived from slides for the book
"Machine Learning: A Probabilistic Perspective" by P. Murphy
MIT Press (2012)
http://www.cs.ubc.ca/~murphyk/MLbook

Material derived from slides for the book
"Machine Learning" by P. Flach
Cambridge University Press (2012)
http://cs.bris.ac.uk/~flach/mlbook

Material derived from slides for the book
"Bayesian Reasoning and Machine Learning" by D. Barber
Cambridge University Press (2012)
http://www.cs.ucl.ac.uk/staff/d.barber/brml

Material derived from slides for the book
"Machine Learning" by T. Mitchell
McGraw-Hill (1997)
http://www-2.cs.cmu.edu/~tom/mlbook.html

Material derived from slides for the course
"Machine Learning" by A. Srinivasan
BITS Pilani, Goa, India (2016)

Aims

This lecture will introduce you to machine learning approaches to the problem of *classification*. Following it you should be able to reproduce theoretical results, outline algorithmic techniques and describe practical applications for the topics:

- outline the probem of learning to classify
- outline a framework for solving machine learning problems
- describe issues of generalisation and evaluation for classification
- outline the use of a linear model as a 2-class classifier
- outline the Perceptron classification algorithm
- describe distance measures and how they are used in classification
- describe the k-nearest neighbour method for classification (and regression)

Introduction

Classification (sometimes called *concept learning*) methods dominate machine learning . . .

. . . however, they often don't have convenient mathematical properties like regression, so are more complicated to analyse. The idea is to learn a *classifier*, which is usually a function mapping from an input data point to one of a set of discrete outputs, i.e., the *classes*.

We will mostly focus on their advantages and disadvantages as learning methods first, and point to unifying ideas and approaches where applicable. In this and the next lecture we focus on classification methods that are essentially *linear models* . . .

and in later lectures we will see other, more expressive, classifier learning methods.

Assassinating spam e-mail

SpamAssassin is a widely used open-source spam filter. It calculates a score for an incoming e-mail, based on a number of built-in rules or 'tests' in SpamAssassin's terminology, and adds a 'junk' flag and a summary report to the e-mail's headers if the score is 5 or more.

```
-0.1 RCVD_IN_MXRATE_WL       RBL: MXRate recommends allowing
                             [123.45.6.789 listed in sub.mxrate.net]
 0.6 HTML_IMAGE_RATIO_02     BODY: HTML has a low ratio of text to image area
 1.2 TVD_FW_GRAPHIC_NAME_MID BODY: TVD_FW_GRAPHIC_NAME_MID
 0.0 HTML_MESSAGE            BODY: HTML included in message
 0.6 HTML_FONx_FACE_BAD      BODY: HTML font face is not a word
 1.4 SARE_GIF_ATTACH         FULL: Email has a inline gif
 0.1 BOUNCE_MESSAGE          MTA bounce message
 0.1 ANY_BOUNCE_MESSAGE      Message is some kind of bounce message
 1.4 AWL                     AWL: From: address is in the auto white-list
```

From left to right you see the score attached to a particular test, the test identifier, and a short description including a reference to the relevant part of the e-mail. As you see, scores for individual tests can be negative (indicating evidence suggesting the e-mail is ham rather than spam) as well as positive. The overall score of 5.3 suggests the e-mail might be spam.

Linear classification

Suppose we have only two tests and four training e-mails, one of which is spam. Both tests succeed for the spam e-mail; for one ham e-mail neither test succeeds, for another the first test succeeds and the second doesn't, and for the third ham e-mail the first test fails and the second succeeds.

It is easy to see that assigning both tests a weight of 4 correctly 'classifies' these four e-mails into spam and ham. In the mathematical notation introduced above we could describe this classifier as $4x_1 + 4x_2 > 5$ or $(4, 4) \cdot (x_1, x_2) > 5$.

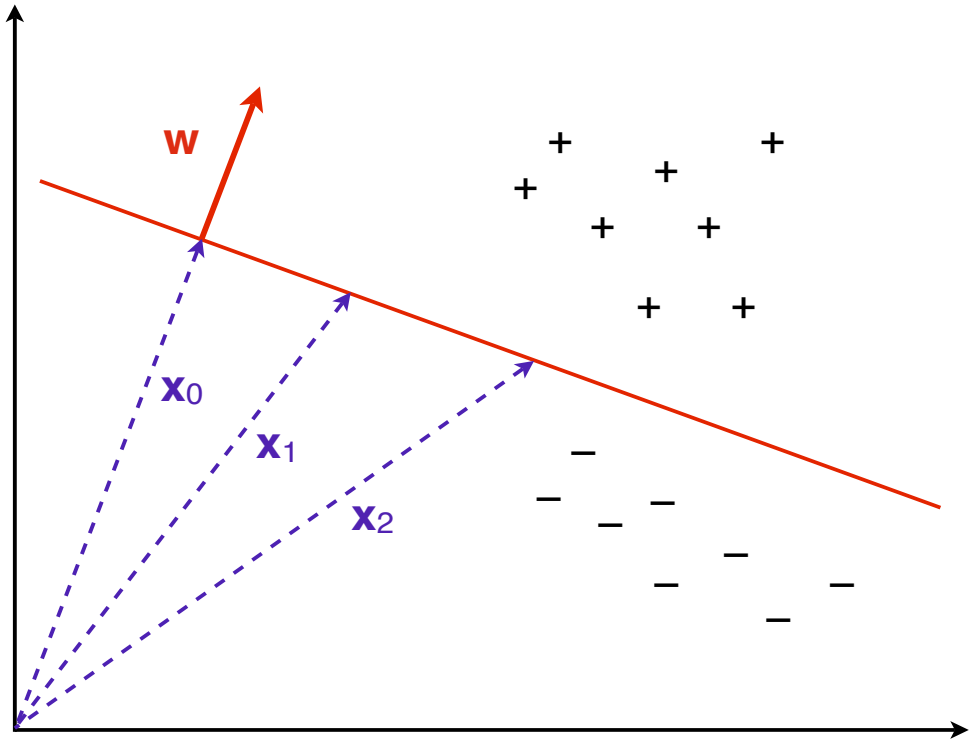
In fact, any weight between 2.5 and 5 will ensure that the threshold of 5 is only exceeded when both tests succeed. We could even consider assigning different weights to the tests – as long as each weight is less than 5 and their sum exceeds 5 – although it is hard to see how this could be justified by the training data.

Spam filtering as a classification task

The columns marked x_1 and x_2 indicate the results of two tests on four different e-mails. The fourth column indicates which of the e-mails are spam. The right-most column demonstrates that by thresholding the function $4x_1 + 4x_2$ at 5, we can separate spam from ham.

E-mail	x_1	x_2	Spam?	$4x_1 + 4x_2$
1	1	1	1	8
2	0	0	0	0
3	1	0	0	4
4	0	1	0	4

Linear classification in two dimensions



- straight line separates positives from negatives (linear "discriminant")
- defined by $\mathbf{w} \cdot \mathbf{x}_i = t$
- $\mathbf{w}$ is perpendicular to decision boundary
- $\mathbf{w}$ points in direction of positives
- t is the decision threshold

Linear classification in two dimensions

Note: $\mathbf{x}_i$ points to a point on the decision boundary.

In particular, $\mathbf{x}_0$ points in the same direction as $\mathbf{w}$,

from which it follows that $\mathbf{w} \cdot \mathbf{x}_0 = ||\mathbf{w}||\ ||\mathbf{x}_0|| = t$ (where $||\mathbf{x}||$ denotes the length of the vector $\mathbf{x}$).

Homogeneous coordinates

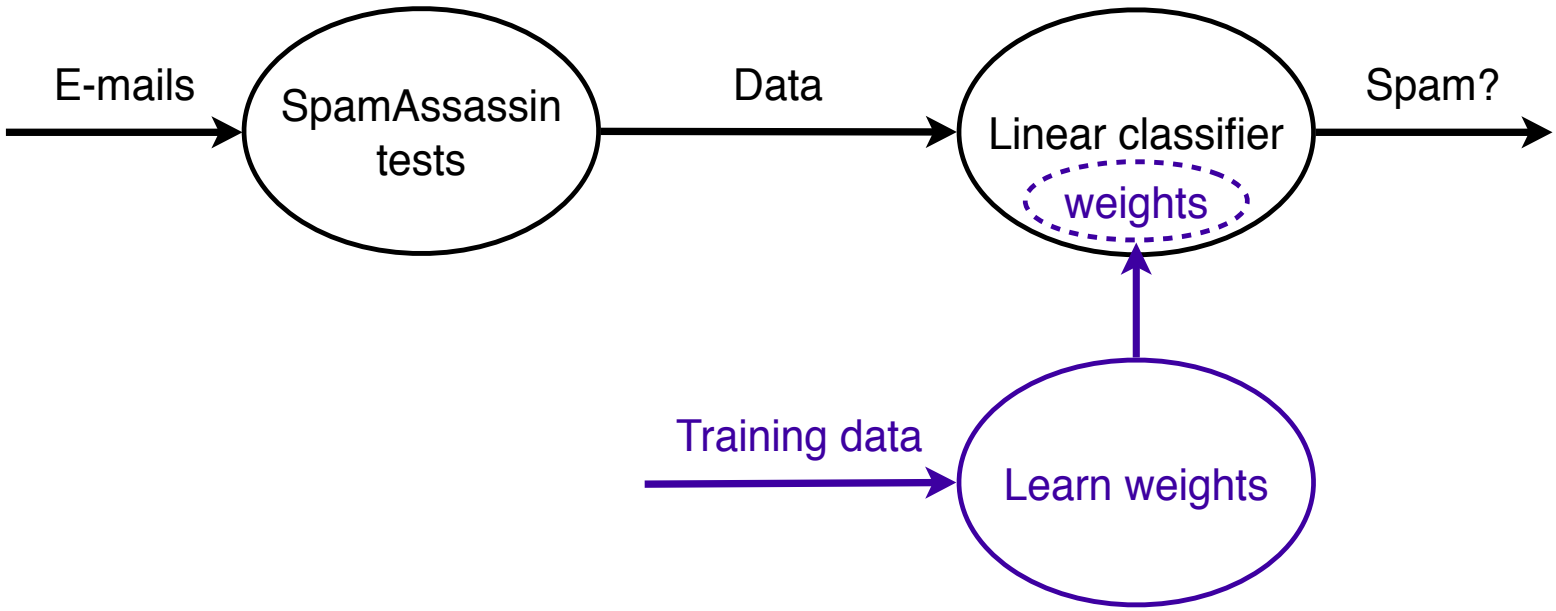
It is sometimes convenient to simplify notation further by introducing an extra constant 'variable' $x_0 = 1$, the weight of which is fixed to $w_0 = -t$.

The extended data point is then $\mathbf{x}^\circ = (1, x_1, \ldots, x_n)$ and the extended weight vector is $\mathbf{w}^\circ = (-t, w_1, \ldots, w_n)$, leading to the decision rule $\mathbf{w}^\circ \cdot \mathbf{x}^\circ > 0$ and the decision boundary $\mathbf{w}^\circ \cdot \mathbf{x}^\circ = 0$.

Thanks to these so-called *homogeneous coordinates* the decision boundary passes through the origin of the extended coordinate system, at the expense of needing an additional dimension.

Note: this doesn't really affect the data, as all data points and the 'real' decision boundary live in the plane $x_0 = 1$.

Machine learning for spam filtering



At the top we see how SpamAssassin approaches the spam e-mail classification task: the text of each e-mail is converted into a data point by means of SpamAssassin's built-in tests, and a *linear classifier* is applied to obtain a 'spam or ham' decision. At the bottom (in blue) we see the bit that is done by machine learning.

Example: a Bayesian classifier I

Bayesian spam filters maintain a *vocabulary* of words and phrases – potential spam or ham indicators – for which statistics are collected from a *training set*.

- For instance, suppose that the word 'Viagra' occurred in four spam e-mails and in one ham e-mail. If we then encounter a new e-mail that contains the word 'Viagra', we might reason that the odds that this e-mail is spam are 4:1, or the probability of it being spam is 0.80 and the probability of it being ham is 0.20.
- The situation is slightly more subtle because we have to take into account the prevalence of spam. Suppose that I receive on average one spam e-mail for every six ham e-mails. This means that I would estimate the odds of an unseen e-mail being spam as 1:6, i.e., non-negligible but not very high either.

Example: a Bayesian classifier II

- If I then learn that the e-mail contains the word 'Viagra', which occurs four times as often in spam as in ham, I need to combine these two odds. As we shall see later, Bayes' rule tells us that we should simply multiply them: 1:6 times 4:1 is 4:6, corresponding to a spam probability of 0.4.

In this way you are combining two independent pieces of evidence, one concerning the prevalence of spam, and the other concerning the occurrence of the word 'Viagra', pulling in opposite directions.

Example: a Bayesian classifier III

The nice thing about this 'Bayesian' classification scheme is that it can be repeated if you have further evidence. For instance, suppose that the odds in favour of spam associated with the phrase 'blue pill' is estimated at 3:1, and suppose our e-mail contains both 'Viagra' and 'blue pill', then the combined odds are 4:1 times 3:1 is 12:1, which is ample to outweigh the 1:6 odds associated with the low prevalence of spam (total odds are 2:1, or a spam probability of 0.67, up from 0.40 without the 'blue pill').

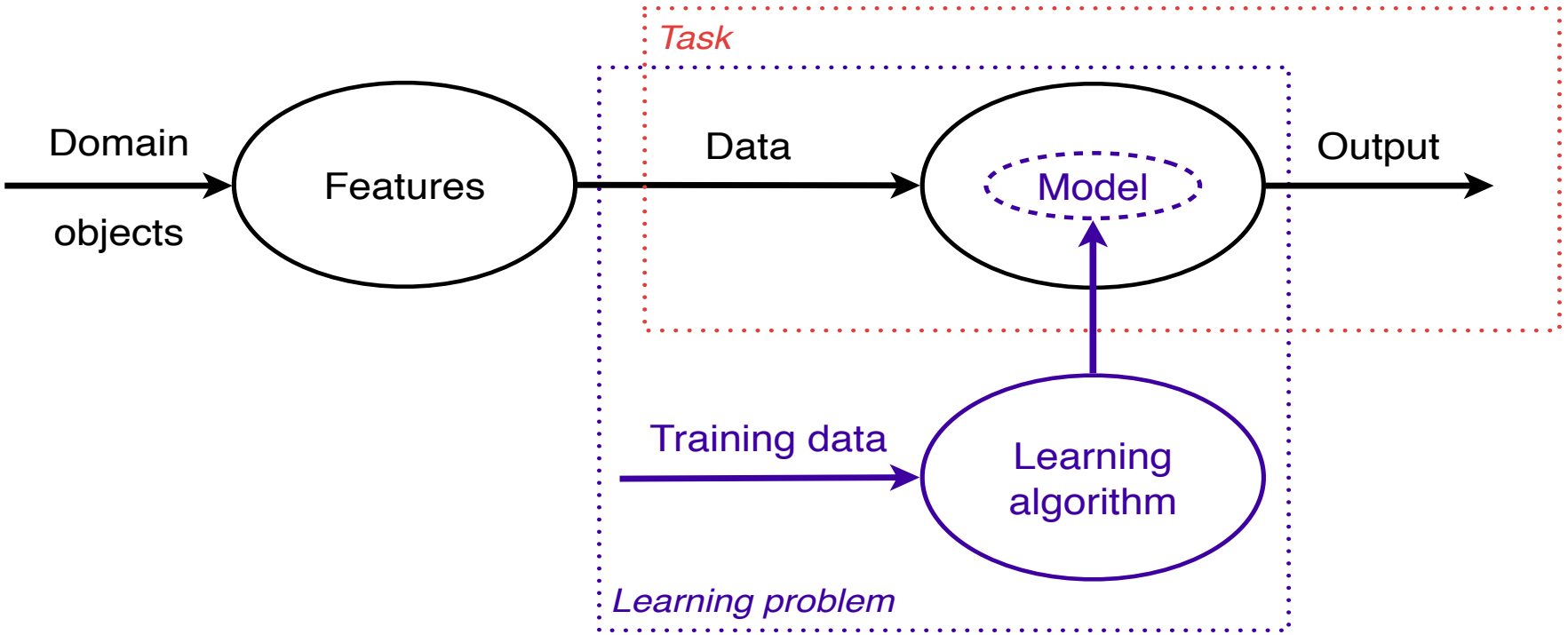
Example: a rule-based classifier

- if the e-mail contains the word 'Viagra' then estimate the odds of spam as 4:1;
- otherwise, if it contains the phrase 'blue pill' then estimate the odds of spam as 3:1;
- otherwise, estimate the odds of spam as 1:6.

The first rule covers all e-mails containing the word 'Viagra', regardless of whether they contain the phrase 'blue pill', so no overcounting occurs. The second rule *only* covers e-mails containing the phrase 'blue pill' but not the word 'Viagra', by virtue of the 'otherwise' clause. The third rule covers all remaining e-mails: those which neither contain neither 'Viagra' nor 'blue pill'.

Step back: some ingredients of machine learning

How machine learning helps to solve a task



An overview of how machine learning is used to address a given task. A task (red box) requires an appropriate mapping – a model – from data described by features to outputs. Obtaining such a mapping from training data is what constitutes a learning problem (blue box).

Some terminology I

Tasks are addressed by models, whereas learning problems are solved by learning algorithms that produce models.

Some terminology II

Machine learning is concerned with using the right features to build the right models that achieve the right tasks.

Some terminology III

Models lend the machine learning field diversity, but tasks and features give it unity.

Some terminology IV

Does the algorithm require all training data to be present before the start of learning ? If yes, then it is categorised as **batch learning** algorithm.

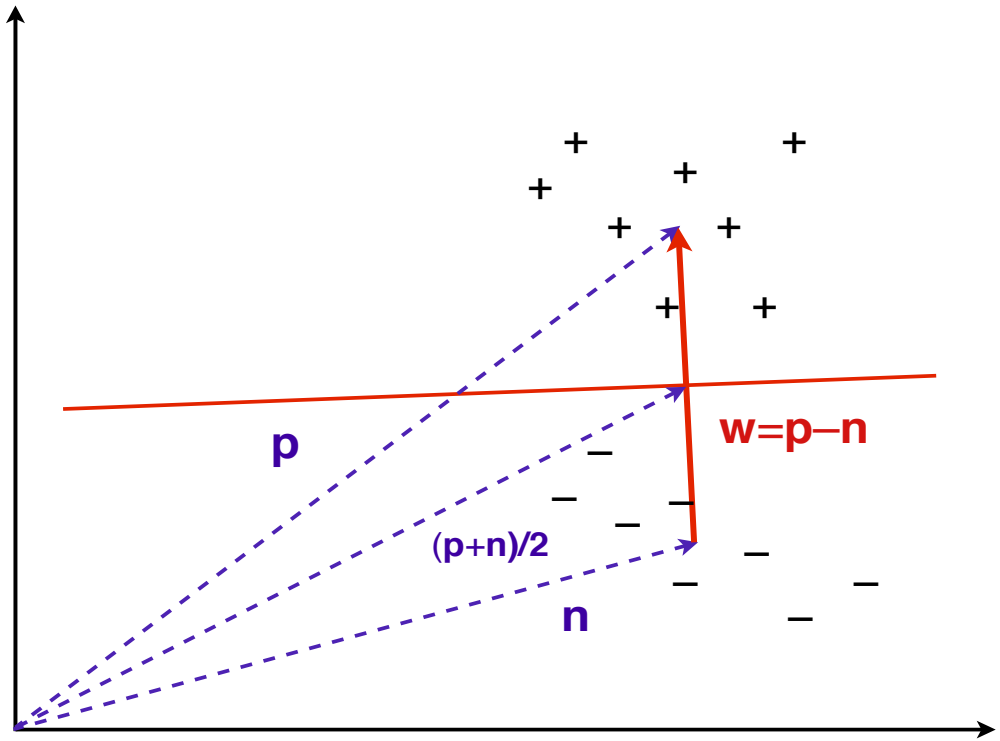
If however, it can continue to learn a new data arrives, it is an **online learning** algorithm.

Some terminology V

If the model has a fixed number of parameters it is categorised as **parametric**.

Otherwise, if the number of parameters grows with the amount of training data it is categorised as **non-parametric**.

Example: basic linear classifier I



The basic linear classifier constructs a decision boundary by half-way intersecting the line between the positive and negative centres of mass.

Example: basic linear classifier II

The basic linear classifier is described by the equation $\mathbf{w} \cdot \mathbf{x} = t$, with $\mathbf{w} = \mathbf{p} - \mathbf{n}$; the decision threshold can be found by noting that $(\mathbf{p} + \mathbf{n})/2$ is on the decision boundary, and hence
$t = (\mathbf{p} - \mathbf{n}) \cdot (\mathbf{p} + \mathbf{n})/2 = (||\mathbf{p}||^2 - ||\mathbf{n}||^2)/2$, where $||\mathbf{x}||$ denotes the length of vector $\mathbf{x}$.

The philosophical problem

Recall: **Deduction:** derive specific consequences from general theories

Induction: derive general theories from specific observations

Deduction is well-founded (mathematical logic).

Induction is (philosophically **and** practically) problematic – induction is useful since it often seems to work – an inductive argument !

Generalisation - the key objective of machine learning

What we are really interested in is *generalising* from the sample of data in our training set. This requires an assumption that can be stated as (last lecture):

The inductive learning hypothesis

> *Any hypothesis found to approximate the target (true) function well over a sufficiently large set of training examples will also approximate the target function well over other unobserved examples.*[1].

[1] T. Mitchell (1997) "Machine Learning"

Generalisation - the key objective of machine learning

A corollary of the *inductive learning hypothesis* is that it is necessary to make some assumptions about the type of target function in a task for an algorithm to go beyond the data, i.e., generalise or learn.

The set of assumptions required for a machine learning algorithm to be able to generalise is known in as the **inductive bias** of the algorithm.

We'll see that this idea of inductive bias is often used when comparing algorithms, and can be formalised in some cases (later lecture).

Evaluating classification – contingency table I

For the two-class prediction case:

Actual Class	Predicted Class: Yes	Predicted Class: No
Yes	True Positive (TP)	False Negative (FN)
No	False Positive (FP)	True Negative (TN)

Evaluating classification – contingency table II

Classification Accuracy on a sample of labelled pairs $(x, c(x))$ given a learned classification model that predicts, for each instance x, a class value $\hat{c}(x)$:

$$\text{acc} = \frac{1}{|\text{Test}|} \sum_{x \in \text{Test}} I[\hat{c}(x) = c(x)]$$

where Test is a test set and $I[]$ is the indicator function which is 1 iff its argument evaluates to true, and 0 otherwise.

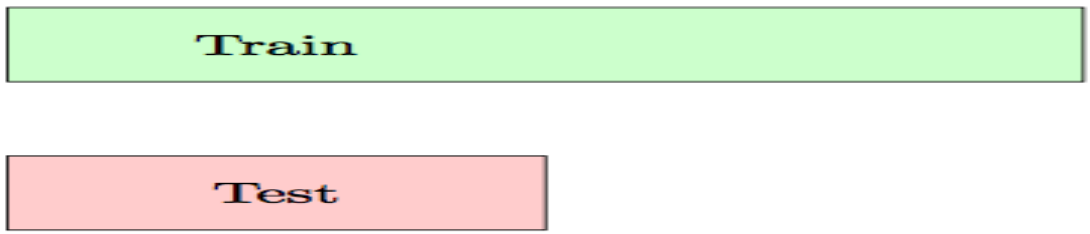
Classification Error is 1-acc.

Cross-validation I

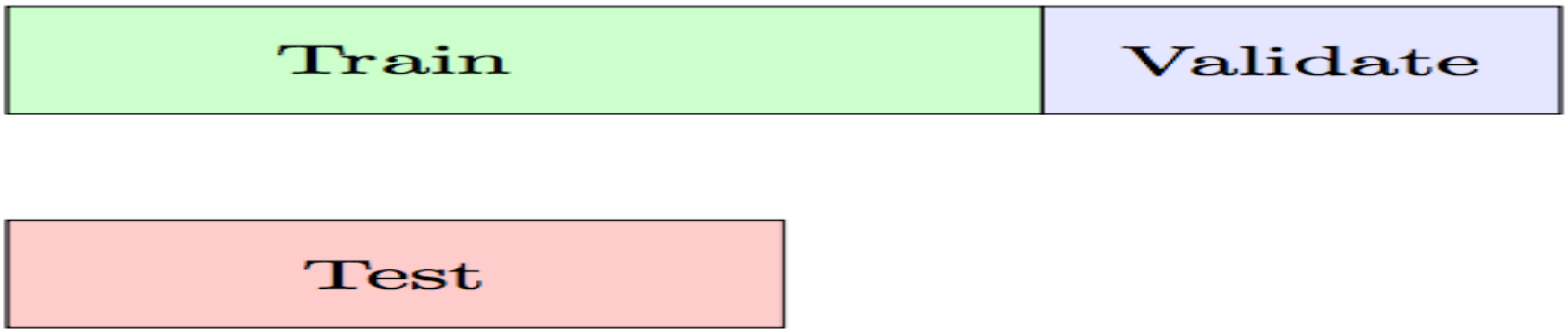
There are certain parameters that need to be estimated during learning.

We use the data, but NOT the training set, OR the test set. Instead, we use a separate *validation* or *development* set.

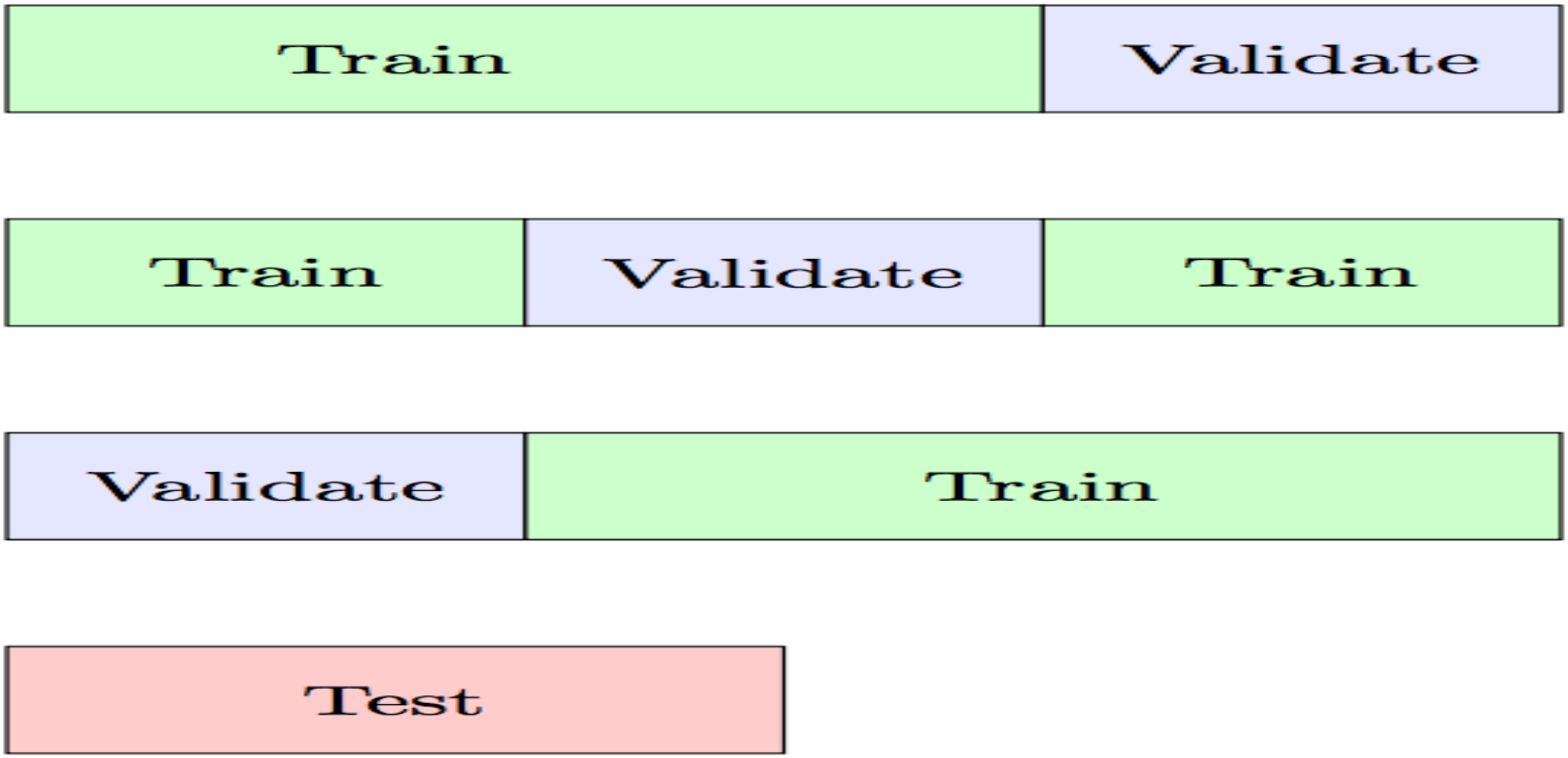
Cross-validation II



Cross-validation III



Cross-validation IV

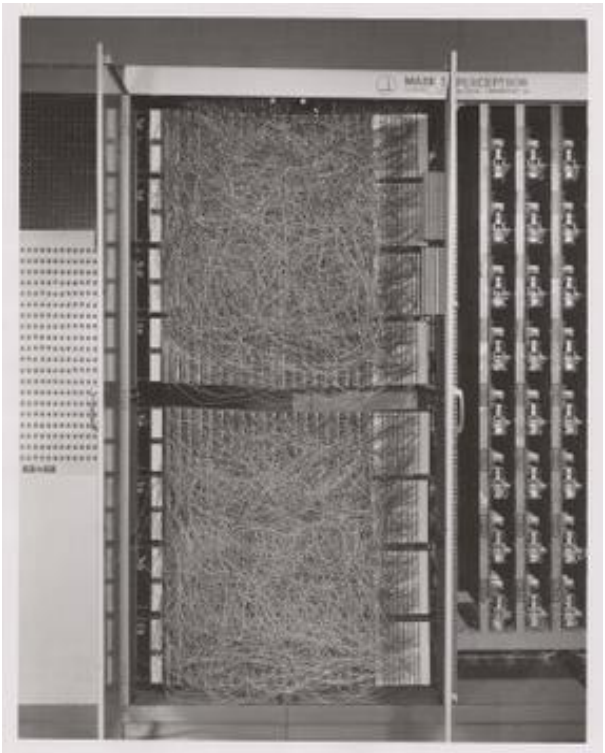


Perceptron

A linear classifier that can achieve perfect separation on linearly separable data is the *perceptron*, originally proposed as a simple *neural network* by F. Rosenblatt in the late 1950s.

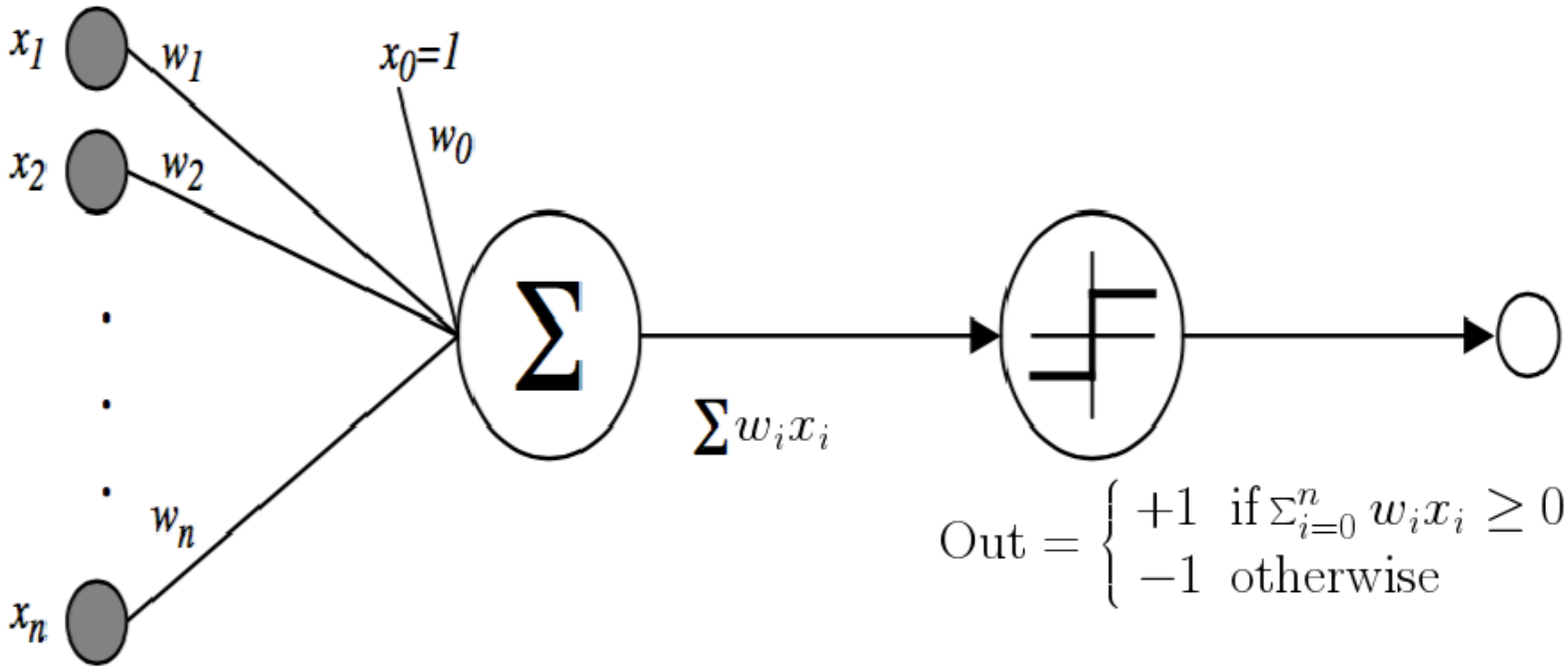
Perceptron

Originally implemented in software (based on the McCulloch-Pitts neuron from the 1940s), then in hardware as a 20x20 visual sensor array with potentiometers for adaptive weights.



Source `http://en.wikipedia.org/w/index.php?curid=47541432`

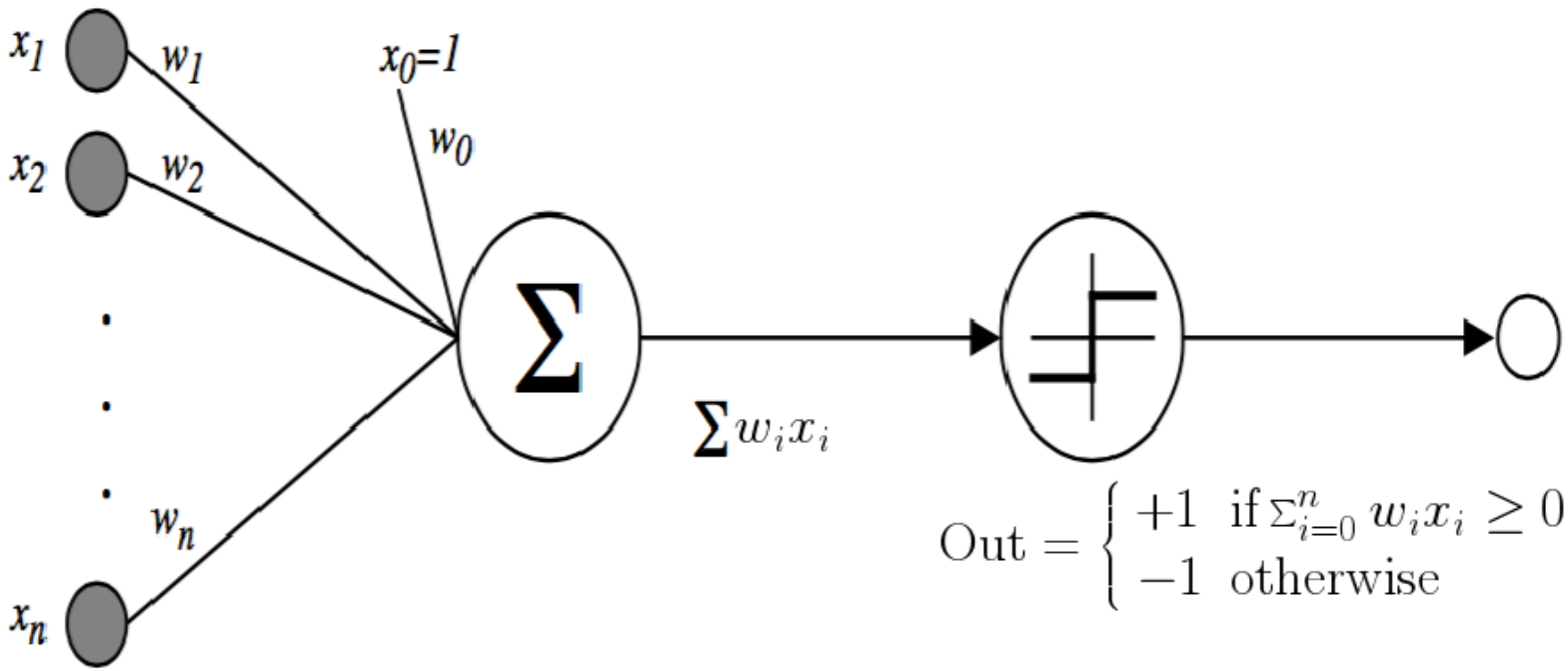
Perceptron



Output o is thresholded sum of products of inputs and their weights:

$$o(x_1, \ldots, x_n) = \begin{cases} +1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

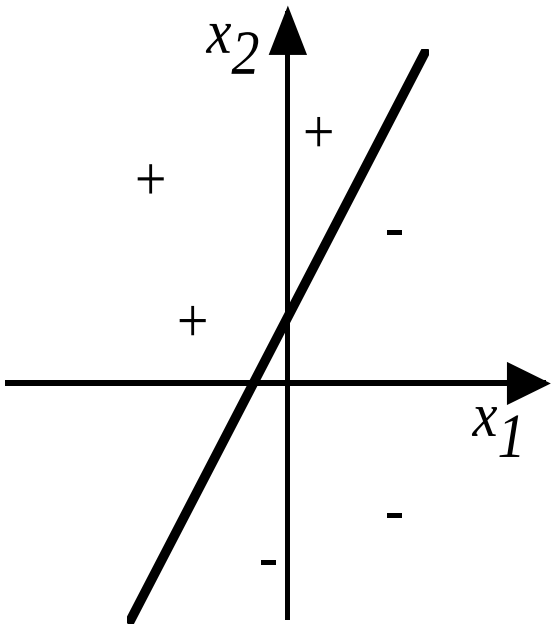
Perceptron



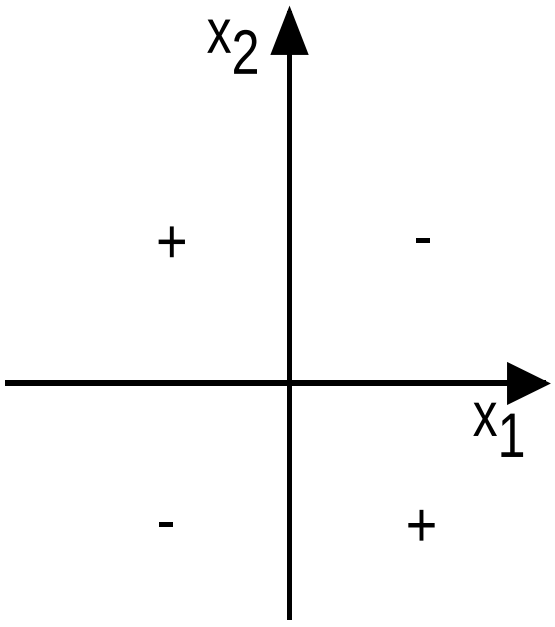
Or in vector notation:

$$o(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Decision Surface of a Perceptron



(a)



(b)

Represents some useful functions

- What weights represent $o(x_1, x_2) = AND(x_1, x_2)$?
- What weights represent $o(x_1, x_2) = XOR(x_1, x_2)$?

Decision Surface of a Perceptron

So some functions not representable

- e.g., not linearly separable
  - a labelled data set is linearly separable if there is a linear decision boundary that separates the classes
- for non-linearly separable data we'll need something else
  - e.g., networks of these ...
  - the start of "deep" networks ...

Perceptron learning

Key idea:

Learning is "finding a good set of weights"

Perceptron learning is simply an iterative weight-update scheme:

$$w_i \leftarrow w_i + \Delta w_i$$

where the weight update Δw_i depends only on *misclassified* examples and is modulated by a "smoothing" parameter η typically referred to as the "learning rate".

Can prove that perceptron learning will converge:

- if training data is linearly separable
- and η sufficiently small

Perceptron learning

The perceptron iterates over the training set, updating the weight vector every time it encounters an incorrectly classified example.

- For example, let $\mathbf{x}_i$ be a misclassified positive example, then we have $y_i = +1$ and $\mathbf{w} \cdot \mathbf{x}_i < t$. We therefore want to find $\mathbf{w}'$ such that $\mathbf{w}' \cdot \mathbf{x}_i > \mathbf{w} \cdot \mathbf{x}_i$, which moves the decision boundary towards and hopefully past x_i.
- This can be achieved by calculating the new weight vector as $\mathbf{w}' = \mathbf{w} + \eta \mathbf{x}_i$, where $0 < \eta \leq 1$ is the *learning rate* (again, assume set to 1). We then have $\mathbf{w}' \cdot \mathbf{x}_i = \mathbf{w} \cdot \mathbf{x}_i + \eta \mathbf{x}_i \cdot \mathbf{x}_i > \mathbf{w} \cdot \mathbf{x}_i$ as required.
- Similarly, if $\mathbf{x}_j$ is a misclassified negative example, then we have $y_j = -1$ and $\mathbf{w} \cdot \mathbf{x}_j > t$. In this case we calculate the new weight vector as $\mathbf{w}' = \mathbf{w} - \eta \mathbf{x}_j$, and thus $\mathbf{w}' \cdot \mathbf{x}_j = \mathbf{w} \cdot \mathbf{x}_j - \eta \mathbf{x}_j \cdot \mathbf{x}_j < \mathbf{w} \cdot \mathbf{x}_j$.

Perceptron learning

- The two cases can be combined in a single update rule:

$$\mathbf{w}' = \mathbf{w} + \eta y_i \mathbf{x}_i$$

- Here y_i acts to change the sign of the update, corresponding to whether a positive or negative example was misclassified
- This is the basis of the *perceptron training algorithm* for linear classification
- The algorithm just iterates over the training examples applying the weight update rule until all the examples are correctly classified
- If there is a linear model that separates the positive from the negative examples, i.e., the data is linearly separable, it can be shown that the perceptron training algorithm will converge in a finite number of steps.

Perceptron training algorithm

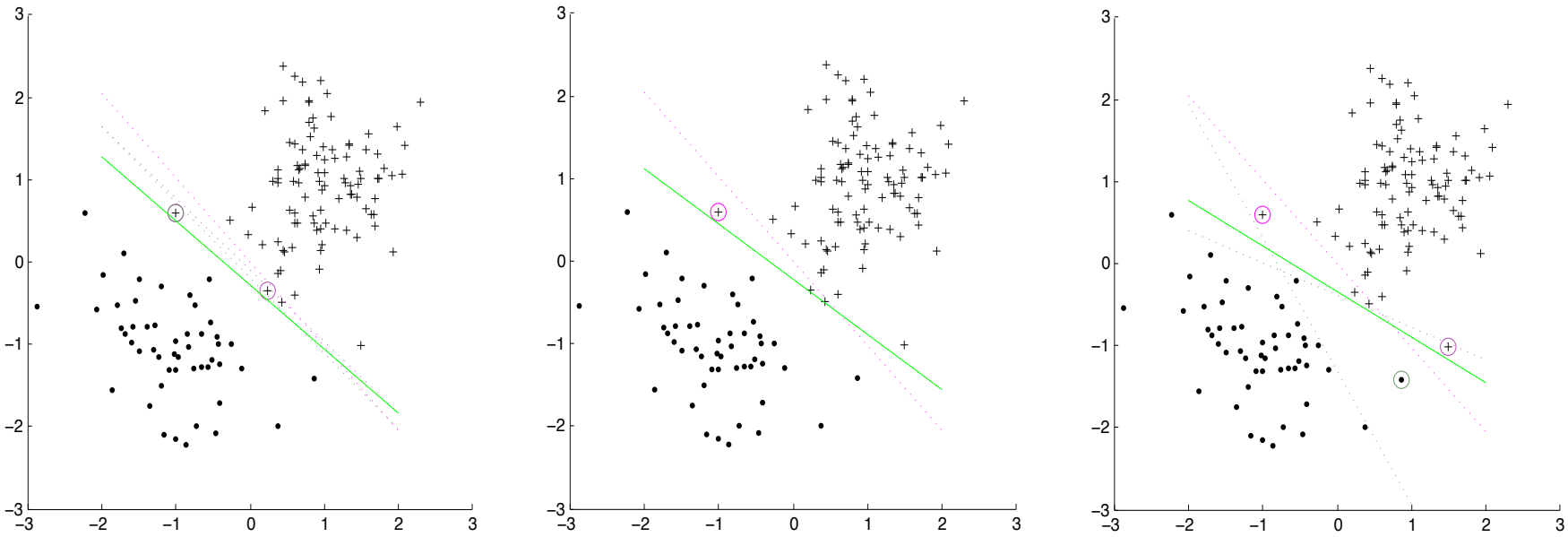
Algorithm Perceptron(D, η) // perceptron training for linear classification

Input: labelled training data D in homogeneous coordinates; learning rate η.

Output: weight vector $\mathbf{w}$ defining classifier $\hat{y} = \text{sign}(\mathbf{w} \cdot \mathbf{x})$.

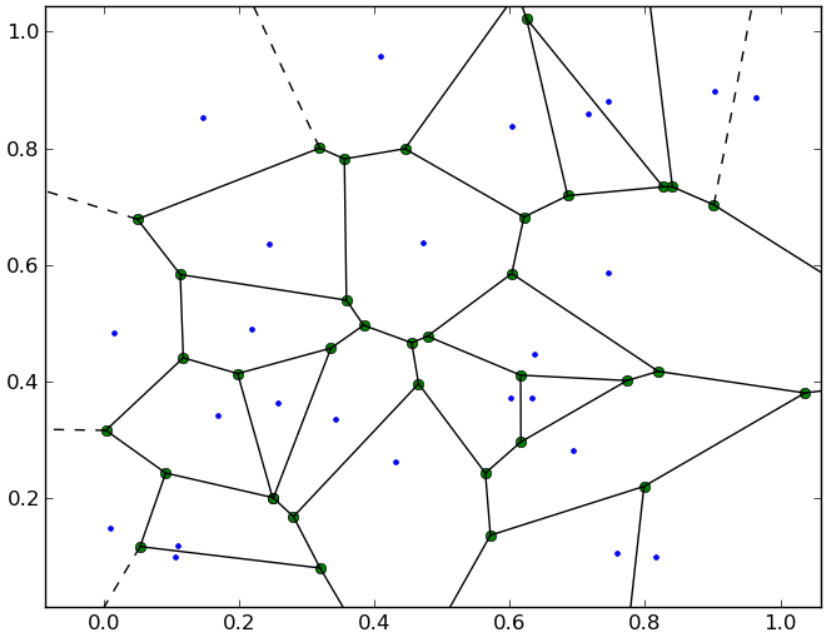
$\mathbf{w} \leftarrow \mathbf{0}$ // Other initialisations of the weight vector are possible
converged$\leftarrow$false
while *converged* $=$ false **do**
    converged$\leftarrow$true
    for $i = 1$ to $|D|$ **do**
        if $y_i \mathbf{w} \cdot \mathbf{x}_i \leq 0$ **then** // i.e., $\hat{y}_i \neq y_i$
            $\mathbf{w} \leftarrow \mathbf{w} + \eta y_i \mathbf{x}_i$
            converged$\leftarrow$false // We changed $\mathbf{w}$ so haven't converged yet
        end
    end
end

Perceptron training – varying learning rate



(left) A perceptron trained with a small learning rate ($\eta = 0.2$). The circled examples are the ones that trigger the weight update. (middle) Increasing the learning rate to $\eta = 0.5$ leads in this case to a rapid convergence. (right) Increasing the learning rate further to $\eta = 1$ may lead to too aggressive weight updating, which harms convergence.
The starting point in all three cases was the basic linear classifier.

Nearest Neighbour



Nearest Neighbour is a classification or regression algorithm that predicts whatever is the output value of the nearest data point to some query point.

Minkowski distance

Minkowski distance If $\mathcal{X} = \mathbb{R}^d$, the *Minkowski distance* of order $p > 0$ is defined as

$$\text{Dis}_p(\mathbf{x}, \mathbf{y}) = \left(\sum_{j=1}^{d} |x_j - y_j|^p \right)^{1/p} = ||\mathbf{x} - \mathbf{y}||_p$$

where $||\mathbf{z}||_p = \left(\sum_{j=1}^{d} |z_j|^p \right)^{1/p}$ is the p-*norm* (sometimes denoted L_p norm) of the vector $\mathbf{z}$.

Note: sometimes p is omitted when writing the norm, often when $p = 2$.

Minkowski distance

- The 2-norm refers to the familiar *Euclidean distance*

$$\text{Dis}_2(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{j=1}^{d} (x_j - y_j)^2} = \sqrt{(\mathbf{x} - \mathbf{y})^{\mathrm{T}} (\mathbf{x} - \mathbf{y})}$$

  which measures distance 'as the crow flies'.

- The 1-*norm* denotes *Manhattan distance*, also called *cityblock distance*:

$$\text{Dis}_1(\mathbf{x}, \mathbf{y}) = \sum_{j=1}^{d} |x_j - y_j|$$

  This is the distance if we can only travel along coordinate axes.

Minkowski distance

- If we now let p grow larger, the distance will be more and more dominated by the largest coordinate-wise distance, from which we can infer that $\text{Dis}_\infty(\mathbf{x}, \mathbf{y}) = \max_j |x_j - y_j|$; this is also called *Chebyshev distance*.
- You will sometimes see references to the 0-*norm* (or L_0 norm) which counts the number of non-zero elements in a vector. The corresponding distance then counts the number of positions in which vectors $\mathbf{x}$ and $\mathbf{y}$ differ. This is not strictly a Minkowski distance; however, we can define it as

$$\text{Dis}_0(\mathbf{x}, \mathbf{y}) = \sum_{j=1}^{d} (x_j - y_j)^0 = \sum_{j=1}^{d} I[x_j = y_j]$$

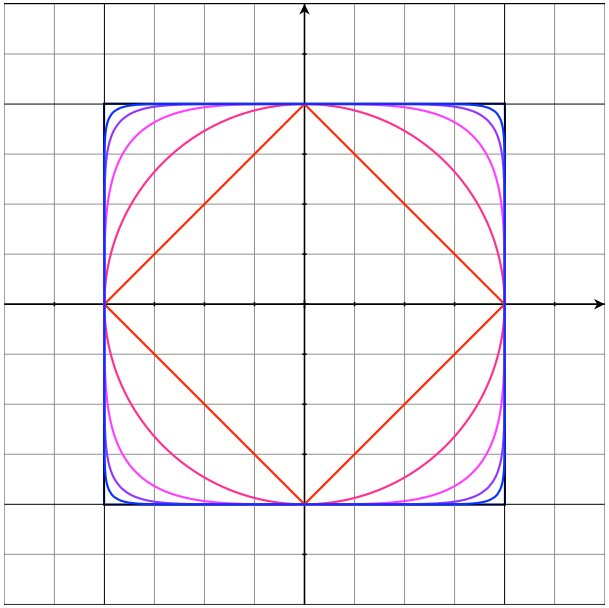
under the understanding that $x^0 = 0$ for $x = 0$ and 1 otherwise.

Minkowski distance

Sometimes the data $\mathcal{X}$ is not naturally in $\mathbb{R}^d$, but if we can turn it into Boolean features, or character sequences, we can still apply distance measures. For example:

- If $\mathbf{x}$ and $\mathbf{y}$ are binary strings, this is also called the *Hamming distance*. Alternatively, we can see the Hamming distance as the number of bits that need to be flipped to change $\mathbf{x}$ into $\mathbf{y}$.
- For non-binary strings of unequal length this can be generalised to the notion of *edit distance* or *Levenshtein distance*.

Circles and ellipses



Lines connecting points at order-p Minkowski distance 1 from the origin for (from inside) $p = 0.8$; $p = 1$ (Manhattan distance, the rotated square in red); $p = 1.5$; $p = 2$ (Euclidean distance, the violet circle); $p = 4$; $p = 8$; and $p = \infty$ (Chebyshev distance, the blue rectangle). Notice that for points on the coordinate axes all distances agree. For the other points, our reach increases with p; however, if we require a rotation-invariant distance metric then Euclidean distance is our only choice.

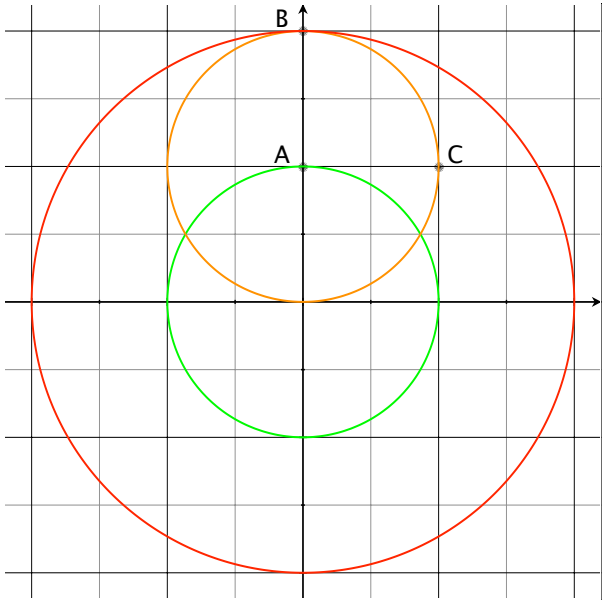
Distance metric

Distance metric Given an instance space $\mathcal{X}$, a *distance metric* is a function $\mathrm{Dis} : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ such that for any $x, y, z \in \mathcal{X}$:

1. distances between a point and itself are zero: $\mathrm{Dis}(x, x) = 0$;
2. all other distances are larger than zero: if $x \neq y$ then $\mathrm{Dis}(x, y) > 0$;
3. distances are symmetric: $\mathrm{Dis}(y, x) = \mathrm{Dis}(x, y)$;
4. detours can not shorten the distance: $\mathrm{Dis}(x, z) \leq \mathrm{Dis}(x, y) + \mathrm{Dis}(y, z)$.

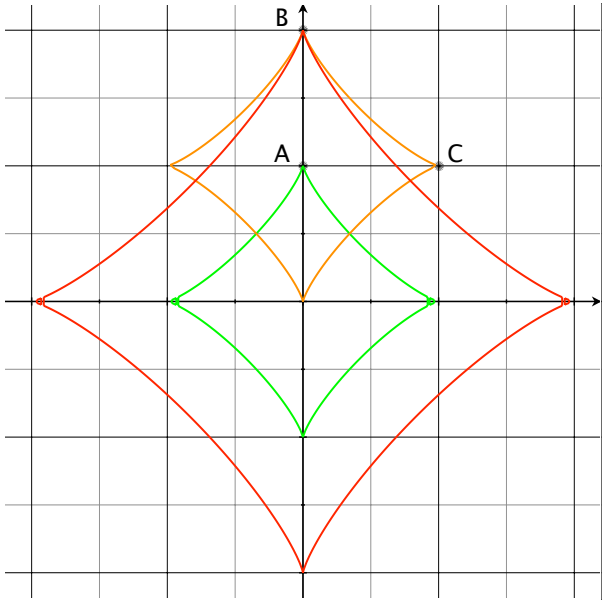
If the second condition is weakened to a non-strict inequality, so that $\mathrm{Dis}(x, y)$ may be zero even if $x \neq y$, then the function Dis is called a *pseudo-metric*.

The triangle inequality – Minkowski distance for $p = 2$



The green circle connects points the same Euclidean distance (i.e., Minkowski distance of order $p = 2$) away from the origin as A. The orange circle shows that B and C are equidistant from A. The red circle demonstrates that C is closer to the origin than B, which conforms to the triangle inequality.

The triangle inequality – Minkowski distance for $p \leq 1$



With Manhattan distance ($p = 1$), B and C are equally close to the origin and also equidistant from A. With $p < 1$ (here, $p = 0.8$) C is further away from the origin than B; since both are again equidistant from A, it follows that travelling from the origin to C via A is quicker than going there directly, which violates the triangle inequality.

Means and distances

The arithmetic mean minimises squared Euclidean distance *The arithmetic mean $\boldsymbol{\mu}$ of a set of data points D in a Euclidean space is the unique point that minimises the sum of squared Euclidean distances to those data points.*

Proof. We will show that $\arg\min_{\mathbf{y}} \sum_{\mathbf{x}\in D} ||\mathbf{x}-\mathbf{y}||^2 = \boldsymbol{\mu}$, where $||\cdot||$ denotes the 2-norm. We find this minimum by taking the gradient (the vector of partial derivatives with respect to y_i) of the sum and setting it to the zero vector:

$$\nabla_{\mathbf{y}} \sum_{\mathbf{x}\in D} ||\mathbf{x}-\mathbf{y}||^2 = -2 \sum_{\mathbf{x}\in D} (\mathbf{x}-\mathbf{y}) = -2 \sum_{\mathbf{x}\in D} \mathbf{x} + 2|D|\mathbf{y} = \mathbf{0}$$

from which we derive $\mathbf{y} = \frac{1}{|D|} \sum_{\mathbf{x}\in D} \mathbf{x} = \boldsymbol{\mu}$.

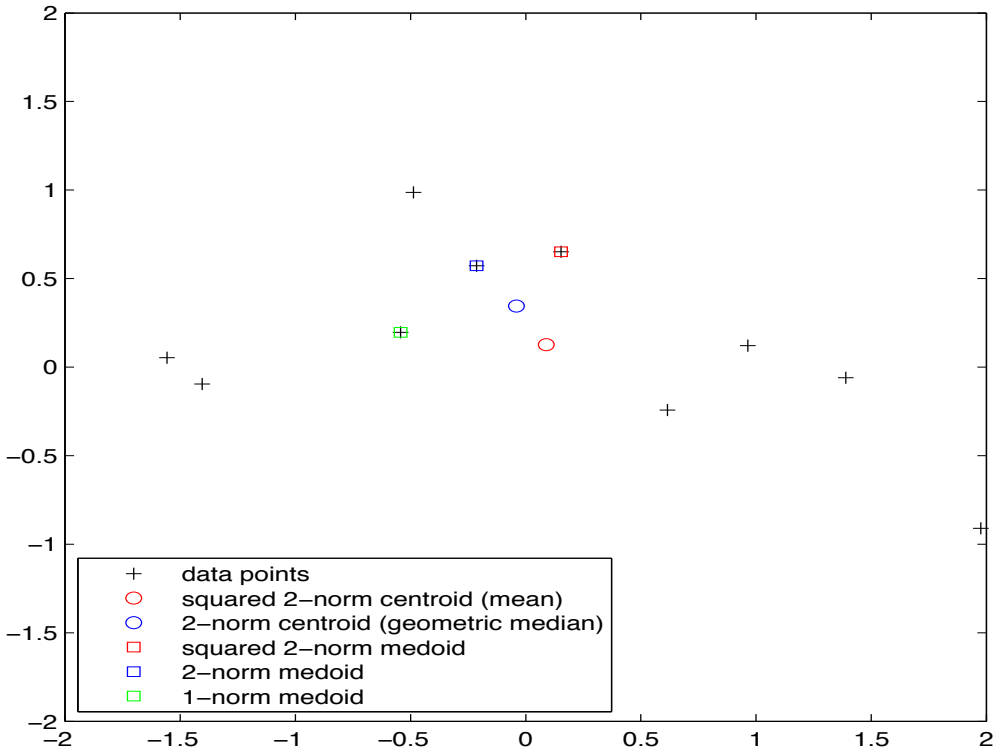
Means and distances

- Notice that minimising the sum of squared Euclidean distances of a given set of points is the same as minimising the *average* squared Euclidean distance.
- You may wonder what happens if we drop the square here: wouldn't it be more natural to take the point that minimises total Euclidean distance as exemplar?
- This point is known as the *geometric median*, as for univariate data it corresponds to the median or 'middle value' of a set of numbers. However, for multivariate data there is no closed-form expression for the geometric median, which needs to be calculated by successive approximation.

Means and distances

- In certain situations it makes sense to restrict an exemplar to be one of the given data points. In that case, we speak of a *medoid*, to distinguish it from a *centroid* which is an exemplar that doesn't have to occur in the data.
- Finding a medoid requires us to calculate, for each data point, the total distance to all other data points, in order to choose the point that minimises it. Regardless of the distance metric used, this is an $O(n^2)$ operation for n points.
- So for medoids there is no computational reason to prefer one distance metric over another.
- There may be more than one medoid.

Centroids and medoids

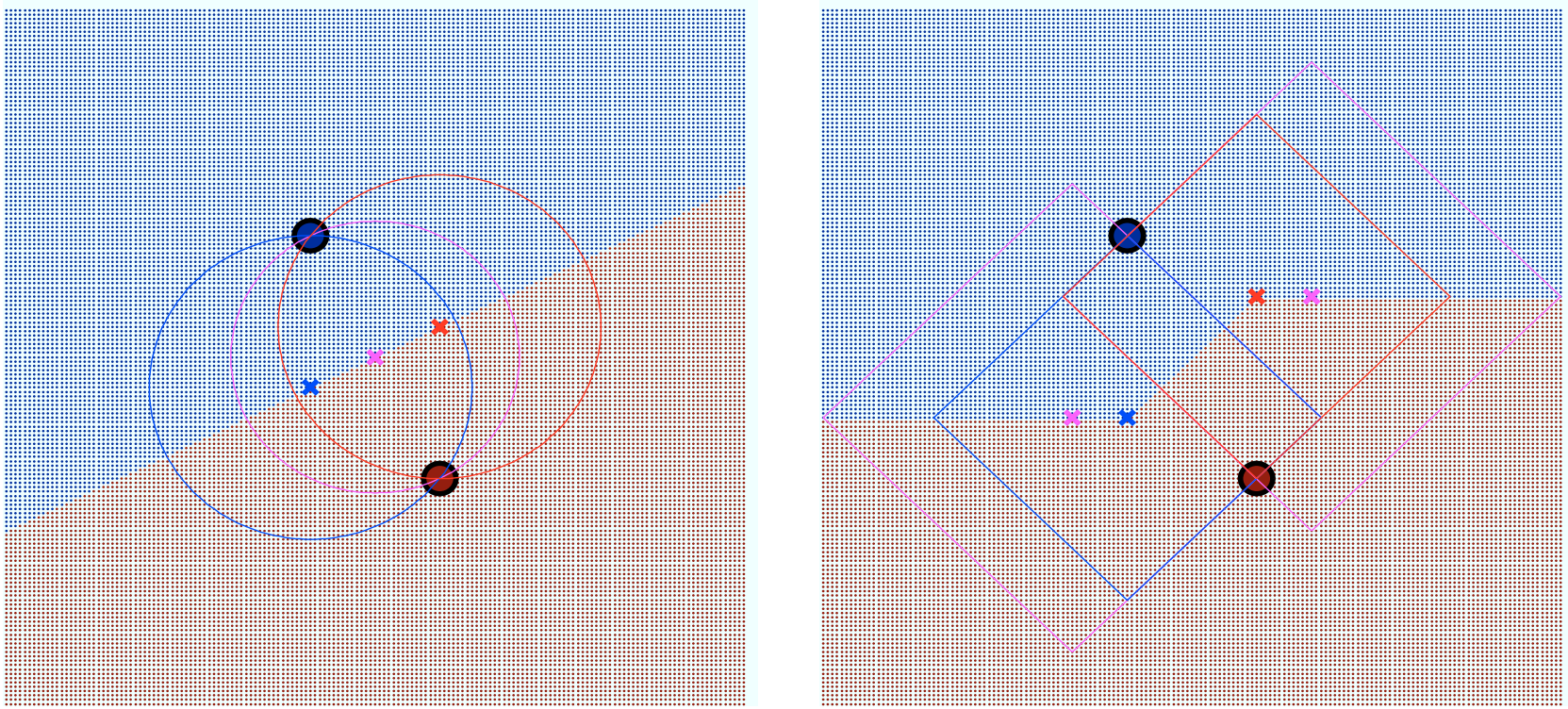


A small data set of 10 points, with circles indicating centroids and squares indicating medoids (the latter must be data points), for different distance metrics. Notice how the outlier on the bottom-right 'pulls' the mean away from the geometric median; as a result the corresponding medoid changes as well.

The basic linear classifier is distance-based

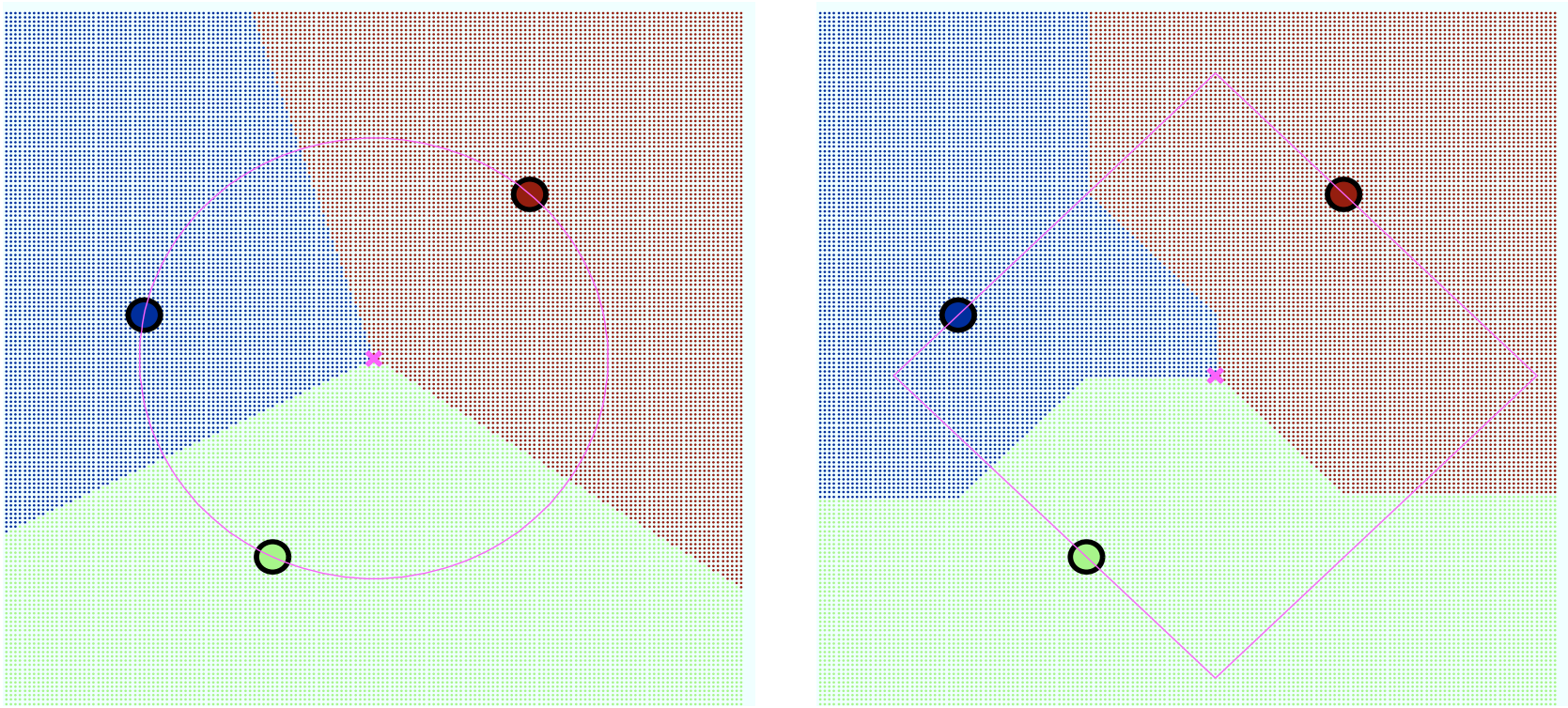
- The basic linear classifier constructs the decision boundary as the perpendicular bisector of the line segment connecting the two exemplars (one for each class).
- An alternative, distance-based way to classify instances without direct reference to a decision boundary is by the following decision rule: if $\mathbf{x}$ is nearest to $\boldsymbol{\mu}^{\oplus}$ then classify it as positive, otherwise as negative; or equivalently, classify an instance to the class of the *nearest* exemplar.
- If we use Euclidean distance as our closeness measure, simple geometry tells us we get exactly the same decision boundary.
- So the basic linear classifier can be interpreted from a distance-based perspective as constructing exemplars that minimise squared Euclidean distance within each class, and then applying a nearest-exemplar decision rule.

Two-exemplar decision boundaries



(left) For two exemplars the nearest-exemplar decision rule with Euclidean distance results in a linear decision boundary coinciding with the perpendicular bisector of the line connecting the two exemplars. (right) Using Manhattan distance the circles are replaced by diamonds.

Three-exemplar decision boundaries



(left) Decision regions defined by the 2-norm nearest-exemplar decision rule for three exemplars. (right) With Manhattan distance the decision regions become non-convex.

Distance-based models

To summarise, the main ingredients of distance-based models are

- distance metrics, which can be Euclidean, Manhattan, Minkowski or Mahalanobis, among many others;
- exemplars: centroids that find a centre of mass according to a chosen distance metric, or medoids that find the most centrally located data point; and
- distance-based decision rules, which take a vote among the k nearest exemplars.

Nearest neighbour classification

Nearest neighbour classification

- Related to the simplest form of learning: rote learning or memorization
  - Training instances are searched for instance that **most closely resembles** new or *query* instance
  - The *instances* themselves represent the knowledge
  - Called: *instance-based*, *memory-based* learning or *case-based* learning; often a form of *local* learning
- The *similarity* or *distance* function defines "learning", i.e., how to go beyond simple memorization
- Intuitive idea — instances "close by", i.e., neighbours or *exemplars*, should be classified similarly
- Instance-based learning is *lazy* learning
- Methods: *nearest-neighbour*, *k-nearest-neighbour*, *lowess*, ...
- Ideas also important for *unsupervised* methods, e.g., clustering (later lectures)

Nearest Neighbour

Stores all training examples $\langle x_i, f(x_i) \rangle$.

Nearest neighbour:

- Given query instance x_q, first locate nearest training example x_n, then estimate $\hat{f}(x_q) \leftarrow f(x_n)$

k-Nearest neighbour:

- Given x_q, take vote among its k nearest neighbours (if discrete-valued target function)
- take mean of f values of k nearest neighbours (if real-valued)

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} f(x_i)}{k}$$

k-Nearest Neighbour (kNN) Algorithm

Training algorithm:

- For each training example $\langle x_i, f(x_i)\rangle$, add the example to the list *training_examples*.

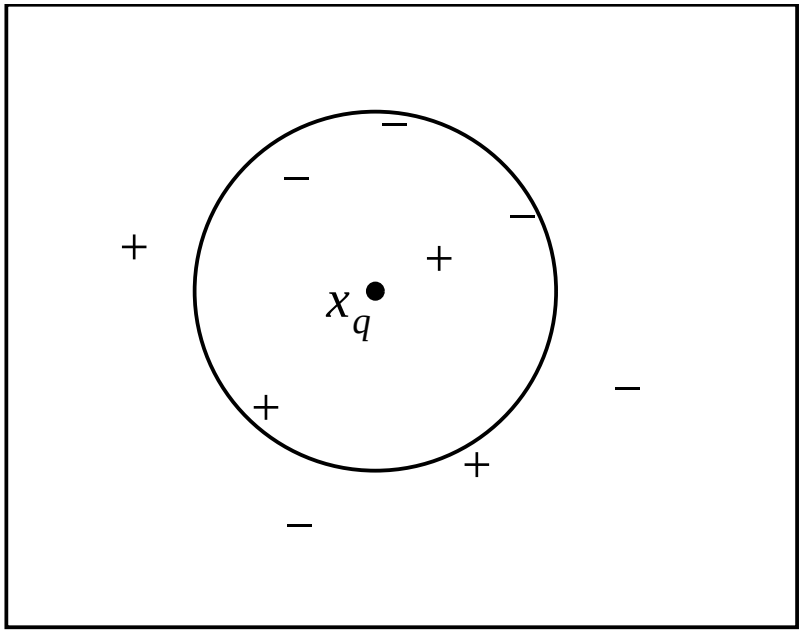
Classification algorithm:

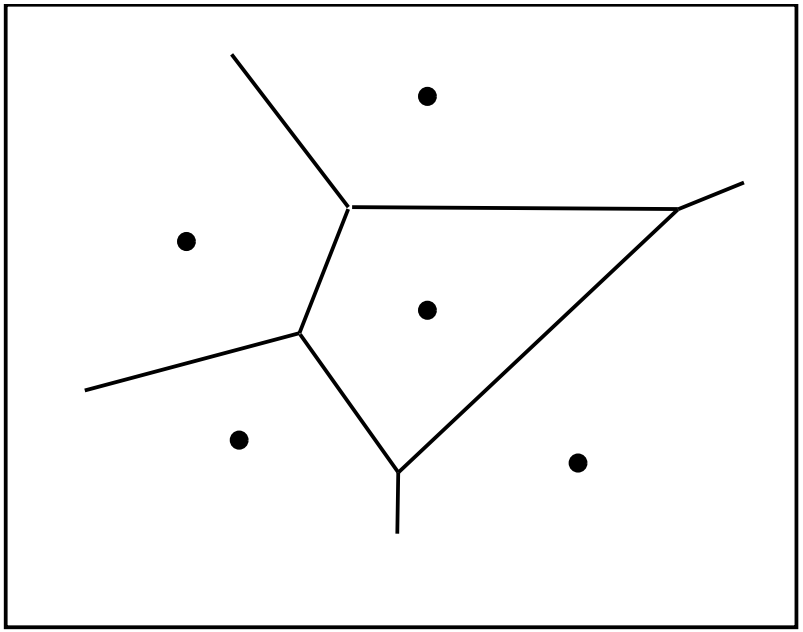
- Given a query instance x_q to be classified,
  - Let $x_1 \ldots x_k$ be the k instances from *training_examples* that are *nearest* to x_q by the distance function
  - Return

$$\hat{f}(x_q) \leftarrow \arg\max_{v \in V} \sum_{i=1}^{k} I[v, f(x_i)]$$

where $I[a, b] = 1$ if $a = b$ and 0 otherwise.

"Hypothesis Space" for Nearest Neighbour





2 classes, $+$ and $-$ and query point x_q. On left, note effect of varying k. On right, $1-$NN induces a Voronoi tessellation of the instance space. Formed by the perpendicular bisectors of lines between points.

Distance function again

The distance function *defines what is learned*, i.e., how an instance x will be classified, given a training dataset.

Instance x is described by a feature vector (list of attribute-value pairs)

$$\langle a_1(x), a_2(x), \ldots a_d(x) \rangle$$

where $a_r(x)$ denotes the value of the rth attribute (feature) of x.

Most commonly used distance function is *Euclidean* distance ...

- distance between two instances x_i and x_j is defined to be

$$\text{Dis}_2(\mathbf{x}_i, \mathbf{x}_j) = \sqrt{\sum_{r=1}^{d} (a_r(x_i) - a_r(x_j))^2}$$

Distance function again

Many other distance functions could be used ...

- e.g., *Manhattan* or *city-block* distance (sum of absolute values of differences between attributes)

$$\text{Dis}_1(\mathbf{x}_i, \mathbf{x}_j) = \sum_{r=1}^{d} |a_r(x_i) - a_r(x_j)|$$

Typically use a vector-based formalization — *norm* L_1, L_2, ...

The idea of distance functions will appear again in *kernel methods*.

Normalization and other issues

- Different attributes measured on different scales
- Need to be *normalized* (why ?)

$$a_r = \frac{v_r - \min v_r}{\max v_r - -\min v_r}$$

  where v_r is the actual value of attribute r
- Nominal attributes: distance either 0 or 1
- Common policy for missing values: assumed to be maximally distant (given normalized attributes)

When To Consider Nearest Neighbour

- Instances map to points in $\Re^n$
- Low-dimensional data, say, less than 20 attributes per instance
  - or number of attributes can be reduced ...
- Lots of training data
- No requirement for "explanatory" model to be learned

When To Consider Nearest Neighbour

Advantages:

- Statisticians have used k-NN since early 1950s
- Can be very accurate
  - at most twice the "Bayes error" for 1-NN[2]
- Training is very fast
- Can learn complex target functions
- Don't lose information by generalization - keep all instances

[2] See Hastie et al. (2009).

When To Consider Nearest Neighbour

Disadvantages:

- Slow at query time: basic algorithm scans entire training data to derive a prediction
- "Curse of dimensionality"
- Assumes all attributes are equally important, so easily fooled by irrelevant attributes
  - Remedy: attribute selection or weights
- Problem of noisy instances:
  - Remedy: remove from data set
  - not easy – how to know which are noisy ?

When To Consider Nearest Neighbour

What is the inductive bias of k-NN ?

- an assumption that the classification of query instance x_q will be most similar to the classification of other instances that are nearby according to the distance function

k-NN uses terminology from statistical pattern recognition (see below)

- *Regression* approximating a real-valued target function
- *Residual* the error $\hat{f}(x) - f(x)$ in approximating the target function
- *Kernel function* function of distance used to determine weight of each training example, i.e., kernel function is the function K s.t. $w_i = K(\text{Dis}(x_i, x_q))$

Nearest-neighbour classifier

- kNN uses the training data as exemplars, so training is $O(n)$ (but prediction is also $O(n)$!)
- 1NN perfectly separates training data, so low bias but high variance[3]
- By increasing the number of neighbours k we increase bias and decrease variance (what happens when $k = n$?)
- Easily adapted to real-valued targets, and even to structured objects (nearest-neighbour retrieval). Can also output probabilities when $k > 1$
- Warning: in high-dimensional spaces everything is far away from everything and so pairwise distances are uninformative (curse of dimensionality)

[3] See Hastie et al. (2009).

Local (nearest-neighbour) regression

Local learning

- Related to the simplest form of learning: rote learnin, or memorization
- Training instances are searched for instance that **most closely resembles** *query* or test instance
- The *instances* themselves represent the knowledge
- Called: *nearest-neighbour*, *instance-based*, *memory-based* or *case-based* learning; all forms of *local learning*
- The *similarity* or *distance* function defines "learning", i.e., how to go beyond simple memorization
- Intuition — classify an instance similarly to examples "close by" — neighbours or *exemplars*
- A form of *lazy* learning – don't need to build a model!

Nearest neighbour for numeric prediction

Store all training examples $\langle x_i, f(x_i) \rangle$.

Nearest neighbour:

- Given query instance x_q,
- first locate nearest training example x_n,
- then estimate $\hat{y} = \hat{f}(x_q) = f(x_n)$
- k-Nearest neighbour:
- Given x_q, take mean of f values of k nearest neighbours

$$\hat{y} = \hat{f}(x_q) = \frac{\sum_{i=1}^{k} f(x_i)}{k}$$

Distance function

The distance function defines what is "learned", i.e., value predicted.

Instance x_i is described by an d-vector of feature values:

$$\langle x_{i1}, x_{i2}, \ldots x_{id} \rangle$$

where x_{ir} denotes the value of the rth feature of x_i.

Most commonly used distance function is *Euclidean* distance.

Local regression

Use kNN to form a local approximation to f for each query point x_q using a linear function of the form

$$\hat{f}(x) = b_0 + b_1 x_1 + \ldots + b_d x_d$$

where x_i denotes the value of the ith feature of instance x.
Where does this linear regression model come from ?

- fit linear function to k nearest neighbours
- or quadratic or higher-order polynomial ...
- produces "piecewise approximation" to f

Distance-Weighted kNN

- Might want to weight nearer neighbours more heavily ...
- Use distance function to construct a weight w_i
- Replace the final line of the classification algorithm by:

$$\hat{f}(x_q) \leftarrow \arg\max_{v \in V} \sum_{i=1}^{k} w_i I[v, f(x_i)]$$

where

$$w_i \equiv \frac{1}{\mathrm{Dis}(x_q, x_i)}$$

and $\mathrm{Dis}(x_q, x_i)$ is distance between x_q and x_i, such as Euclidean distance.

Distance-Weighted kNN

For real-valued target functions replace the final line of the algorithm by:

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} w_i f(x_i)}{\sum_{i=1}^{k} w_i}$$

(denominator normalizes contribution of individual weights).

Now we can consider using *all* the training examples instead of just k

$\rightarrow$ using all examples (i.e., when $k = n$) with the rule above is called "Shepard's method"

Evaluation

Lazy learners do not construct an explicit model, so how do we evaluate the output of the learning process ?

- 1-NN – training set error is always zero !
  - each training example is always closest to itself
- k-NN – overfitting may be hard to detect

Use leave-one-out cross-validation (LOOCV) – leave out each example and predict it given the rest:

$$(x_1, y_1), (x_2, y_2), \ldots, (x_{i-1}, y_{i-1}), (x_{i+1}, y_{i+1}), \ldots, (x_n, y_n)$$

Error is mean over all predicted examples. Fast — no models to be built !

Curse of Dimensionality

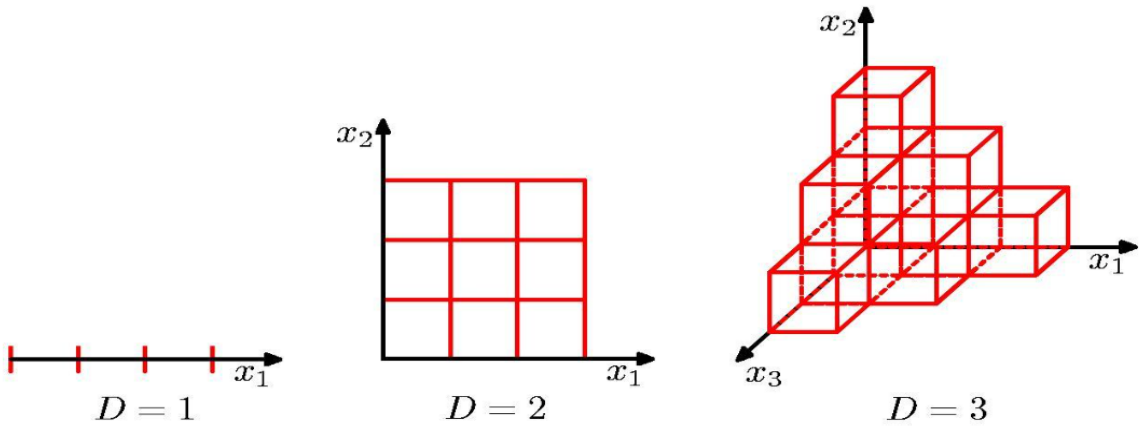
Bellman (1960) coined this term in the context of dynamic programming.

Curse of dimensionality: nearest neighbour is hard for high-dimensional instances x.

One approach:

- "Stretch" jth axis by weight z_j, where $z_1, \ldots, z_n$ chosen to minimize prediction error
- Use cross-validation to automatically choose weights $z_1, \ldots, z_n$
- Note: setting z_j to zero eliminates this dimension altogether

Curse of Dimensionality



- number of "cells" in the instance space grows exponentially in the number of features
- with exponentially many cells we would need exponentially many data points to ensure that each cell is sufficiently populated to make nearest-neighbour predictions reliably

Instance-based (nearest-neighbour) learning

Recap – Practical problems of 1-NN scheme:

- Slow (but fast $k-d$ tree-based approaches exist)
  - Remedy: removing irrelevant data
- Noise (but k-NN copes quite well with noise)
  - Remedy: removing noisy instances
- All attributes deemed equally important
  - Remedy: attribute weighting (or simply selection)
- Doesn't perform explicit generalization
  - Remedy: rule-based (like local regression) or tree-based NN approaches

Summary

- A framework for classification
- Classification viewed in terms of distance in feature space
  Distance-based. The key ideas are geometric.
- A classifier as a linear model
- Nearest neighbour classifiers
- Later we will see how to extend by building on these ideas

Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning*. Springer, 2nd edition.