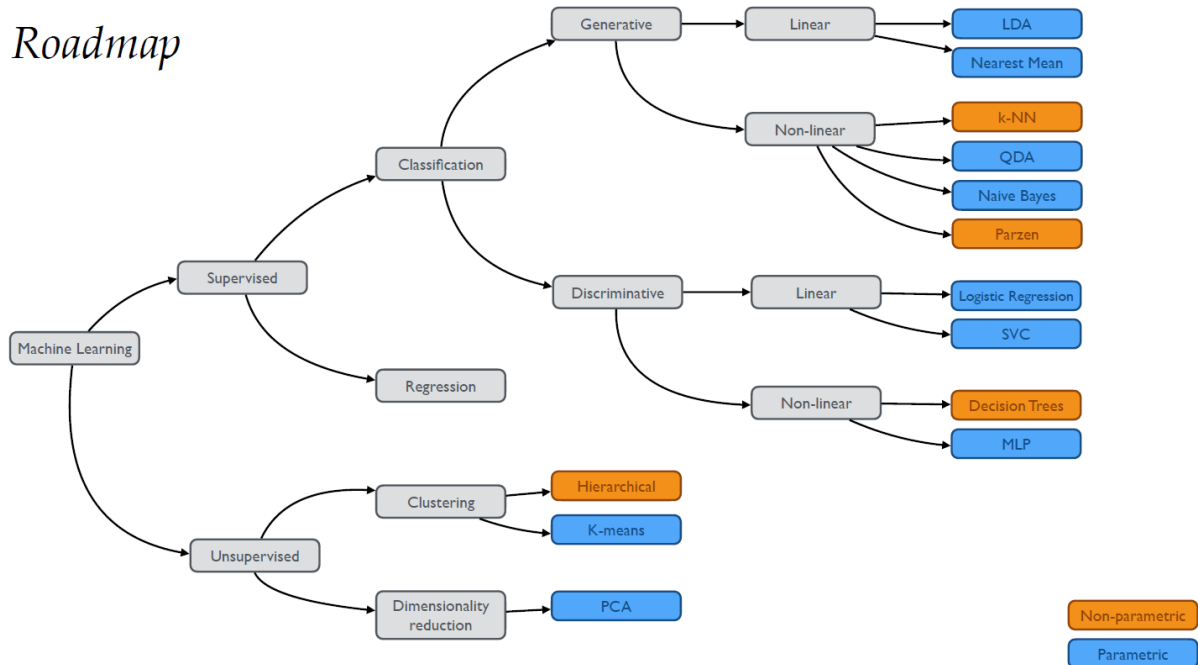


Machine Learning Summary

Aug 27, 2021

This is a summary of the CSE2510 Machine Learning subject

Roadmap



- To do: do a small cheatsheet chapter for each node in the image stating some facts, some pros and some cons, try to compare it to other nodes.

What is machine learning?

- With machine learning we can automatize boring simple repetitive tasks such as identifying patterns in a picture and group the patterns found into labels (i.e. recognising if the picture has a dog, a cat, a fruit...)
- In ML the algorithm **"learns"** from a **"training" input** data set and it applies what it **"learned"** to **new data inputs** via **"generalisation"**
 - **Learning == training on data** (more on this later)
 - Formally, **generalisation == Coming to general conclusions from (a limited number of) specific observations**
 - Example is to think "Bayesian", guessing the gender of a random pool is a 50/50 chance, but if the (training) data is updated with a pool selected from a synchronised swimming class, then the chance of guessing the gender is skewed towards female.
 - This is what the teacher means when she says that **Prior knowledge = counting observations** (i.e. number of girls in a class) and that **Learning is counting**.
 - Usually the more (representative) data (examples) the more "counting" (counting in proportion to the total pool, thus probability estimation)
 - Machine learning is about **predicting through counting (historical) data and assigning a most likely outcome** (probability)
- In a nutshell: **Machine learning is "probabilistic" classification**

Note on notation

- $p(x)$ is used for continuous variables
- $P(x)$ is used for discrete variables

- In machine learning the feature vector x is often continuous (i.e. weight, height) while the class labels are discrete (i.e. woman, man)
 - Therefore in machine learning we just use $p(x)$ for both
- In ML it is arbitrary to store the measurements of the feature vector in rows (i.e. one object example per row) or in columns (one column contains 1 object example)
- In CSE2510 we store object in rows such that:

$$\begin{bmatrix} -1 & -1 \\ -1 & 1 \\ 2 & 0 \\ 3 & 0 \end{bmatrix}$$

- Is a two-feature dataset with four objects

Supervised learning

- CSE2510 (and most of applications of ML) focus on supervised learning, that is "**learning by example**":
 - Given input-output examples, (explicitly) determine input-output function (i.e. probability of belonging to a "label" (apple, dog, cat, woman):
 - General input-output function: $y = AX + b$
 - X = input = set of features = design matrix
 - A = weight or coefficients = size of the effect of x on prediction y
 - b = bias
 - Dataset with label for each training example. Learns the association between example and label.
 - Example: inputs are apples and the output is the probability of the apple being red (for a human it's rather obvious or inferred by context (i.e. lighting) whether an apple is red or not, but a ML input-output function won't output binary but as we observed before in the gender example the probability of the apple being red)
 - If we use "good" training data, the function should be able to generalise to new and previously unseen (apple) examples
 - The data needs to be relevant, you can't train the red apple example with oranges, or with only red apples, you need to feed the training set with both green and red apples with already defined outcomes to let the algorithm learn the difference between those sets of pictures
- Alternatively, unsupervised learning does not use predefined outcome labels for the algorithm and the algorithm decides on its own how to group the inputs, i.e. an algorithm might group dogs and chimps in the same group while bald bodybuilders and hippopotamus in the same one because it decided to use hair/fur as a label feature rather than bipedalism.
- There are other types of ML but they are not discussed in this course

Terminology

Machine Learning Algorithms

- A machine **learning algorithm** is an algorithm that is able to learn from data. Formally defined in 1997 as:
 - A computer program is said to learn from **experience** (E) with respect to some class of **tasks** (T) when it's task **performance** (P) improves with (more) experience.

Task

- **Learning** is our **means of attaining the ability to successfully perform a task**
 - **Learning != the task**

- I.e. a robot being able to walk with "walking" as the task:
 - We could (manually) program the robot to walk (QWOP style)
 - Alternatively, we could just program the robot to **learn** how to walk (this being just `import minecraft.py` style in python as opposed to coding the biomechanics yourself...)
 - So ML is a lazy yet effective solution at the expense of consuming lots of CPU and training resources (and as being a brute-force approach rather than an elegant "mathematical deduction proof-based" approach (it's induction instead))
- Machine learning tasks are usually described in terms of how the machine learning system should process an **example**
 - An example is a collection of **features** that have been quantitatively measured from some object or event (i.e. a picture of a cat) that we want the machine learning to process
 - We typically represent the example as a vector $x \in \mathcal{R}^n$ where each entry X_i of the vector is a feature (such that colors and shapes of groups of pixels)
 - Formally, the features of an image are usually the values of the pixels in the image
- Many **tasks** can be solved with machine learning ML such as:
 - **Classification:**
 1. Take measurements (features) of the objects (examples) (training data)
 2. Plot each object (examples) (training data)
 3. Label (apple, cat...) each object (examples) and draw the decision boundary (training data)
 - Avoid overfitting the decision boundary to the training data as this will make it harder to predict the label of new data
 4. Predict label of new objects (test data)
 - **Regression:** predicting a continuous value (i.e. house prices)
 - **Transcription:** Transcribe a relatively unstructured representation of some kind of data into discrete textual form (i.e. speech recognition like autogenerated subtitles)
 - More... (not covered in this course)

Experience (dataset)

- This is the dataset to train the ML algorithm
- Dataset: Collection of many examples or objects
- May or may not be supervised, if it is, then each of the examples in the training dataset is provided with an explicit label

Training set

- The set of examples used to finetune the algorithm "parameters"
 - At the end of the day, the function that returns the likeliness of an input to belong to a specific label is a function of the form $y = ax + by + cz \dots$ with zillions of weights known as parameters (well, or just use the matrix notation)

Test set

- Independent from the test set, thus it is used exclusively to determine whether the finetune of the parameters was accurate or not (formally known as **generalisability** of the trained model to unseen data).
- It must come from the same pool (same probability distribution as the training set)
- Goal of training: Learn a function that can predict a label y for a new x with as little error as possible = an input-output function that can generalise to new, unseen examples (without labels)
 - Learn model parameters a and b so that the error of the function's predictions is minimised $y = ax + b$

Features

- Each data point/example/object is described in terms of features (i.e. shape, color, length, width of something in image)
- Features give a specific view of the objects: YOU (the user) are responsible for it
- Good features allow for pattern recognition, bad features allow for nothing
 - Thus more is not always better

Design matrix

- This matrix is just a way of describing a dataset
- Recall that each observation/example/object was regarded as a vector $x \in \mathcal{R}^n$ where each entry X_i of the vector is a feature
- The design matrix X is a matrix that contains a different observation in each row, each column of the matrix corresponds to a different feature
 - This is expressed as $X \in \mathcal{R}^{m \times n}$
 - with m = number of observations and n = number of features

- One example/object per row

- Design matrix: $X \in \mathbb{R}^{344 \times 5}$

where, $X_{i,2}$ is the bill length of penguin i

and $X_{i,3}$ is the bill depth of penguin i , etc.

	species	bill_length _mm	bill_depth _mm	flipper_len gth_mm	body_m ass_g	sex
1	Adelie	39.1	18.7	181	3750	male
2	Adelie	39.5	17.4	186	3800	female
3	Adelie	40.3	18	195	3250	female
4	Adelie	NA	NA	NA	NA	<NA>
5	Adelie	36.7	19.3	193	3450	female
6	Adelie	39.3	20.6	190	3650	male



- See that $X_{i,0}$ is the label, $X_{i,n}$ are the features with $n > 0$ and the task: classify the penguins into 3 species based on the measurements (= features)
 - In unsupervised learning the label is not provided in the training set and the dataset is divided into clusters of "similar" objects

Performance

- Usually this performance measure is specific to the task, but we often just use the **accuracy** of the model (i.e. to predict the correct label)
 - The 1 - error rate is equivalent. The error rate is also regarded as "the expected 0-1 loss"
- Other performance measures include arbitrary weights for true/false positives/negatives (i.e. we might prefer to avoid false positives over maximizing true positives in insurance fraud detection, a true positive saves you 10k on average but a false positive can sue you on court and you lose 1 million and get bad reputation)
- Different ML techniques require different performance measures.

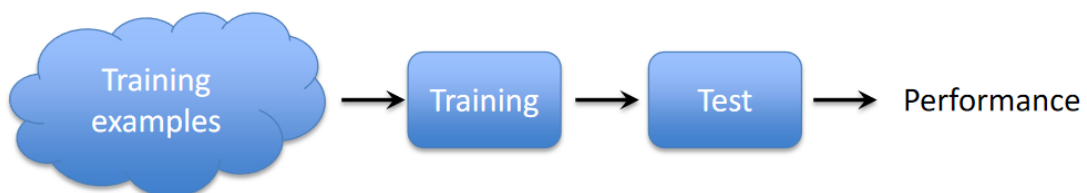
Machine Learning is soft and hard classification (probabilistic)

- ML learning (algorithms whose performance improves with experience) regards “soft” classification tasks when probabilities to belong to a label are assigned with **conditional probabilities** based on **Baye’s rule** (previous facts update the probability)
 - This is covered in Probability theory
- ML uses “hard” classification when establishing decision boundaries (percentage threshold that divides belonging to a label or not)
 - This is covered in decision theory
- The combination is called probabilistic classification, which is done using one or multiple ML techniques. This course covers:
 - Support vector machines
 - Linear regression
 - Neural networks
- No free-lunch theorem: no machine learning algorithm is better than any other as every classification algorithm has the same error rate for classifying new data. However, each algorithm is suitable for a different set of resources available by the researcher and for a specific task.

Machine Learning Pipeline

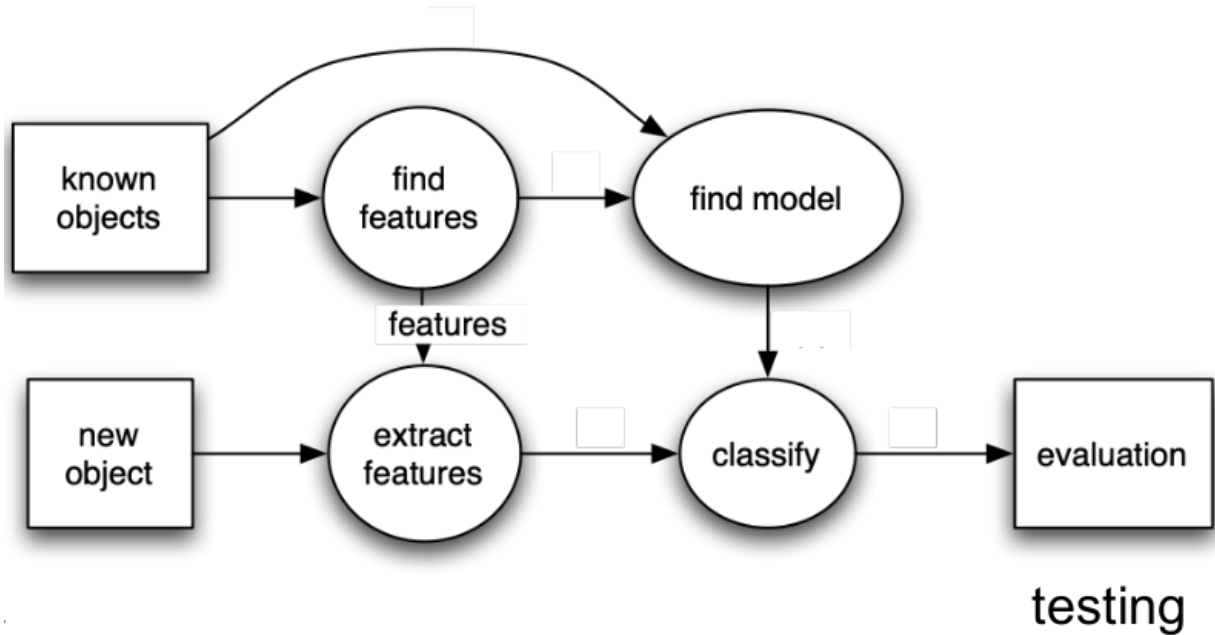
- Regardless of the technique:

1. Train the ML algorithm using a *dataset* of *examples* with *features* for a specific *task*
2. Test the *generalisability* of the ML algorithm on an unseen testset
3. Quantify *performance* using an accurate and suitable measurement



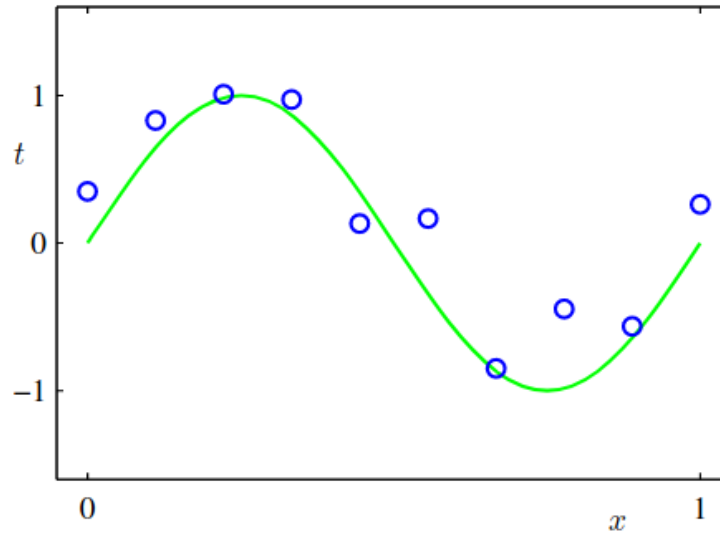
- Factors that determine Performance:
 - The ML algorithm’s ability to Make the **training error** small
 - The ML algorithm’s ability to Make the **gap between training and test error small**

applying, generalisation



Non-probabilistic example: Polynomial Curve Fitting (Linear Model)

- Regression problem used to introduce some terms. But note that this is not generally the correct ML approach.
- Suppose we observe a float `input variable` x and we want to use this input variable x to predict the value of another float `target variable` t
 - Secretly $t(x) = \sin(2\pi x) + b$ with b being some random noise and the goal of the exercise is to identify a function that approximates the real $t(x)$.
- We're given a training set comprising N observations of x together with their corresponding t
 - This may be written as tuples in set theory notation: $N = \{(x_1, t_1), (x_2, t_2), \dots, (x_n, t_n)\}$
 - Or it could be written separately as:
 - $\mathbf{x} \equiv (x_1, \dots, x_n)^T$
 - $\mathbf{t} \equiv (t_1, \dots, t_n)^T$
- Figure below shows a plot of a training set comprising $N = 10$ data points (blue circles) (x_i, t_i)
- The green curve shows the function $t(x) = \sin(2\pi x)$ (without the noise) that implicitly applies to the observed the data.
 - Our goal is to predict the value of new \hat{t} for some new value of \hat{x} (hat denoting new data set) different from training set), without knowledge of the green curve.
 - This implicitly involves trying to discover the underlying function $\hat{t}(x) = \sin(2\pi x)$

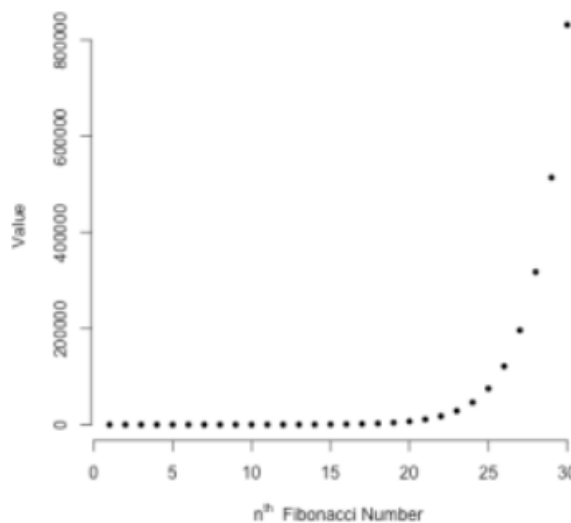


- The real observed data (and very likely the new one as well) are corrupted with noise, and so for a given \hat{x} there is uncertainty as to the appropriate value for \hat{t}
 - [Probability theory](#) provides a framework for expressing such uncertainty in a precise and quantitative manne
 - [Decision theory](#) allows us to exploit this probabilistic representation in order to make predictions that are optimal
- For the moment, however, we shall proceed rather informally and consider a simple approach based on curve fitting.
- In particular, we shall fit the data using a polynomial function of the form (note that $t \approx y$):
 - $y(x, \mathbf{x}) = w_0 + w_1x + w_2x^2 + \dots + w_nx^n$

◦

$$y(x, \mathbf{x}) = \sum_{i=0}^n w_i x^i$$

- This polynomial to approximate $\hat{t}(x)$ is an approximation to a known calculus series (series are infinite)
 - sequences are infinite list of numbers written in a definite order i.e. $f(n) = f(n - 2) + f(n - 1)$ for $n \leq 3$
 - Since a sequence is a function whose domain is the set of integers, its graph consists of discrete point coordinates.



- series are the sum of a sequence of numbers (they're also discrete)
 - A series of the form $w_0 + w_1(x - a) + w_2(x - a)^2 + \dots$ are called a power series
 - Any function f for which all derivatives exist in some point (such as a continuous function like $\sin(x)$) can be expressed as a power series where
 - $w_0 = f(a)/0! = f(a)$
 - $w_1 = f'(a)/1!$

- $w_2 = f''(a)/2!$
- $w_3 = f'''(a)/3!$
- ...
- These are known as **taylor series**
- Taylor polynomials are "finite taylor series" that better approximate the function f as M increases (degree of the polynomial)

$$\circ \sum_{i=0}^M \frac{f^{(i)}(a)}{i!} (x - a)^i$$

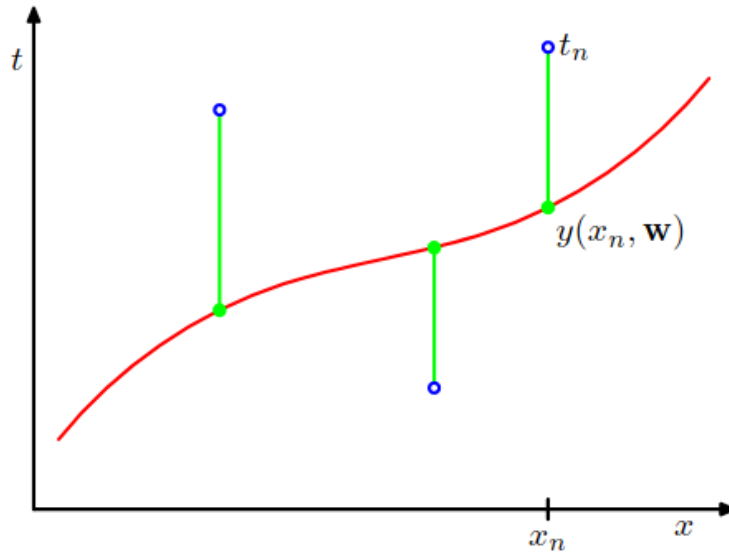
- where $f^{(i)}(a)$ denotes the i th derivative of f evaluated at the point a .
- We do not know which function our taylor polinomial is supposed to approximate but we will see that we can come up with a linear model that regards the selection of weights and then with an error function for those weights that can be minimized into a single solution.
- The weights (polynomial coefficients) w_0, w_1, \dots, w_n are collectively denoted by the vector \mathbf{w} although in statistics they are the vector β and X is the matrix that represents the polynomial powers (and ϵ is noise) such that $y = X\beta + \epsilon$ and:

$$y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, X = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 \end{bmatrix}, \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix}, \epsilon = \begin{bmatrix} \epsilon_0 \\ \epsilon_1 \\ \epsilon_2 \end{bmatrix}$$

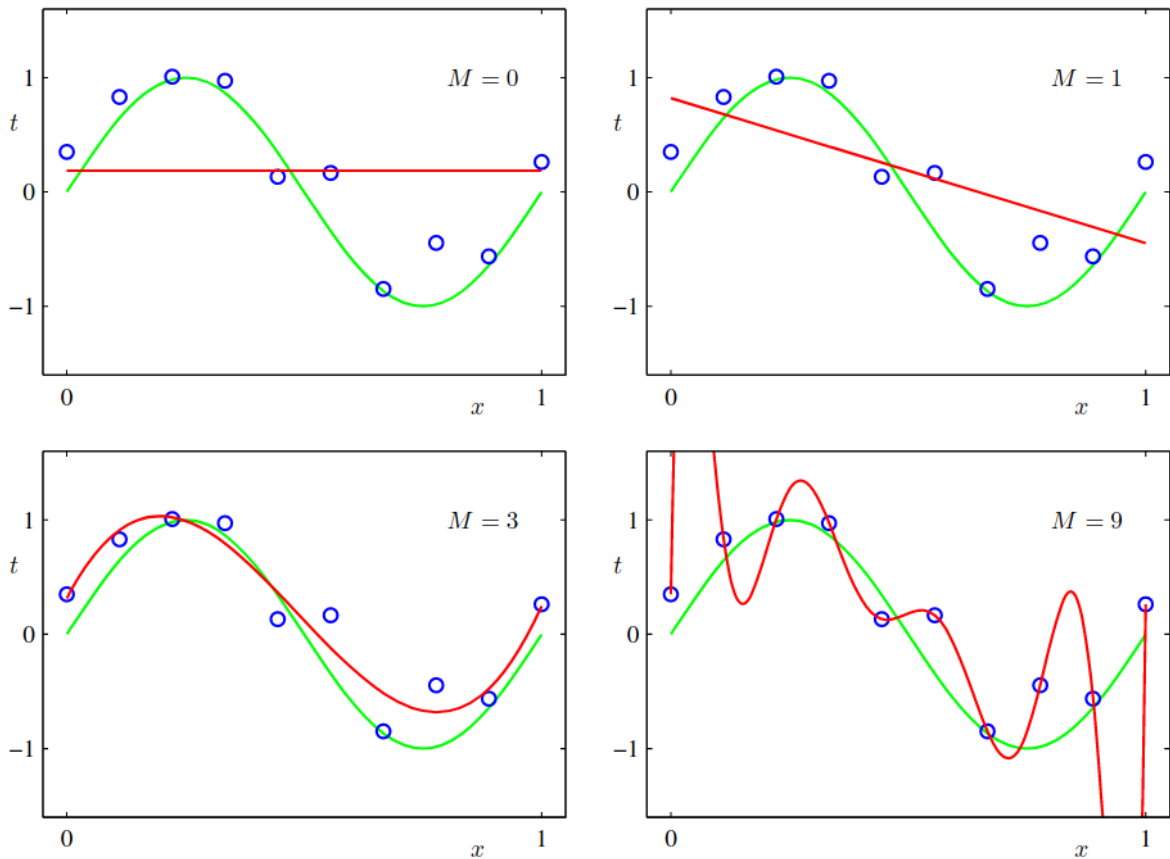
- In the CSE2510 this is expressed as $y(x, \mathbf{w}) = X\mathbf{w} + b$
- Although the function having various powers of x makes it implicitly nonlinear, it immediately becomes a linear algebra equation once the x values have been inserted and we just then have $y = \text{constantMatrix} * \text{vector} + \text{ErrorVector}$, which is reduced to a system of linear equations with n unknowns and n equations, which in matrix notation is just solving for $A\mathbf{x} = b$.
 - This is formally known as a linear model
 - From linear algebra we know that a system of equations either has 0, 1 or infinite solutions
 - It is very likely that the taylor polynomial won't solve the system (the polynomial degree is far from ∞ to approximate y "perfectly" and we secretly know that there was b random noise added deliberately)
- The values of the coefficients \mathbf{w} will be determined by fitting the polynomial to the training data
 - Since there's very likely no solution, we will just minimize the **error function** that measures the misfit between the model function $y(x, \mathbf{w})$ and the actual training set data points (see that we're not using the hat notation as we are actually using training data and not new data)
 - One simple and popular choice of error function is given by the sum of the squares of the errors between the approximation model $y(x, \mathbf{w})$ for each data point x_j and the actual target value t_j .

$$\circ E(\mathbf{w}) = \frac{1}{2} \sum_{i=0}^n (y(x_i, \mathbf{w}) - t_i)^2$$

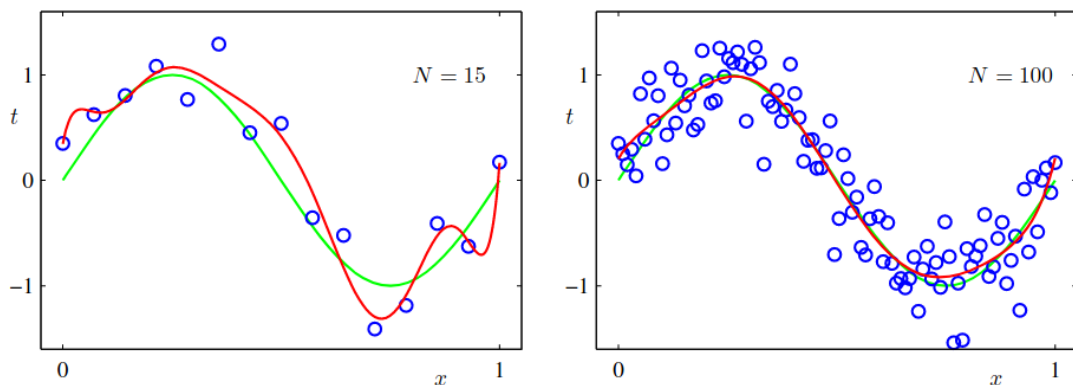
- Squaring takes care of "absolutizing" the negative differences and halving the error is just for convinience, it doesn't change the final solution anyway
- The equation pretty much gives the error size of a given set of polynomial weights (vector w).



- The error function corresponds to (one half of) the sum of the squares of the displacements (shown by the vertical green bars) of each data point from the function $y(x, \mathbf{w})$.
- $E(x) = 0$ if and only if the function $y(x, \mathbf{w})$ passes exactly through each training data point
- Because the error function is a quadratic function of the coefficients w , its derivatives with respect to the coefficients will be linear in the elements of w , and so the minimization of the error function has a unique solution, denoted by w^* , with the resulting polynomial given by the function $y(x, \mathbf{w}^*)$
 - We can actually solve the least square solution \mathbf{w}^* with the so called "normal equation" of linear algebra (instead of calculating a long derivative).
 - $$X^T X \mathbf{w}^* = X^T y$$
 - This requires X to be invertible, if it isn't you can instead project y onto the columnsspace of X and then solve for $Xw = \hat{y}$
- There remains the problem of choosing the order M of the polynomial, as we see below, a taylor ponimial with a high a low degree and a high degree produce poor results (note that the higher the degree of the polynomial, the more columns X has)



- Low order polynomial fit the training data very bad and we can clearly infer that they'll approximate new data very poorly as well
- In the example below we see that $M=3$ produces close results for the training data but also for new data.
 - The goal is to achieve good generalization by making accurate predictions for new data, therefore we might be inclined to choose $M=3$. Although it seems paradoxical. After all, the best solution would be $y = \sin(2\pi x)$ itself.
- High order polynomials fit the training data perfectly, in fact $M=9$ has 0 error. But we can see that once new data is evaluated the model (red line) will give many errors. This is known as **over-fitting**
 - The reason it was possible to achieve $E(x)=0$ with $M=9$ is that this polynomial contained 10 degree of freedom corresponding to the 10 weight coefficients, which can be tuned exactly to fit the 10 data points in the training set.
 - The problem is that the polynomial is tuning too much to the random errors on the target values of the training set
 - For a given model complexity, the over-fitting problem becomes less severe as the size of the training data set increases, we can see that the higher order polynomial get's closer to $y = \sin(2\pi x)$ than $M=3$ did



- This follows the machine learning nature: an algorithm gets better with more training data

- One rough heuristic that is sometimes advocated is that the number of data points should be no less than some multiple (say 5 or 10) of the number of adaptive parameters in the model (i.e. the Taylor polynomial degree)
- However there's no need to use thumbrules. By adopting a Bayesian approach, overfitting problem can be avoided. There is no difficulty from a Bayesian perspective in employing models for which the number of parameters greatly exceeds the number of data points
 - In a Bayesian model the effective number of parameters adapts automatically to the size of the data set

Probability theory in machine learning

- In the context of ML: we want to design functions that classify an unknown object in the most likely class
- Our task is to determine what "most likely" is
 - For which we for any ML algorithm we will always use a form of conditional probability:
 - $$p(y|x)$$
 - which is read as "Probability of y knowing that x happened" (formally: probability of y given x)
 - In the context of ML:
 - $$p(\text{class}|\text{object})$$
 - "Probability of a particular object belonging to a particular class"
 - $$p(\text{label}|\text{featurevector})$$
 - "Probability of an example being a specific label given a set of features"
 - Recall that the feature vector is how we describe an object
- Recall that all the tuples of $(x, p(x))$ are regarded as the "probability distribution" of x.
 - The joint probability distribution would be all the tuples of $(x, y, p(x, y))$
- Our ML classifier algorithm function will therefore not return a black and white answer saying that "input belongs to class N", instead the algorithm function will return the probability of an input belonging to a class (i.e. when image recognition have "90% dog" output).
- Only after having then outputted a probabilistic score, will we use decision theory to decide whether the score (probability) is high enough to assign it to a label, to not assign, or to leave it as "unclear".

Discrete random variable

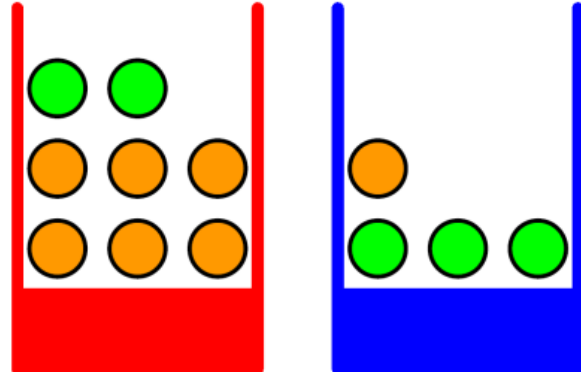
- Discrete (i.e. "int", "enum" instead of "float"(kinda)) **random** variables are capitalized
 - each of the possible values that they can take are expressed in small cap (for enum) or the actual discrete number value they can take

Random variable: box, B

- Prob of selecting the red box = $p(B = r): \frac{4}{10}$
 - Prob of selecting the blue box = $p(B = b): \frac{6}{10}$
- } Mutually exclusive
Probability must sum to 1

Random variable: fruit, F

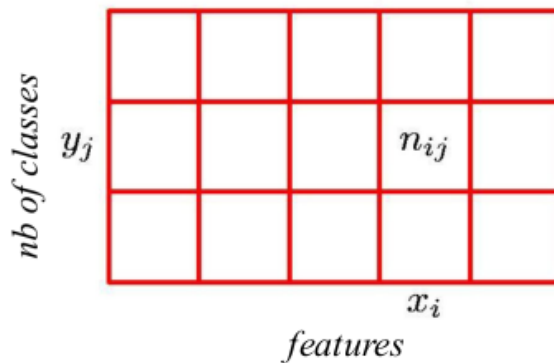
- Values: o (range) and a (pple)



Joint probability

- The joint probability of 2 random variables occurring can be inferred graphically:
 - Probability that $X = x_i$ (F) and $Y = y_j$ (B):

$$p(X = x_i, Y = y_j)$$
 - E.g., probability that $F=o$ (x) and $B=r$ (y)



Joint Probability

$$p(X = x_i, Y = y_j) = \frac{n_{ij}}{N}$$

JDelft

- The joint probability is equal to the number n_{ij} of cases where both x_i, y_j occurred, divided by the total number of cases N (which is the number of cases from the cross product of X and Y possible values sets only when the objects of the sets have all the same probability, if not you have to count them manually yourself from the context)

Sum rule

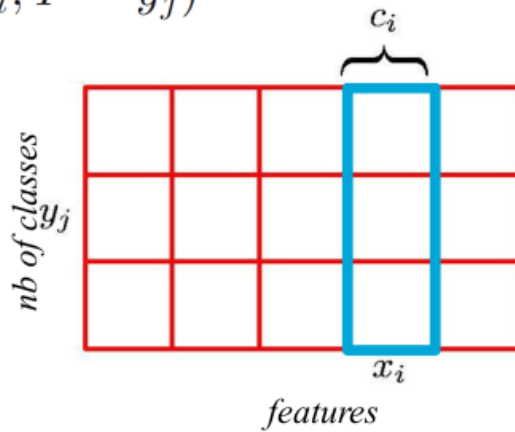
- Probability that $X = x_i$ irrespective of $Y: p(X = x_i)$

$$p(X = x_i) = \sum_{j=1}^L p(X = x_i, Y = y_j)$$

c_i : nb of trials in column i

$X = x_i$ irrespective of Y

N : total number of instances



Marginal Probability

$$p(X = x_i) = \frac{c_i}{N}$$

JDelft

- L = number of total possible y values
- Example let S = sum of 2 dice, M = max of 2 dice
 - $P(S=3, M=2)$ would be 2 scenarios $((1,2), (2,1))$ of out of the 6×6 possible, so $p(S = 3, M = 2) = 2/36$
- If we would just be given the joint probability mass function $p(S = a, M = b)$ (inside cells) we can derive the individual probabilities of $p(S = a)$ and $p(M = b)$ respectively: i.e. $P(S = 6) = P(S = 6, M = b_1) + P(S = 6, M = b_2) + \dots + P(S = 6, M = b_L)$.

Joint distribution and marginal distributions of S and M .

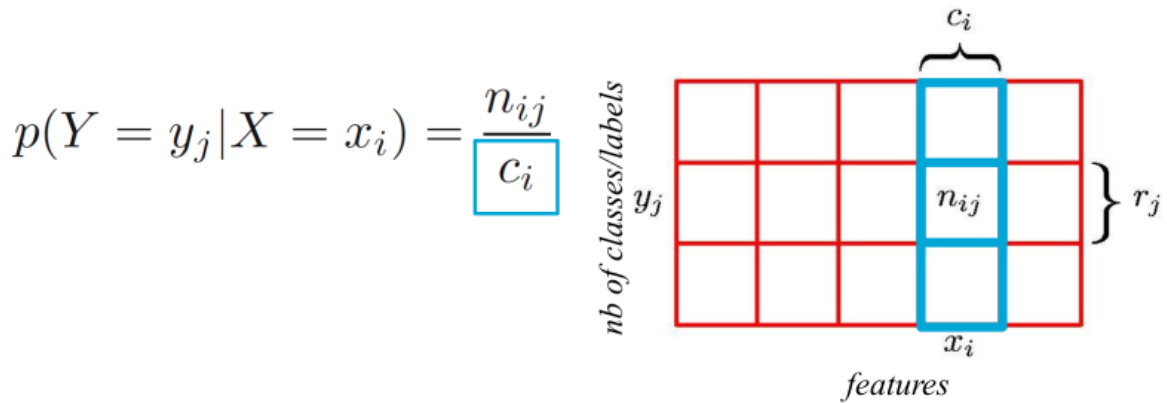
		b						
		1	2	3	4	5	6	
a								$p_S(a)$
2		1/36	0	0	0	0	0	1/36
3		0	2/36	0	0	0	0	2/36
4		0	1/36	2/36	0	0	0	3/36
5		0	0	2/36	2/36	0	0	4/36
6		0	0	1/36	2/36	2/36	0	5/36
7		0	0	0	2/36	2/36	2/36	6/36
8		0	0	0	1/36	2/36	2/36	5/36
9		0	0	0	0	2/36	2/36	4/36
10		0	0	0	0	1/36	2/36	3/36
11		0	0	0	0	0	2/36	2/36
12		0	0	0	0	0	1/36	1/36
	$p_M(b)$	1/36	3/36	5/36	7/36	9/36	11/36	1

- Marginal distributions are those on the borders and the joint distribution is made from the inside cells.
- With the joint, we can always find the marginals (and conditionals in all orders), but with the marginals we cannot always find the joint (nor the conditionals)

Conditional probability (and product rule)

- for $p(y|x)$ it's the number of cases where y and x holds, divided by the number of cases where x holds

- Probability $Y = y_j$ given that $X = x_i$: $p(Y = y_j | X = x_i)$
- E.g., probability that $B=r$ given that $F=o$



- If you divide both the numerator and the denominator by N each, then you actually get:

- $$p(y|x) = \frac{p(y, x)}{p(x)}$$

- $$p(y, x) = p(y|x)p(x)$$

- Which is the product rule

Fundamental probability rules

- $p(X, Y)$: joint probability = probability of X **and** Y
- $p(Y|X)$: conditional probability = probability of **Y given X**
- $p(X)$: marginal probability of X
- $p(X, Y) = p(Y, X)$: symmetry property
- $p(X, Y) = p(Y|X) p(X)$: product rule

Bayes rule

Bayes' Rule

With labeled examples of the classes → estimate a probability density per class

$$p(\text{label} | \text{object}) = \frac{p(X|Y)p(Y)}{p(X)}$$

Difficult to estimate for continuous variables

- The power behind bayes rule is that while $P(Y|X)$ might not be given, we can on our own find $P(X|Y)$ as well as the marginal probabilities.
 - Note that bayes rule is just a conditional probability rule whose joint probability term has been replaced with the product rule equivalent
- With that we can calculate the conditional probability of an object belonging to a label, which is basically the solution for the machine learning algorithm we're trying to design!
- In ML you'll hear the marginal probability of y being referred to as the "prior probability"
- In ML you'll hear the conditional probability of y given x as the "**posterior probability**"
- x is often regarded as feature or object (object with certain features)

Prior probability

Posterior probability

$$p(Y|X) = \frac{p(X|Y)p(Y)}{p(X)}$$

Prior prob of selecting the red box (Y), i.e., *before* observing an orange:

$$p(B=r): \frac{4}{10}$$

Posterior prob of selecting the red box (Y), i.e., *after* observing an orange:

$$p(B=r | F=o): \frac{2}{3}$$



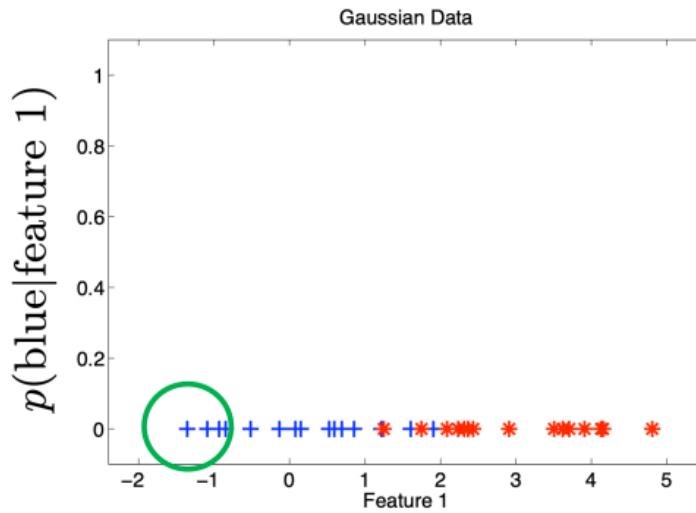
Relation to ML

- During the training phase, the training data helps us estimate the "probability distribution over a set of classes" (aka $p(\text{label})$)
- During the testing phase, the training data helps us estimate the "probability that a sample belongs to a class" (aka $p(\text{label}|\text{object})$)

Conditional probability for continuous variables

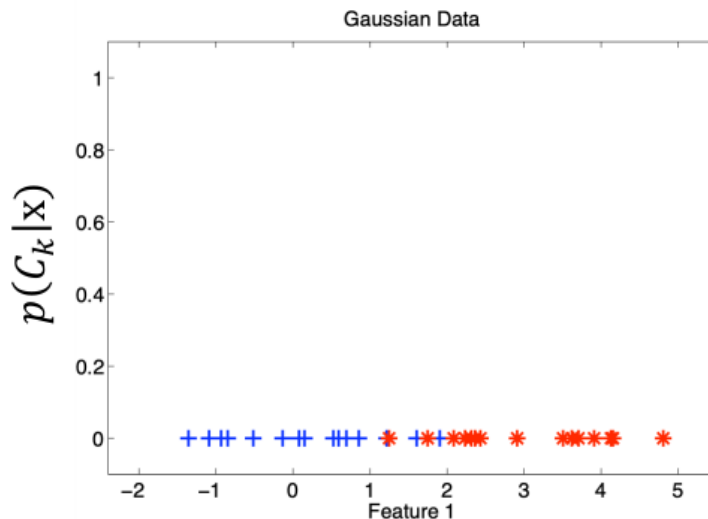
Estimating the probability distribution over a set of classes: The continuous case

- For each object we want to estimate $p(\text{blue}|\text{feature 1})$

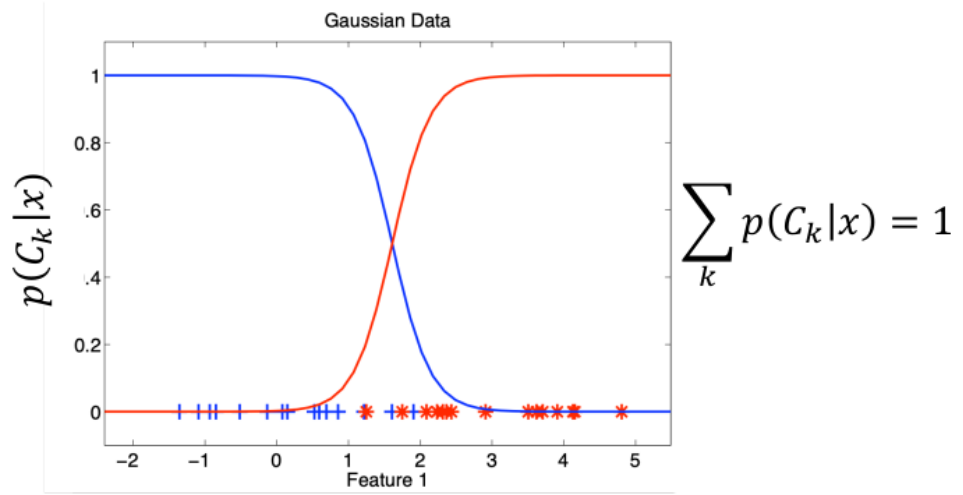


- If we define feature 1 as having a value between -2 and -1, we can see that $p(\text{blue}|\text{feature 1})$ is 100%

- For each object we want to estimate $p(C_k|x)$



- The training data (historical) eventually estimates the probability distributions for each of the class posterior probabilities
 - Although we start with a "discrete" amount of training data, the computer will estimate for us the continuous probability functions based on the data that we have



- In this example the classes (apple vs orange) are mutually exclusive, therefore the class posterior probabilities add to 1 for any continuous value of feature 1

Decision theory

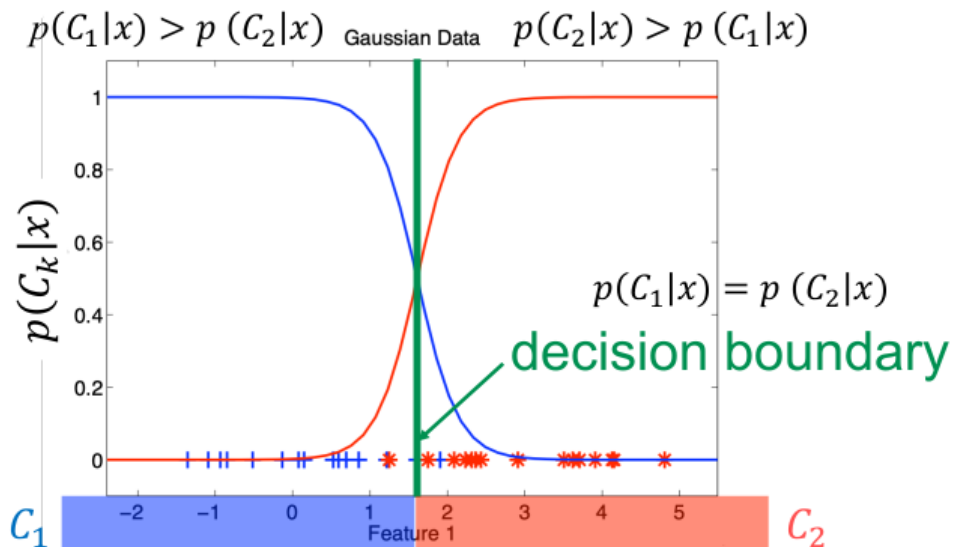
- Decision theory complements the “soft” classification (giving probability of belonging to a class rather than black and white outcomes) with “hard” (black and white) final outcomes.

Decision boundary

- In order to ultimately classify x to a given class, we *generally* have $n-1$ decisions boundaries for n classes

“Hard” classification of the new object x

- Assign the label of the class with the highest posterior probability

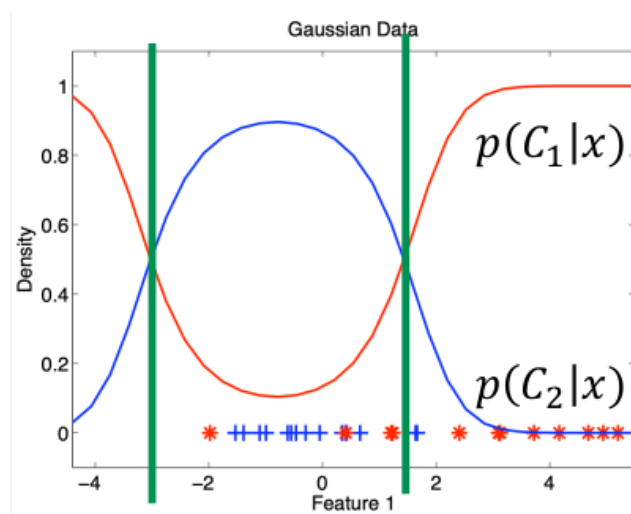


- The criteria to assign an object to a class is simply that the (posterior) conditional probability of that class is higher than the other ones. Basically the most likely class is selected.

Soft-Hard classification process recap and complicated decision boundaries

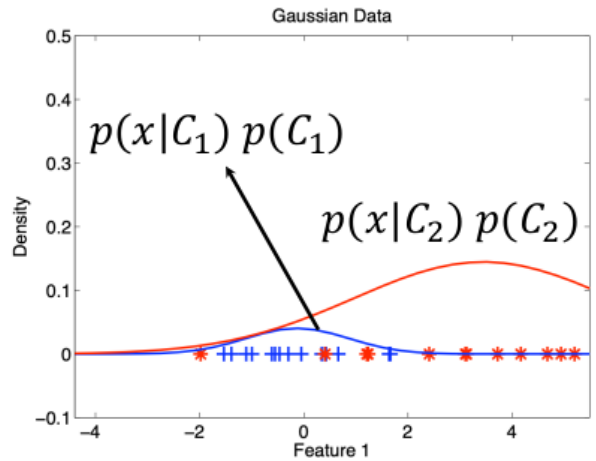
- The training data defines the probability of a feature present in a label (note that this class conditional probability is the **reversed** of the posterior (but it's not the prior, the prior would be $p(C_k)$, we're just reversing the roles of C_k since the training data itself intrinsically describes $p(x|C_k)$).
 - (Soft classification) From the training set we do have the marginal probabilities to plug the values into bayes rule and get:

$$p(C_k|x) = \frac{p(x|C_k)p(C_k)}{p(x)}$$
 - (Hard classification) Assign the objects to the label that has the highest class posterior probability $p(C_k|x)$



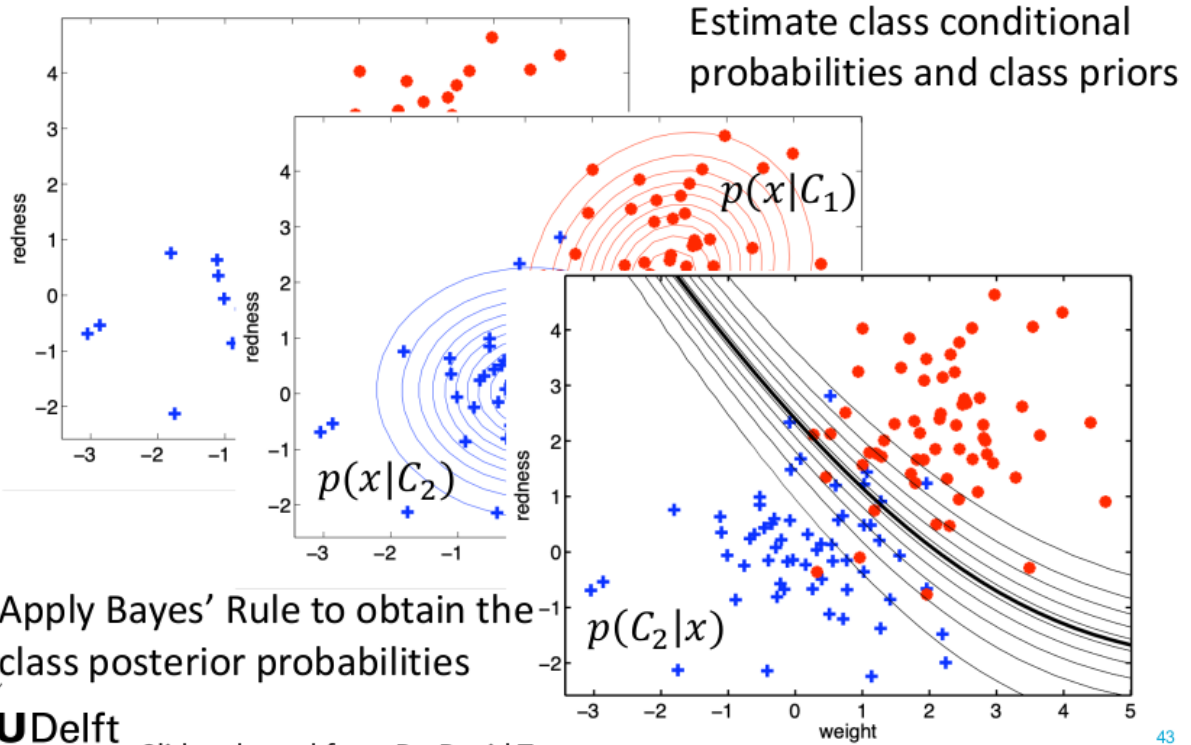
- Depending on the class conditional probability densities, complicated decision boundaries can appear

Missing decision boundary

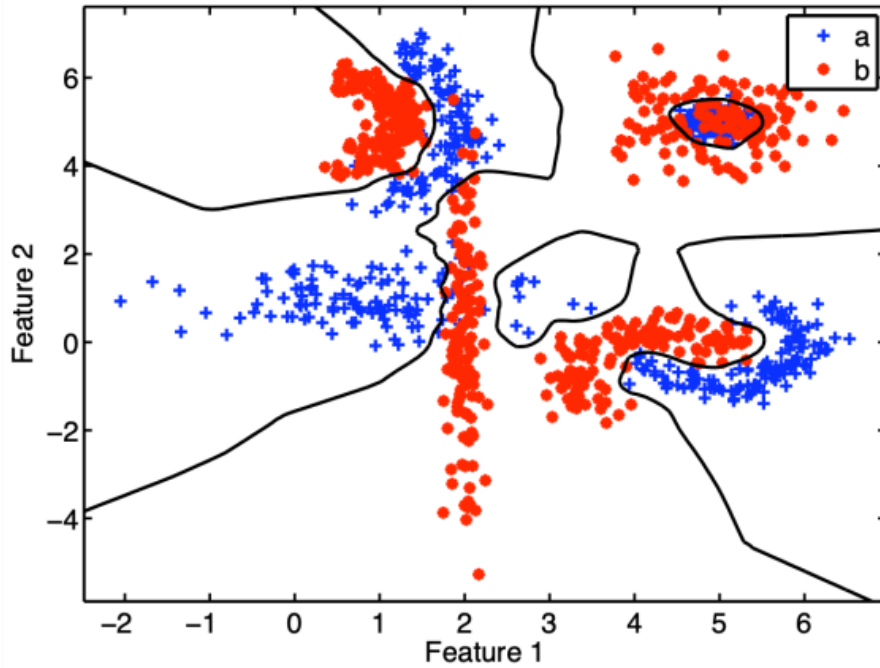


- A class can be too small (class prior is low) or too dispersed, that no objects are assigned to that class

Higher (2)-dimensional feature space



Multi-modal distributions



- Depending on the class distributions, the decision boundary can have arbitrary shapes

Rewriting the classifier for convenience

Description of a classifier

There are several ways to describe a classifier:

- if $p(y_1|\mathbf{x}) > p(y_2|\mathbf{x})$ then assign to y_1
otherwise y_2
- if $p(y_1|\mathbf{x}) - p(y_2|\mathbf{x}) > 0$ then assign to y_1
- or $\frac{p(y_1|\mathbf{x})}{p(y_2|\mathbf{x})} > 1$
- or $\log(p(y_1|\mathbf{x})) - \log(p(y_2|\mathbf{x})) > 0$

- Classifiers are often rewritten i.e. such that the derivative is easier to compute

Model

- Note on the probability function for the conditional probability $p(x|C_k)$ (called class conditional probability): The shape of the distribution (utterly based on the training data) is based on the model used in the training phase.

- We covered a sample **linear model** but more details on selecting a model come in next sections
- During training estimate the model parameters such that the example objects fit well: maximum likelihood estimators

Classification error

Bayes Error

- Even if we know the true distribution, errors predicting y from x will occur because the posteriors $p(y|x)$ are often not exactly 0 or 1

➔ lowest possible prediction error

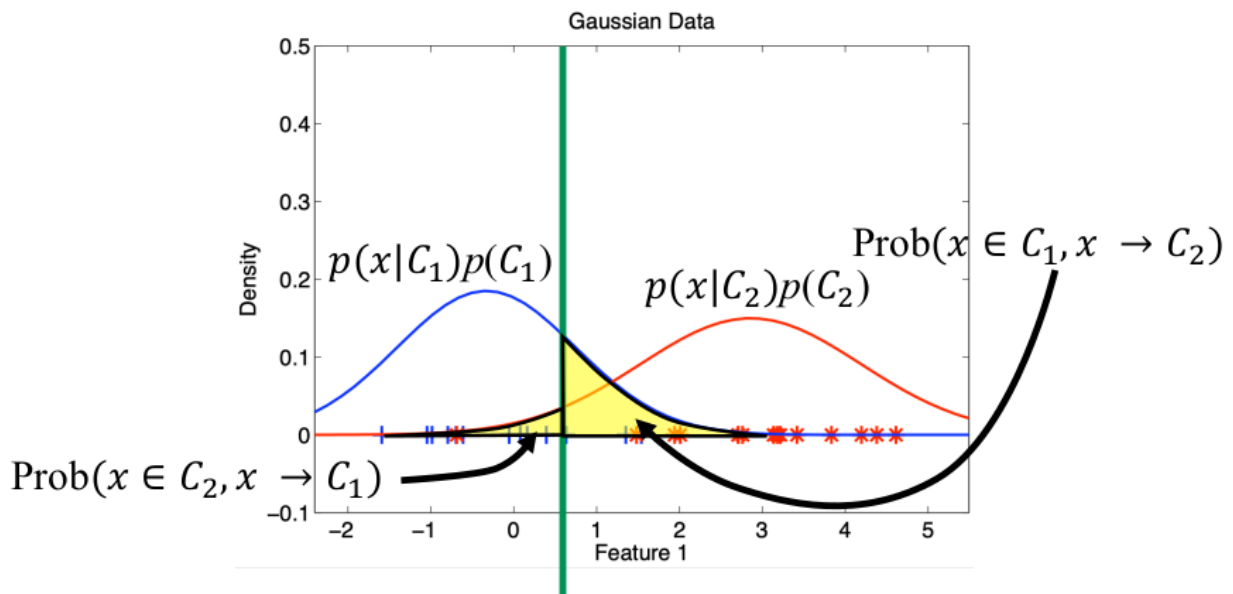
== Bayes' error

== irreducible error

- This is due to noise in the data as well as how well it can be naturally split into different categories (classifier quality)

Classification error

- The error: $P(error) = \sum_{i=1}^C P(error|C_i)P(C_i)$

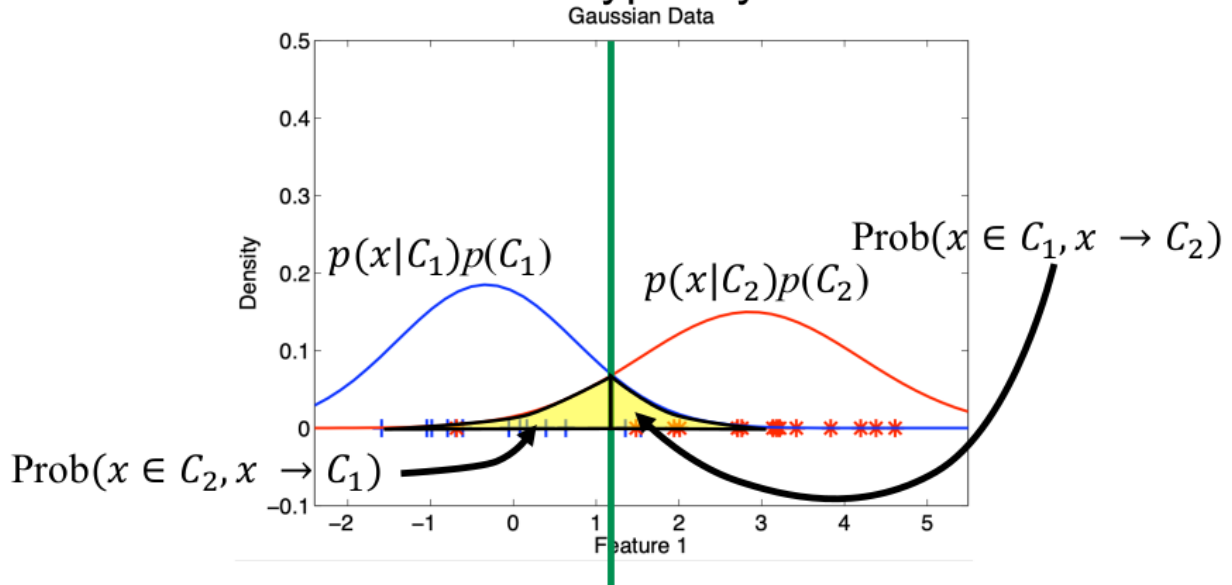


- The yellow part are actually errors (but we can shift the decision boundary to one side to minimize the errors)

- $Prob(x \in C_1, x \rightarrow C_2)$ is read as probability of x being assigned to C_2 but actually being C_1 .
- The total classification error is basically adding up the yellow spaces, or the weighted sum of errors

Bayes Error ϵ^*

- The **minimum** error: typically $> 0!$



- Bayes error is the minimum attainable error (not necessarily intersection of 2 distributions but just the place where the yellow area is the smallest)
- In practice we do not have the true distributions, and we cannot obtain them
- The Bayes error does not depend on the classification rule that you apply, but on the distribution of the data
- In general you cannot compute the bayes error
 - You do not know the true class conditional probabilities
 - The high dimensional integrals are very complicated

Missclassification costs

- We want to make as few errors as possible with the assignment of x to class C
- Error: x is assigned to C_1 but should have been assigned to C_2 and viceversa
- Sometimes missclassification of class A to class B is much more costly than missclassification of class B to class A
 - We can add a loss value for false positives and false negatives respectively
- Then the goal becomes to minimize the loss function

Density-based classifiers

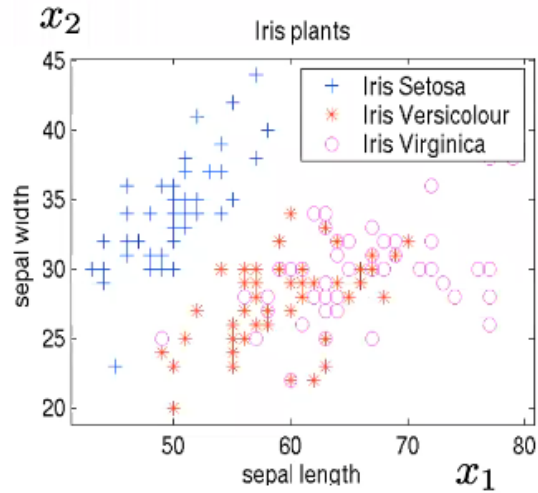
- Most basic fundamental assumption for a machine learning algorithm: goal is to estimate the "posterior probability" $P(\text{label}|\text{object})$ (if we already know the real posterior probability, then the job is already done and we just jump onto drawing decision boundaries)
- First we encode the object into a set of features
 - Height
 - Weight
 - Color
 - etc
- The "feature space" has as many dimensions as there are features

Objects in feature space

- We can interpret the measurements as a vector in a vector space:

$$\mathbf{x} = (x_1, x_2, \dots, x_p)$$

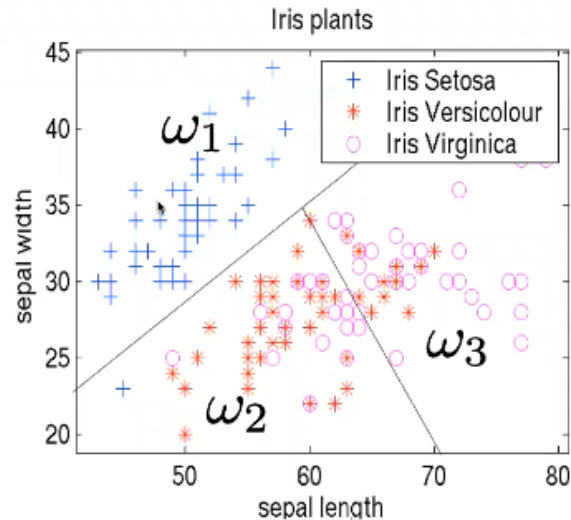
- This originates, in principle, from a probability density over the whole feature space $p(\mathbf{x}, y)$



- The plotted crosses/circles/stars are the training data of this classification task of 3 possible labels (setosa, versicolour, virginica) and 2 features (sepal width and sepal length)
 - This can be plotted because we have $p(x,y)$ (joint probability of x and y) which is explicitly derived from the training set that has labels
- Then we can draw decision boundaries

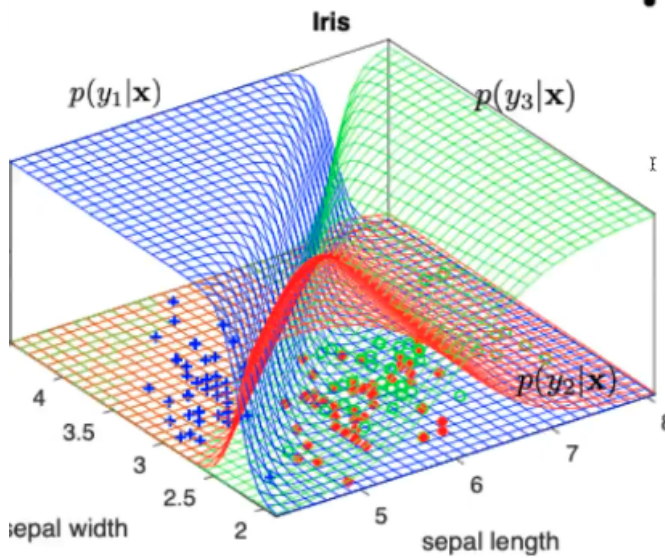
Classification

- Given labeled data: \mathbf{x}
- Assign to each object a class label ω
- In effect splits the feature space in separate regions



- The underlying criteria to draw this lines is that the posterior probability of an object belonging to that class is higher than to all the other classes (individually)
 - $p(y_1|x) > p(y_2|x)$
 - w is also expressed as y to indicate the class
- We can use computers to calculate the probability density functions for each label/class/ y/w y_n for a given object x with k features (k denoting the dimension of the model).
 - For the 2 features flowers training set we get:

Output of the model



- For each object in the feature space, we should find:

$$p(y|\mathbf{x})$$

In practice, we approximate: $\hat{p}(y|\mathbf{x})$

or we fit a function:

$$f(\mathbf{x})$$

- For each point in the feature space we should be able to get n posterior probabilities (1 for each class) and all of them should sum up to 1

Estimating probability (density) functions

- It is very hard to calculate posterior class probability distributions (shape, that they all add up to one, etc)
 - The class conditional distribution is also hard, but that integral has been studied and estimated for much more time and there have been satisfying solutions. (It is thus slightly less hard)
- All estimations are eventually computed from the training set/sample/examples
- Instead of directly estimating the posterior probability we'll use baye's rule to estimate an equivalent distribution based on the baye's rule terms

- $$p(y|x) = \frac{p(x|y)p(y)}{p(x)}$$

- $p(x|y)$ is the class conditional distribution
 - Probability distribution of a feature vector given that it belongs to a certain class: This is equal to $p(x,y)/p(y)$ (which we both know)
 - If we go to the iris flower example, $p(x|iris\ setosa)$ is just the integral fuction that wraps approximately all of the blue data points underneath the curve
- $p(y)$ is the class prior
 - it's a constant (for each discrete class is different), easy to compute (class occurence / total occures)
- $p(x)$ is the unconditional data distribution
 - What's the distribution of x regardless of the class

Data distribution

- In Bayes rule you see $p(\mathbf{x})$. How to get it?
- You can explicitly compute it:

$$p(\mathbf{x}) = p(\mathbf{x}|y_1)p(y_1) + p(\mathbf{x}|y_2)p(y_2)$$

- But if you just find the largest posterior:

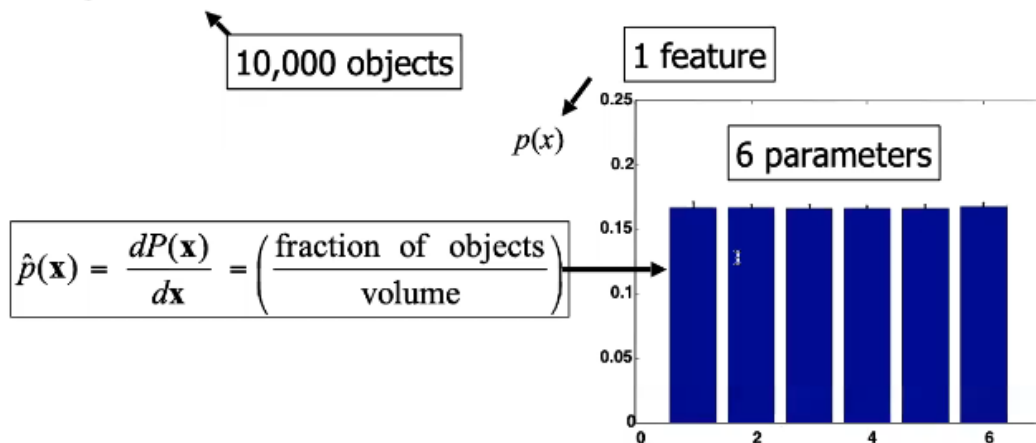
$$\frac{p(\mathbf{x}|y_1)p(y_1)}{\cancel{p(\mathbf{x})}} > \frac{p(\mathbf{x}|y_2)p(y_2)}{\cancel{p(\mathbf{x})}}$$

- Since $p(x)$ is a common term in all class comparisons, we can remove it from the inequation, so we just have to compute to terms
- To approximate $p(x, y)p(y)$ which is a poor man's version of $p(y|x)$ (only within the context of comparing classes, as $p(x)$ is defenetly missing in the first one) we will consider:
 - Discriminative and generative models
 - Parametric and nonparametric models

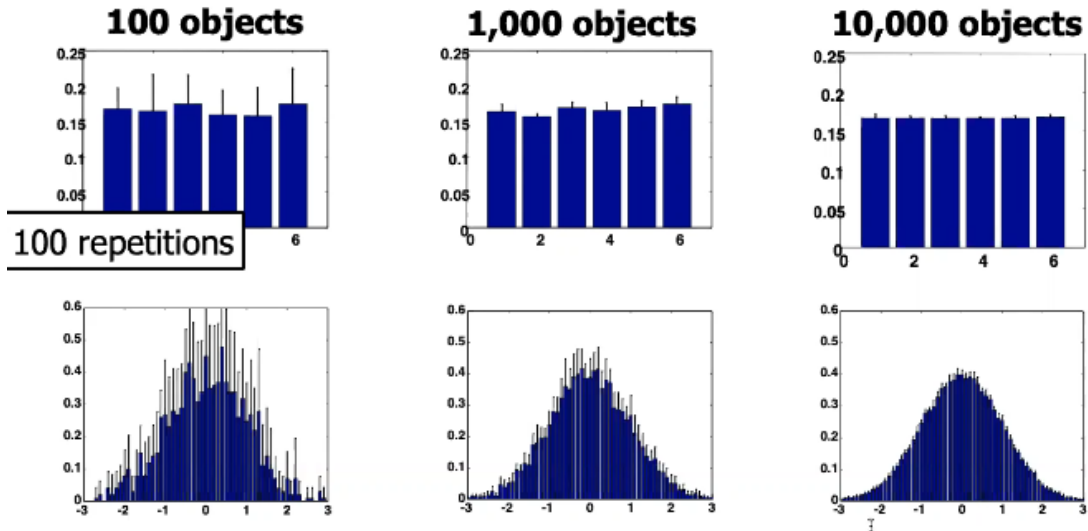
Histogram-based density estimation

Histogram-based Density Estimation

- Relatively simple approach : approximate density by histogram
E.g. 10,000 throws of a dice



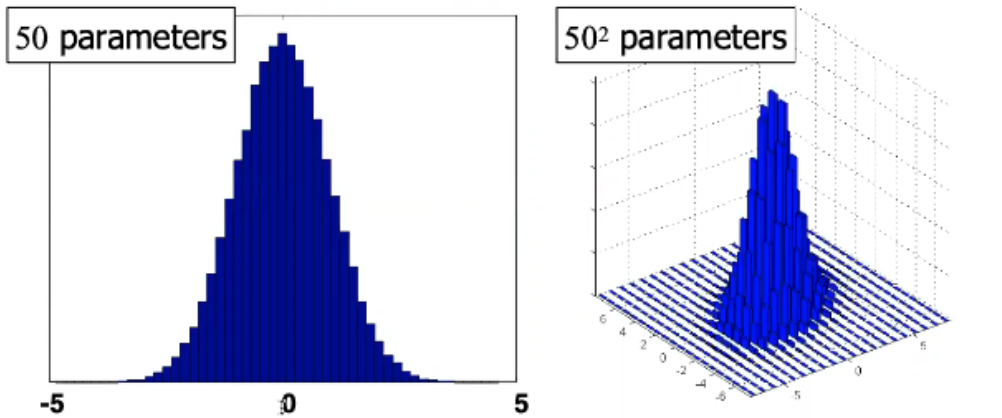
- The problem is to get enough data points to reach the central limit theorem



- The thumbrule is to use 1k training objects to the power of the number of features.
- parameter = bin = rectangle

For 1-dimensional data,
± 1000 objects needed

For p -dimensional data,
± 1000 ^{p} objects needed



• Unworkable for $p > 2$ features

- However this is unworkable for probability density functions with more than 2 features as the require sample size increases dramitically with the number of features

Curse of dimensionality

- There is a trade-off between having more features (and thus distinguishing objects better) and having to estimate density functions that require exponentially more training objects
- The number of parameters (bins) increases with the number of features as well

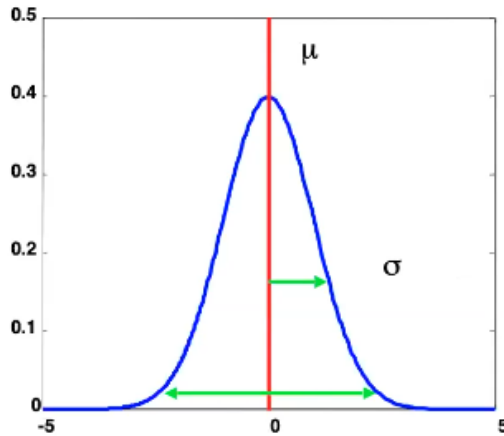
Parametric density estimation

- In a parametric model you assume you know the shape of the full distribution (i.e. it should be round, normally distributed, etc.) and the only thing you need to estimate is the paramaters of that well known established shape
- Picking it from where we left it, we're gonna use parametric modeling & estimation for the class conditional probability (that is, not the class posterior, but $p(x|y)$), which we later combine with $p(y)$ to define the classifier: $p(x|y_1)p(y_1) > p(x|y_2)p(y_2)$

Assume Gaussian (Bell shaped) distribution

- By using a known shape we can just focus on estimating it's parameters, for a gaussian that'd be the mean and standard deviation
- However, to make the model accurate, we should choose features that are also distributed in the same way in real life, such as height and weight

Gaussian Distribution



- Normal distribution = Gaussian distribution
- Standard normal distribution: $\mu = 0, \sigma^2 = 1$
- 95% of data between $[\mu - 2\sigma, \mu + 2\sigma]$ [in 1D!]

- 1D :
$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2} \frac{(x - \mu)^2}{\sigma^2}\right)$$

- We choose gaussian distribution because it is the one from the central limit theorem
 - sums of large numbers of independent identically distributed random variables will have a gaussian distribution
- It occurs in real life
- It has few parameters (2 (mean and variance) as opposed as lots of bins for histograms)
 - These parameters are easy to estimate

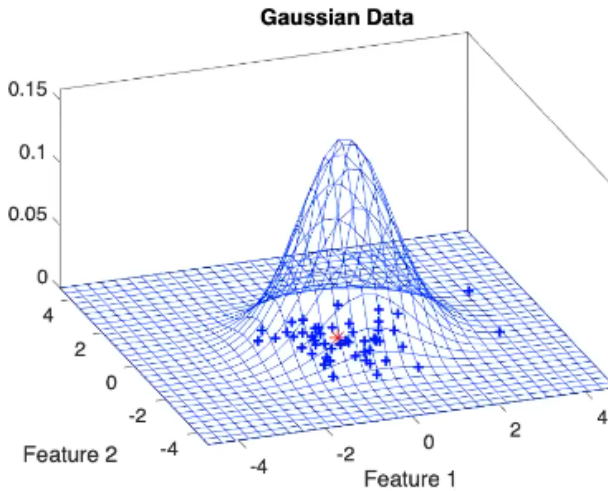
Multivariate Gaussians

- p - dimensional density :

$$p(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^p \det(\Sigma)}} \exp\left(-\frac{1}{2} (\mathbf{x} - \mu)^T \Sigma^{-1} (\mathbf{x} - \mu)\right)$$

- Both \mathbf{x} and μ are vectors (aka feature vector and mean feature vector) and Σ is a covariance matrix (variance for higher dimensions)

Gaussian Distribution 2D



Mean:

$$\mu = \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}$$

Covariance matrix:

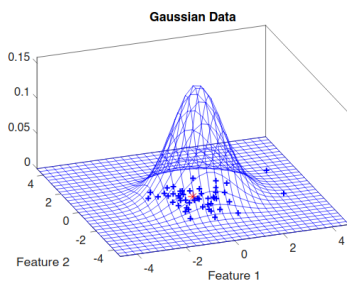
$$\Sigma = \begin{bmatrix} \sigma_1^2 & \rho_{12}\sigma_1\sigma_2 \\ \rho_{12}\sigma_1\sigma_2 & \sigma_2^2 \end{bmatrix}$$

$$p(\mathbf{x}) = \frac{1}{\sqrt{2\pi^p \det(\Sigma)}} \exp\left(-\frac{1}{2}(\mathbf{x} - \mu)^T \Sigma^{-1}(\mathbf{x} - \mu)\right)$$

- The red star is the mean vector μ
- The covariance matrix is $k \times k$ with k being the number of dimensions (features)
- On the diagonal we find the variances for each of the features
- ρ_{12} is the correlation coefficient between features 1 and 2

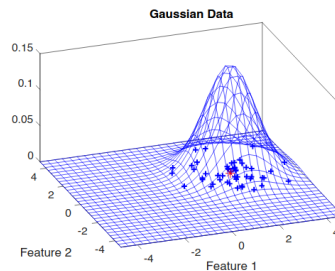
Covariance matrix and mean vector in a 2D gaussian distribution

- The mean vector determines the center of the distribution
- The covariance matrix determines the shape in 2 ways
 - The diagonal matrix variances determine the breadth on their respective axis
 - The correlation coefficients determine the "rotation" (it's the "slope" if we watched the data points from the top)



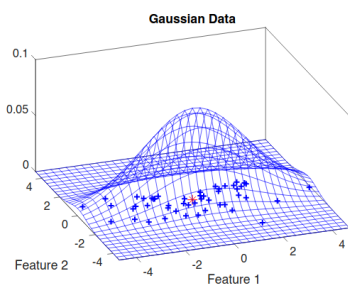
$$\mu = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\Sigma = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$



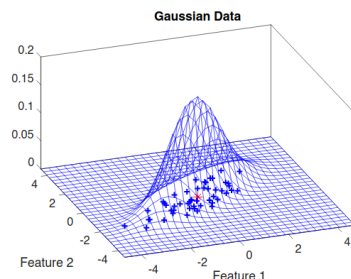
$$\mu = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

$$\Sigma = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$



$$\mu = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

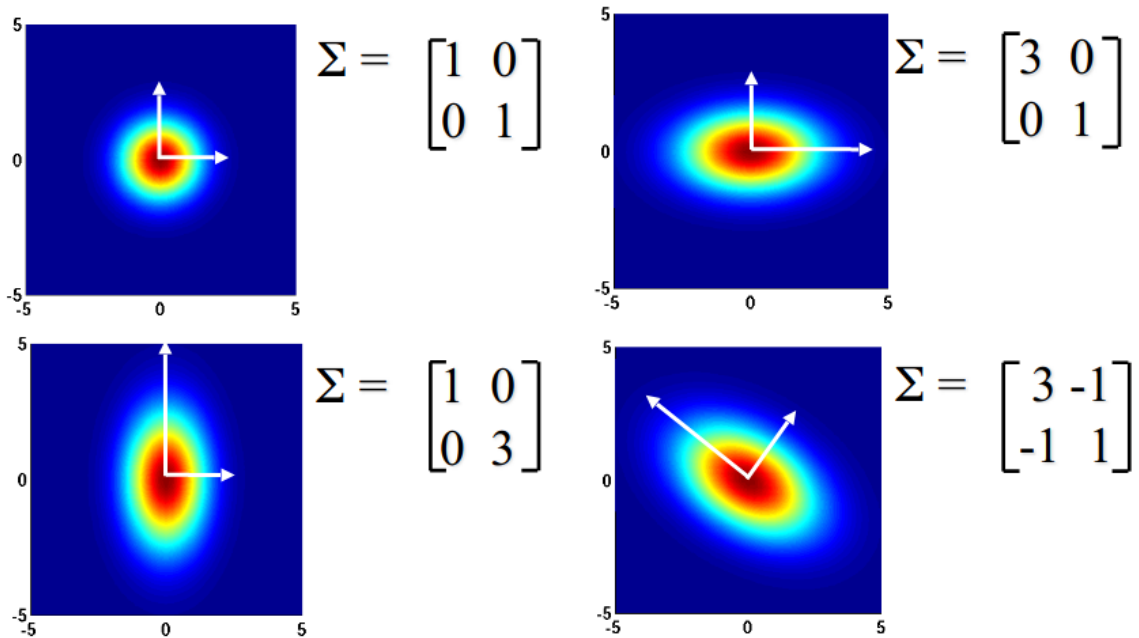
$$\Sigma = \begin{bmatrix} 4 & 0 \\ 0 & 1 \end{bmatrix}$$



$$\mu = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\Sigma = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$$

- Note that in the pictures above the mean feature vector is expressed in one column, but in CSE2510 we will express the design matrix in rows for each object. This doesn't change the fact that we'll use the feature vector as a vector (thus $n \times 1$) (as in the picture)
- Top view of some gaussian distributions with 0,0 mean vector



Maximum likelihood estimates (for the mean and covariance matrix)

- Note that the hat denotes estimation variable
- For the mean:

-

$$\hat{\mu} = \frac{1}{n} \sum_{i=1}^n x_i$$

- just sum all feature vectors and divide all entries by the number of objects in the data set

- For the covariance matrix:

-

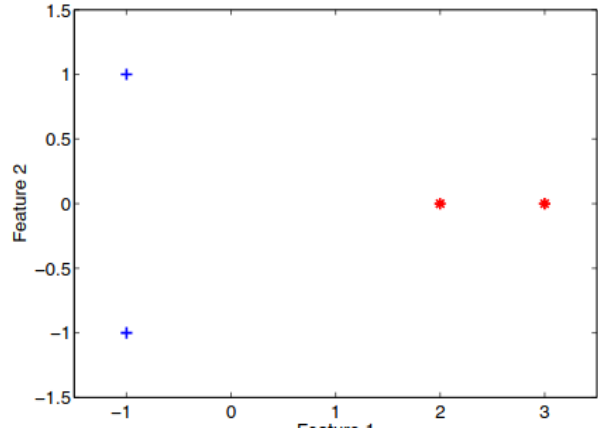
$$\hat{\Sigma} = \frac{1}{n} \sum_{i=1}^n ((x_i - \hat{\mu})(x_i - \hat{\mu})^T)$$

- T denotes the transpose
- Multiplying an $n \times 1$ matrix with its transpose makes a $n \times n$ matrix
- They are estimations because they are based on the training data and may not be exactly the same as the real parameters
- If you recall the gaussian model for p -dimensions, we need the inverse of the covariance matrix.
 - The covariance matrix will be invertible only with at least $0.5p(p + 1)$ data points
 - The number of data needed increases quadratically
 - For a 32 by 32 pixels images you need at least 32^2 examples per class

No inverse...

- One of the variances is 0
- We don't have enough data
- The Gaussian density is not defined

- Boohoo



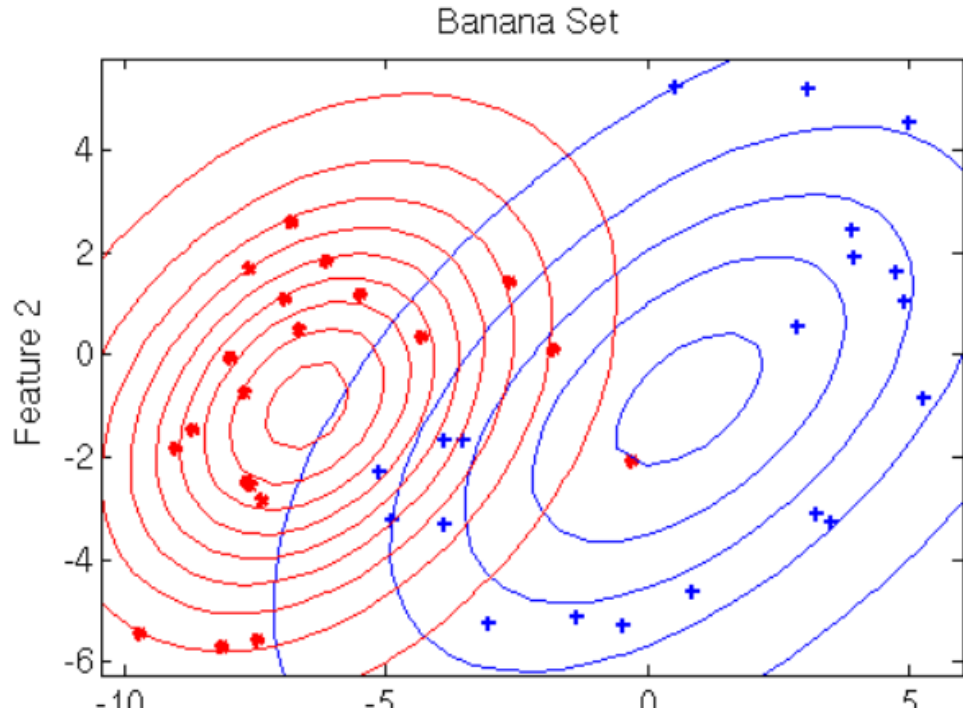
Gaussian density based classifiers

- As discussed earlier, the classifier is to classify to class 1 when $p(y_1|x) > p(y_2|x)$ or the proportionally equivalent inequation $p(x|y_1)p(y_1) > p(x|y_2)p(y_2)$
- The model behind either posterior or class conditional density distribution is the gauss bell shape distribution
 - For each class y we have a gaussian distribution
 -

$$\hat{p}(\mathbf{x}|y) = \frac{1}{\sqrt{(2\pi)^p \det(\hat{\Sigma}_y)}} \exp\left(-\frac{1}{2}(\mathbf{x} - \hat{\mu}_y)^T \hat{\Sigma}_y^{-1} (\mathbf{x} - \hat{\mu}_y)\right)$$

- We have to estimate the parameters $\hat{\mu}$ and $\hat{\Sigma}$
- recall that all density functions derived from the training set are only estimations of the real world distributions, and hence the "hat"

- A single Gaussian distribution on each class:



Two-Class case

- We can use the log of the conditionals as the classifier.
- Instead of using an inequality, we just do the subtraction and if the number is positive it goes to the class on the left operand class and if it's negative it goes to the right operand class.
- The function is called the "discriminant" and it's a quadratic classifier because the decision boundary is a quadratic function of x

- Define the discriminant

$$f(\mathbf{x}) = \log p(y_1|\mathbf{x}) - \log p(y_2|\mathbf{x})$$

- Rewriting this, one gets

$$f(\mathbf{x}) = \mathbf{x}^T \mathbf{W} \mathbf{x} + \mathbf{w}^T \mathbf{x} + w_0$$

- Taking the logs encapsulates the gaussian distribution inside a log, whose exponent is cancelled out, (the other term of the product is a constant that we can ignore) and we're left with a quadratic distribution in terms of the feature vector x

Class Posterior Probability Gaussian

- Combining

$$\hat{p}(\mathbf{x}|y) = \frac{1}{\sqrt{(2\pi)^p \det(\hat{\Sigma}_y)}} \exp\left(-\frac{1}{2}(\mathbf{x} - \hat{\mu}_y)^T \hat{\Sigma}_y^{-1}(\mathbf{x} - \hat{\mu}_y)\right)$$

$$p(y|\mathbf{x}) = \frac{p(\mathbf{x}|y)p(y)}{p(\mathbf{x})}$$

We can derive for $\log(p(y_i|\mathbf{x}))$:

$$\log(\hat{p}(y_i|\mathbf{x})) = -\frac{p}{2} \log(2\pi) - \frac{1}{2} \log(\det \Sigma_i)$$

$$- \frac{1}{2}(\mathbf{x} - \mu_i)^T \Sigma_i^{-1}(\mathbf{x} - \mu_i) + \log p(y_i) - \log p(\mathbf{x})$$

- $p(\mathbf{x})$ is independent of the classes, it can be dropped:

$$g_i(\mathbf{x}) = -\frac{1}{2} \log(\det \Sigma_i) - \frac{1}{2}(\mathbf{x} - \mu_i)^T \Sigma_i^{-1}(\mathbf{x} - \mu_i) + \log p(y_i)$$

- You can work out the square term
- Classifier becomes :

Assign \mathbf{x} to class y_i when for all $i \neq j$:

$$g_i(\mathbf{x}) - g_j(\mathbf{x}) > 0$$

- General form for a two-class classifier:

$$f(\mathbf{x}) = \mathbf{x}^T \mathbf{W} \mathbf{x} + \mathbf{w}^T \mathbf{x} + w_0$$

- with

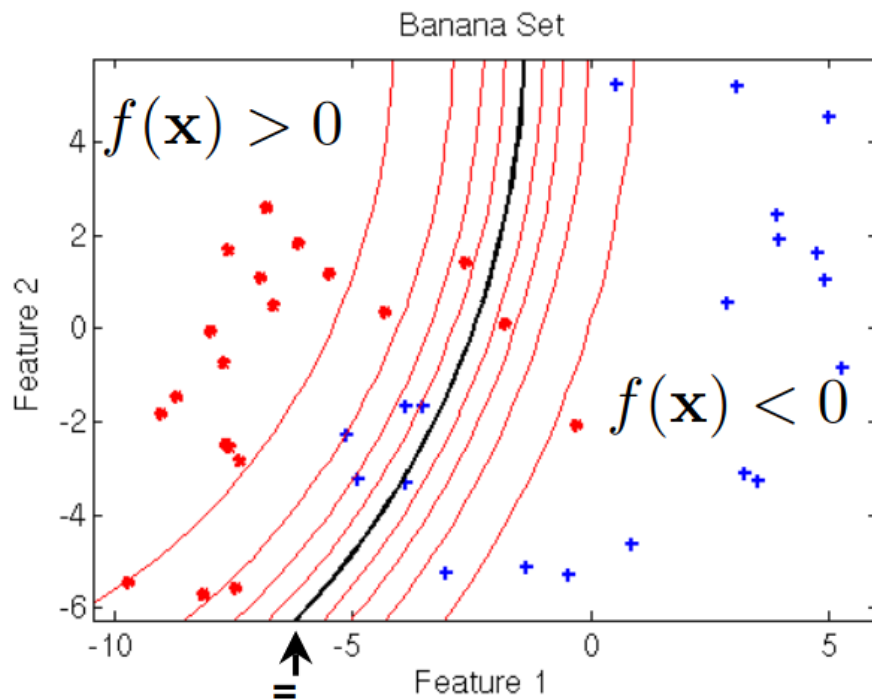
$$\mathbf{W} = \frac{1}{2} (\Sigma_2^{-1} - \Sigma_1^{-1})$$

$$\mathbf{w}^T = \mu_1^T \Sigma_1^{-1} - \mu_2^T \Sigma_2^{-1}$$

$$w_0 = -\frac{1}{2} \log \det \Sigma_1 - \frac{1}{2} \mu_1^T \Sigma_1^{-1} \mu_1 + \log p(y_1) \\ + \frac{1}{2} \log \det \Sigma_2 + \frac{1}{2} \mu_2^T \Sigma_2^{-1} \mu_2 - \log p(y_2)$$

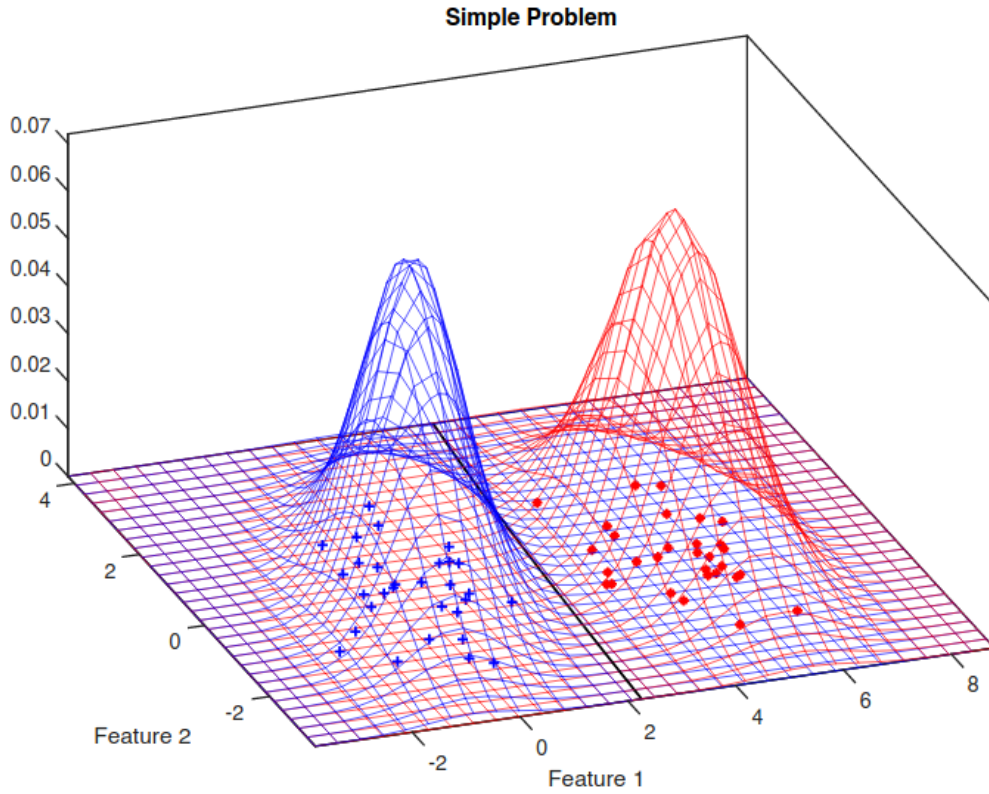
- The black line of the quadratic classifier below is the set of \mathbf{x} features that have a 0 output from the discriminant function (neither one class nor the other, the decision boundary)

Quadratic Classifier on Banana Data



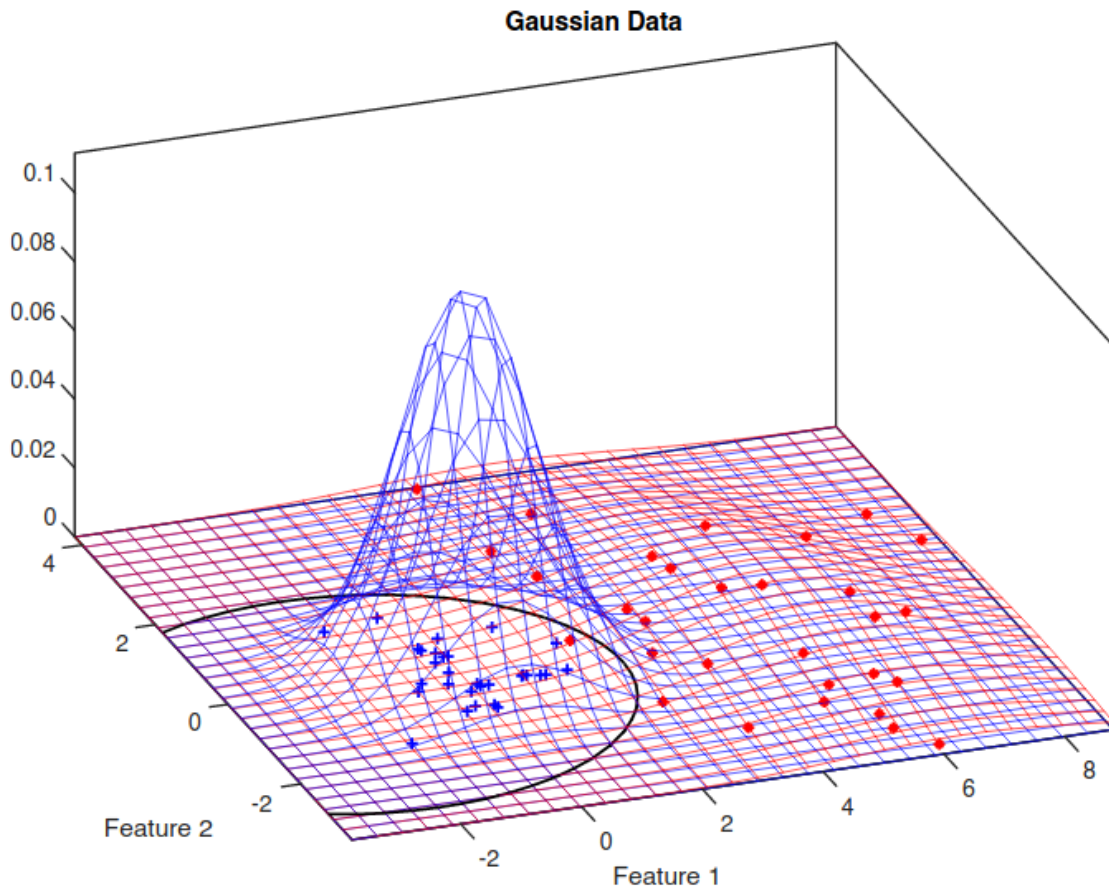
- The quadratic (hiperbola) discriminant function can take multiple shapes (within quadratic nature) depending on the shapes of the distributions of the classes (depending on the covariances)

(Almost) linear decision boundary



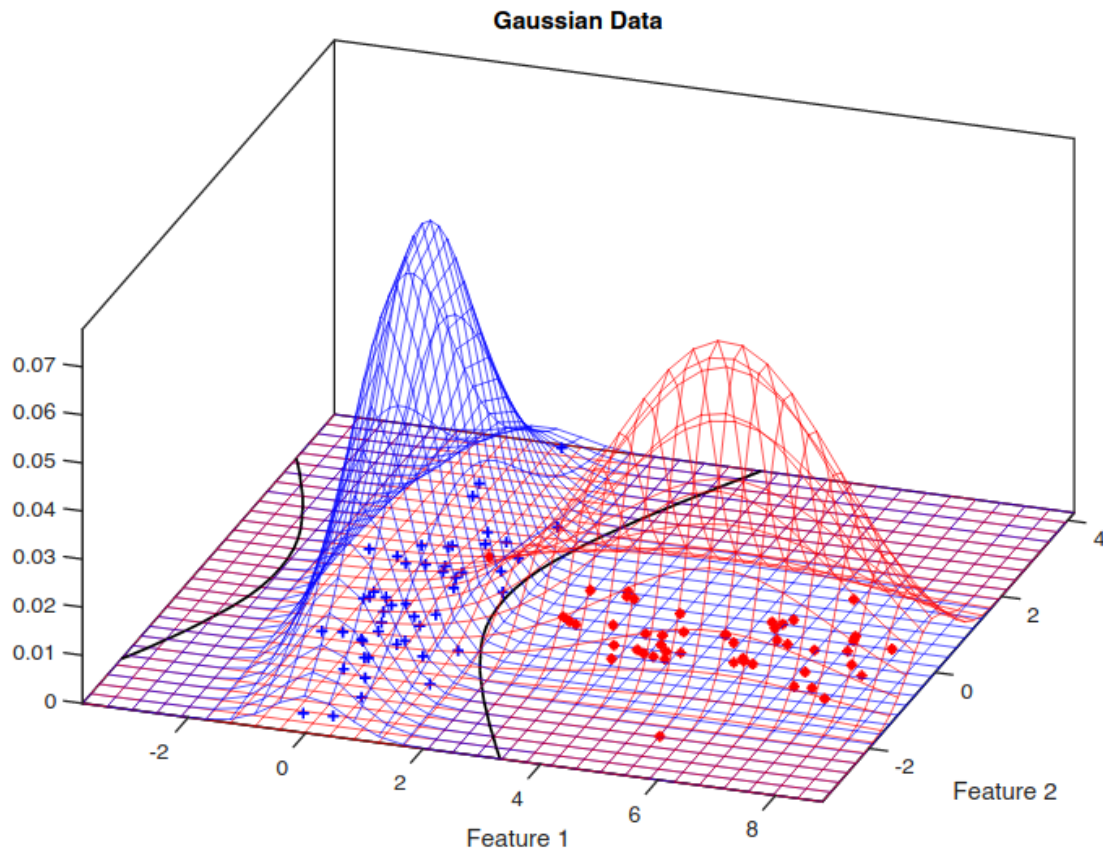
- Linear when the covariances are the same (and the means are different)

Circular decision boundary



- Happens when one of the classes is spread across the feature space while the other class is concentrated in a smaller area

Hyperbolic decision boundary



- 2 decision boundaries described by a single function

Fixing no inverse covariance matrix for Two-classes case

- Assume that gaussian shape of the other class is the same (i.e. just different mean but same covariance)

Estimating Covariance Matrices

- For quadratic classifier need to estimate covariances, e.g., by

$$\hat{\Sigma}_k = \frac{1}{n} \sum_{i=1}^n (\mathbf{x}_i - \hat{\mu}_k)(\mathbf{x}_i - \hat{\mu}_k)^T$$

for each of the classes

- When there is insufficient data, this covariance matrix cannot be inverted
- Alternative : average over all class covariance matrices

$$\hat{\Sigma}_k = \frac{1}{C} \sum_{k=1}^C \hat{\Sigma}_k$$

- This allows us to use all our data from all classes to estimate 1 single covariance matrix
- The $1/C$ times sum of C elements is just the average of the term inside the summation
- This makes the classifier a linear function since the \mathbf{W} term is multiplied by 0 by assuming that all class covariances are the same
 - We are then just left with the constant w_0 and with $w^T x$

- General form for a two-class classifier:

$$f(\mathbf{x}) = \mathbf{x}^T \mathbf{W} \mathbf{x} + \mathbf{w}^T \mathbf{x} + w_0$$

- with

$$\mathbf{W} = \frac{1}{2} (\Sigma_2^{-1} - \Sigma_1^{-1})$$

$$\mathbf{w}^T = \mu_1^T \Sigma_1^{-1} - \mu_2^T \Sigma_2^{-1}$$

$$w_0 = -\frac{1}{2} \log \det \Sigma_1 - \frac{1}{2} \mu_1^T \Sigma_1^{-1} \mu_1 + \log p(y_1) \\ + \frac{1}{2} \log \det \Sigma_2 + \frac{1}{2} \mu_2^T \Sigma_2^{-1} \mu_2 - \log p(y_2)$$

- We end up with the LDA (linear discriminant analysis), assumes both distributions to have the same covariance matrix

The Two-Class Case: LDA

- Define the discriminant

$$f(\mathbf{x}) = \log p(y_1|\mathbf{x}) - \log p(y_2|\mathbf{x})$$

- We get

$$f(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + w_0$$

with

$$\mathbf{w} = \hat{\Sigma}^{-1}(\hat{\mu}_1 - \hat{\mu}_2)$$

$$w_0 = \frac{1}{2}\hat{\mu}_2^T \hat{\Sigma}^{-1} \hat{\mu}_2 - \frac{1}{2}\hat{\mu}_1^T \hat{\Sigma}^{-1} \hat{\mu}_1 + \log \frac{p(y_1)}{p(y_2)}$$

- It's the classic (now rather old) approach
- It's simple and fast
- Doesn't need enough data
- Works well for simple data sets
- Never the best, but always in the top best when multiple features are used

Linear discriminant

- Let us assume that the decision boundary can be described by:

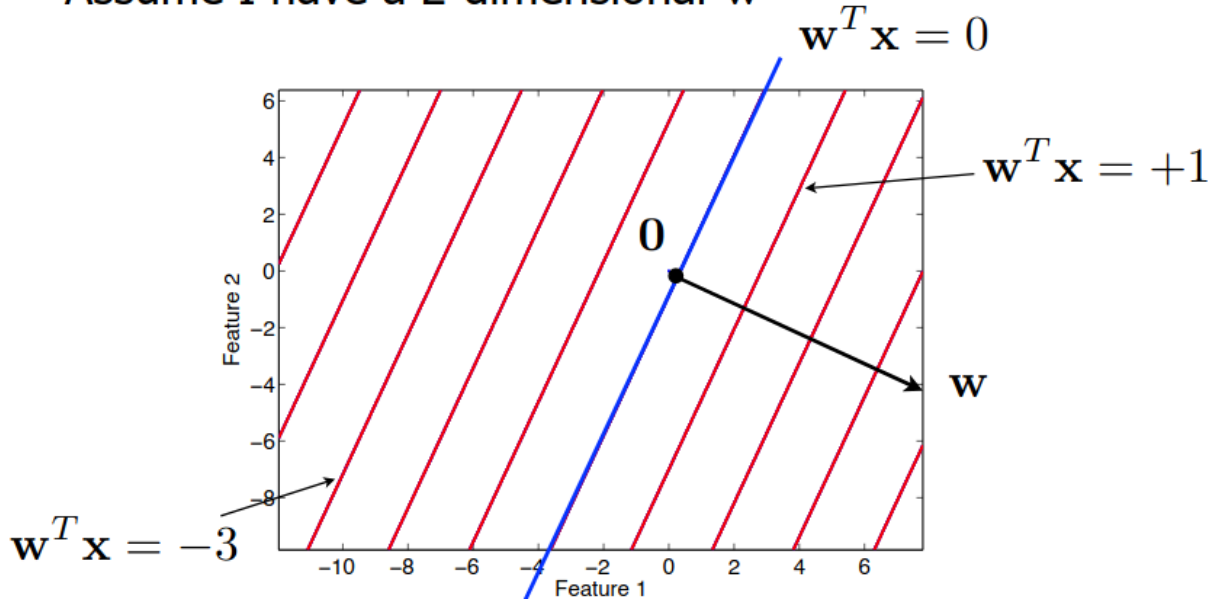
$$g(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + w_0 = 0$$

- Weight vector \mathbf{w} and bias term (offset) w_0
- Classify:

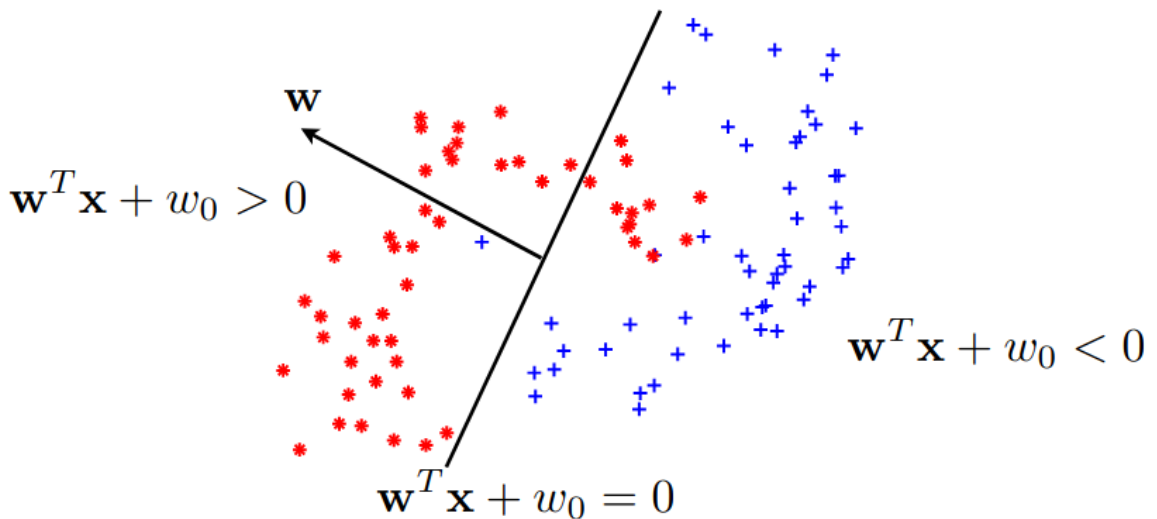
$$\text{classify } \mathbf{x} \text{ to } \begin{cases} y_1 & \text{if } \mathbf{w}^T \mathbf{x} + w_0 \geq 0 \\ y_2 & \text{if } \mathbf{w}^T \mathbf{x} + w_0 < 0 \end{cases}$$

- In the most general sense, this is called linear discriminant analysis
 - The way \mathbf{w} was defined makes it a vector that is perpendicular to the decision boundary
 - w_0 is a constant that shifts the decision boundary
 - A positive constant shifts the decision boundary to the left, and a negative one to the right

Assume I have a 2-dimensional w



- Recall that w and w_0 can be both found just by having the mean and variance parameters of the distribution



- Classifier is a linear function of the features
- The classification depends if the weighted sum of the features is above or below 0
- This dot product of w and x allows us to observe that certain "weights" (entries) of w have larger impact on the dot product outcome (by being larger), thus we can observe from w which features are important for the classification.
 - Those equal or close to zero could be removed

Rewriting the bias term (w_0)

- Quite often you see $g(\mathbf{x}) = \mathbf{w}^T \mathbf{x} > 0$

instead of

$$g(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + w_0 > 0$$

- No problem, if you (re-)define the feature vector as:

(homogeneous coordinates)

- Then:
$$\tilde{\mathbf{x}} = \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix}$$

$$g(\mathbf{x}) = [\mathbf{w}^T \ w_0] \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} = \tilde{\mathbf{w}}^T \tilde{\mathbf{x}}$$

- Sometimes called homogenous coordinates

Nearest Mean Classifier (NMC)

No estimated covariance matrix:

- When the number of features far exceeds the sample size it becomes hard to estimate even the average covariance matrix
- Instead one can assume that all features have the same variance and all are uncorrelated
- Thus the covariance matrix is just a diagonal matrix with the same values, the variance of all features:

- Quite often you see $g(\mathbf{x}) = \mathbf{w}^T \mathbf{x} > 0$

instead of

$$g(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + w_0 > 0$$

- No problem, if you (re-)define the feature vector as:

(homogeneous coordinates)

- Then:
$$\tilde{\mathbf{x}} = \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix}$$

$$g(\mathbf{x}) = [\mathbf{w}^T \ w_0] \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} = \tilde{\mathbf{w}}^T \tilde{\mathbf{x}}$$

Nearest Mean Classifier

- Define the discriminant :

$$f(\mathbf{x}) = \log p(y_1|\mathbf{x}) - \log p(y_2|\mathbf{x})$$

- We get

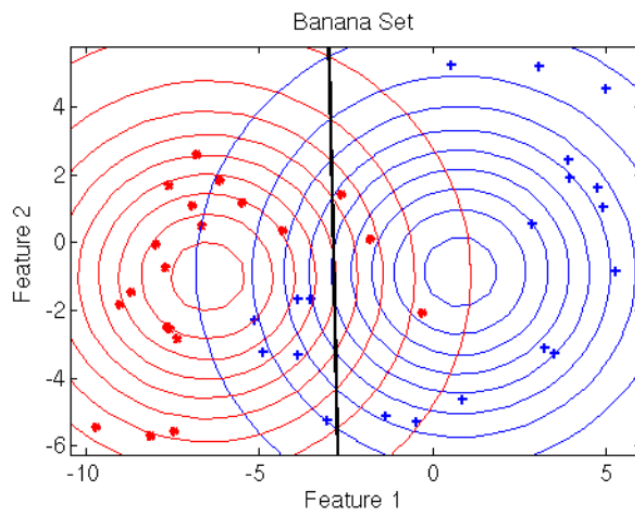
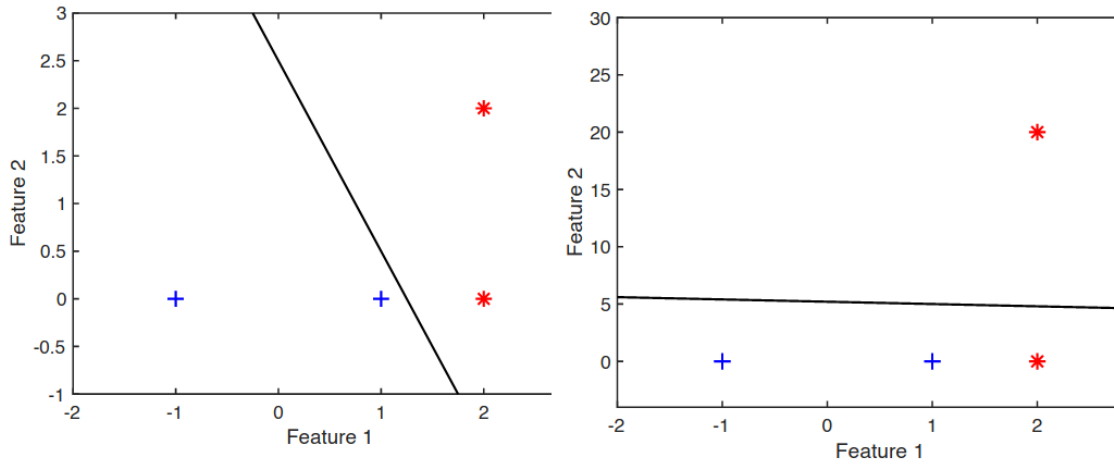
$$f(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + w_0$$

with

$$\mathbf{w} = \hat{\mu}_1 - \hat{\mu}_2$$

$$w_0 = \frac{1}{2} \hat{\mu}_2^T \hat{\mu}_2 - \frac{1}{2} \hat{\mu}_1^T \hat{\mu}_1 + \sigma^2 \log \frac{p(y_1)}{p(y_2)}$$

- Again a linear classifier, but it only uses distance to the mean of each of the classes : nearest mean classifier



Feature scaling

- When a classifier depends on euclidian distances, scaling of the features matter
- Changing the scaling can improve/deteriorate the classifier (check the previous NMC example with different scales)

- A good practice is to standardize the features (objects) to Z scores (how many standard deviations away is a feature from the mean), here they are denoted as

$$\tilde{x} = \frac{x - \mu}{\sigma}$$

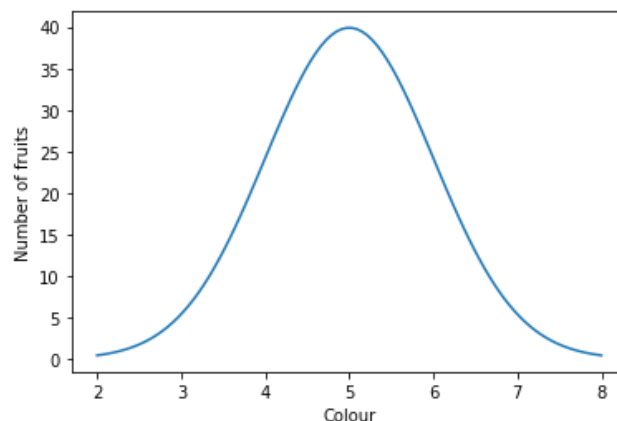
Model complexity and available data trade-off

- Using plug-in Bayes' rule with normal distribution for every class can give rise to different classifiers. From more flexible/complex to more simple:
 - **Quadratic classifier:** Separate mean and covariance matrix per class
 - **Linear classifier:** Separate mean, equal covariance matrix per class
 - **Nearest mean classifier:** Separate mean, same diagonal covariance matrix per class
 - This is the only one from the list that suffers from non-standardize data points
- More flexible classifier needs more training data
- Simple classifiers still perform well in practice
- **Curse of dimensionality:** The more features the more training data required

Lab: Parametric Bayes classifier

Classification using Gaussian distributions

- Occurrences of data typically follow probability distributions that we know how to model.
- In this assignment, we will assume that **data has a normal distribution** and try to estimate the parameters of the assumed normal distribution to correctly fit our data, hence the name parametric classifiers.



- We will then use Bayes' rule to build a classifier based on the probability distribution.
- We will try to classify flowers from Fisher's Iris dataset into Iris setosa, versicolor xor virginica based on the length and width of the spals and petals of 150 flowers (3 classes, 4 features, 150 examples)



- This dataset is such a classic example that it is even included in machine learning libraries. The following code will load the dataset from scikit-learn (this was installed with conda) into the

variable iris

```
import numpy as np
from sklearn import datasets

iris = datasets.load_iris()
iris
```

Getting to know the data

- The dataset is stored as a **dictionary**, a data structure in Python that resembles a Java(script) object (map with keys and values)
- We can access items in the dictionary with a dot ., so we access the data and their target labels with `iris.data` and `iris.target`, these are both **NumPy arrays**.

```
print("First five flowers: \n", iris.data[:5, :])
print("Their labels: ", iris.target[:5])
print("And the label names: ", iris.target_names)

last_five_flowers = None
third_feature_only = None
first_ten_names = None

# START ANSWER
last_five_flowers = iris.data[len(iris.data)-5:len(iris.data)]
third_feature_only = iris.data[:,2]
first_ten_names = iris.target[:10]
# END ANSWER

setosa_flowers = None
versicolor_flowers = None
virginica_flowers = None

# START ANSWER
setosa_flowers = iris.data[np.where(iris.target == 0)]
versicolor_flowers = iris.data[np.where(iris.target == 1)]
virginica_flowers = iris.data[np.where(iris.target == 2)]
# END ANSWER

print("Last five flowers: \n", last_five_flowers)
print("Only the third feature: ", third_feature_only)
print("All label names: ", first_ten_names)

print("Class: ", iris.target_names[0], "; Items: \n", setosa_flowers)

assert last_five_flowers.shape == (5,4), "Expected a two dimensional array of shape (5,4)"
assert third_feature_only.shape == (150,), "Expected an array of shape (150,)"
assert first_ten_names.shape == (10,), "Expected an array of shape (10,)"

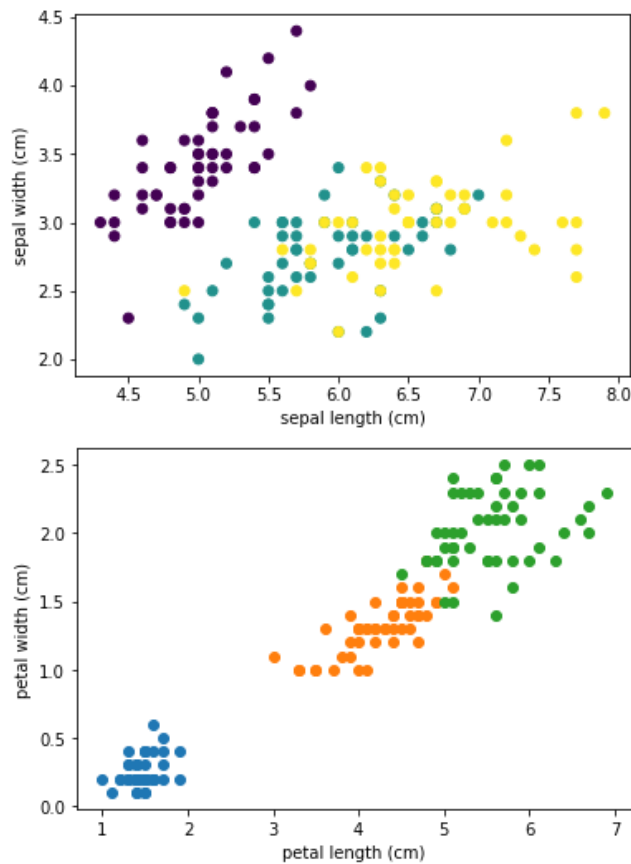
assert setosa_flowers.shape == (50,4), "Expected a two dimensional array of shape (50,4)"
assert versicolor_flowers.shape == (50,4), "Expected a two dimensional array of shape (50,4)"
assert virginica_flowers.shape == (50,4), "Expected a two dimensional array of shape (50,4)"
```

- To get an idea of the distribution of our data, we can make plots:

```
# From the Matplotlib library, import pyplot. We will refer to this library later as plt
# This is a widely used library that lets you create images and plot your data.
from matplotlib import pyplot as plt

# Create a scatterplot of the first two features, and use their labels as colour values.
plt.scatter(iris.data[:, 0], iris.data[:, 1], c=iris.target)
```

```
plt.xlabel(iris.feature_names[0])
plt.ylabel(iris.feature_names[1])
plt.show()
# Create a scatterplot of the third and fourth feature.
plt.scatter(iris.data[np.where(iris.target == 0), 2], iris.data[np.where(iris.target ==
plt.scatter(iris.data[np.where(iris.target == 1), 2], iris.data[np.where(iris.target ==
plt.scatter(iris.data[np.where(iris.target == 2), 2], iris.data[np.where(iris.target ==
plt.xlabel(iris.feature_names[2])
plt.ylabel(iris.feature_names[3])
plt.show()
```



Test sets

- Now that we have an idea what our dataset looks like, our goal is to create a model that will predict the class of each flower based on its features
- In order to evaluate how well the model fits, we will also need a separate test set where we can evaluate our final model on
- For this, we will split the data randomly in a train and test set.

```
from sklearn.model_selection import train_test_split #to split in train and test set

# Load the data and create the training and test sets
iris = datasets.load_iris()
# X is the feature vectors for the data points, and Y is the target (ground truth) class
# the iris.data and iris.target entries are randomly divided into training and test sets
X_train, X_test, Y_train, Y_test = train_test_split(iris.data, iris.target, test_size=0.

# Due to the randomness of the split, number of each flowers is not necessarily the same
# Separate the training dataset into the three flower types.
setosa_X_train = None
versicolor_X_train = None
virginica_X_train = None
# START ANSWER
setosa_X_train = X_train[np.where(Y_train == 0)]
versicolor_X_train = X_train[np.where(Y_train == 1)]
```

```

virginica_X_train = X_train[np.where(Y_train == 2)]
# END ANSWER

assert setosa_X_train.shape[0] != versicolor_X_train.shape[0]
assert setosa_X_train.shape[0] != virginica_X_train.shape[0]
assert versicolor_X_train.shape[0] != virginica_X_train.shape[0]

setosa_X_train.shape, versicolor_X_train.shape, virginica_X_train.shape

```

Univariate model

- The plots shows that sepal length and sepal width have lot's of overlapping datapoints, which creates a natural high bayes error, and the plot shows that petal length and width are a much better feature to use to identify classes
 - Furthermore, one can see a strong correlation between the later two, so in practice we will only need one of them, we'll use the petal length (3rd feature)

```

# We use the third feature
feature_idx = 2

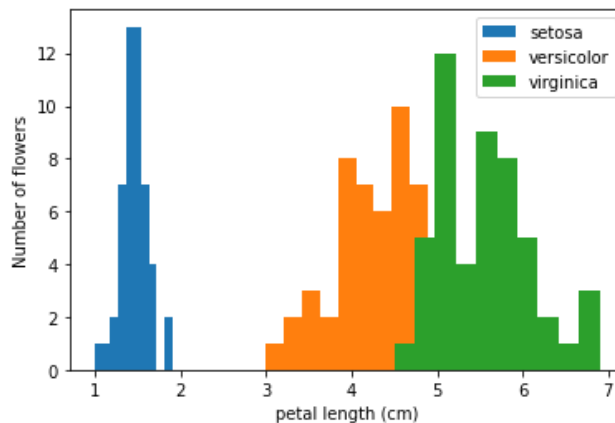
```

- Let's first take a look at the distribution of all flowers (both train and test) along this feature to confirm that our assumption of a normal distribution is correct.

```

plt.hist(setosa_flowers[:,feature_idx], label=iris.target_names[0])
plt.hist(versicolor_flowers[:,feature_idx], label=iris.target_names[1])
plt.hist(virginica_flowers[:,feature_idx], label=iris.target_names[2])
plt.xlabel(iris.feature_names[feature_idx])
plt.ylabel('Number of flowers')
plt.legend()
plt.show()

```



- That looks about correct!
- Now, let's find the parameters of the normal distribution that describe our data best.
- The parameters that we need to describe the distribution are the mean and standard deviation.

```

def compute_mean(x):
    mean = 0
    # START ANSWER
    mean = np.sum(x)/len(x)
    # END ANSWER
    return mean

def compute_sd(x, mean):
    sd = 0
    # START ANSWER
    sd = np.sqrt(np.sum((x-mean)*(x-mean)/len(x)))

```

```

# END ANSWER
return sd

# Compute the mean for each flower type.
mean_setosa = compute_mean(setosa_X_train[:, feature_idx])
mean_versicolor = compute_mean(versicolor_X_train[:, feature_idx])
mean_virginica = compute_mean(virginica_X_train[:, feature_idx])

# Compute the standard deviation for each flower type.
sd_setosa = compute_sd(setosa_X_train[:, feature_idx], mean_setosa)
sd_versicolor = compute_sd(versicolor_X_train[:, feature_idx], mean_versicolor)
sd_virginica = compute_sd(virginica_X_train[:, feature_idx], mean_virginica)

# Print the computed means and standard deviations.
print("setosa", mean_setosa, sd_setosa)
print("versicolor", mean_versicolor, sd_versicolor)
print("virginica", mean_virginica, sd_virginica)

assert np.isclose(mean_setosa, 1.4729729729729728), "Expected a different mean"
assert np.isclose(mean_versicolor, 4.25), "Expected a different mean"
assert np.isclose(mean_virginica, 5.572222222222222), "Expected a different mean"

assert np.isclose(sd_setosa, 0.17652600857089654), "Expected a different standard deviat
assert np.isclose(sd_versicolor, 0.44300112866673375), "Expected a different standard de
assert np.isclose(sd_virginica, 0.547017728288333), "Expected a different standard deviat

```

Probability density function

- A gaussian probability density function p goes like $p(x|\mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$ (it's the one we use for $p(x|y)$)

```

from scipy.stats import norm

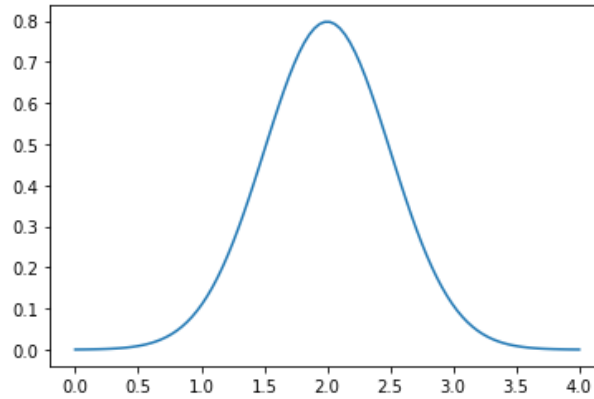
def normal_PDF(x, mean, sd):
    pdf = 0
    # START ANSWER
    pdf = (1/np.sqrt(2*np.pi*np.power(sd,2)))*np.power(np.e, -np.power((x-mean),2)/(2*sd*
    # END ANSWER
    return pdf

# Set x, mean and standard deviation
x = 0.5
mean = 2
sd = 0.5
my_pdf = normal_PDF(x, mean, sd)

# You can compare your outcome to scipy's built-in normal PDF
scipy_pdf = norm.pdf(x, mean, sd)
print("Your pdf function outcome: ", my_pdf, " Scipy's function outcome: ", scipy_pdf)
assert np.isclose(my_pdf, scipy_pdf)

# And we plot the result of your PDF function for 100 points between 0 and 4: np.linspace
xs = np.linspace(0, 4, 100)
plt.plot(xs, normal_PDF(xs, mean, sd))
plt.show()

```

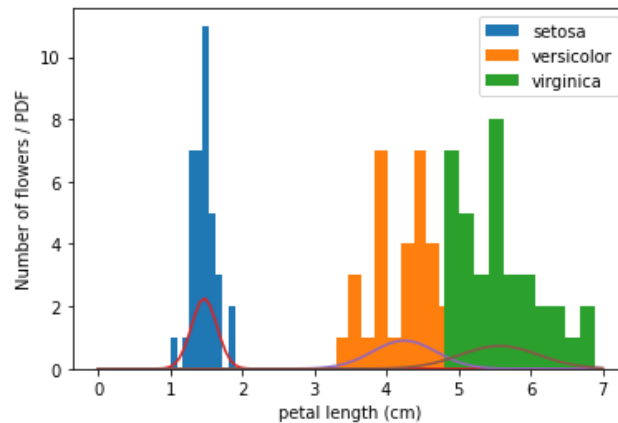


- we can now also define PDFs for each separate class by inserting their respective parameters into the pdf (their respective means and standard deviations)

```
# Histograms of the flower types of the training set
plt.hist(setosa_X_train[:,feature_idx], label=iris.target_names[0])
plt.hist(versicolor_X_train[:,feature_idx], label=iris.target_names[1])
plt.hist(virginica_X_train[:,feature_idx], label=iris.target_names[2])

# Plot your PDFs here
xs = np.linspace(0, 7, 100)
# START ANSWER
plt.plot(xs, normal_PDF(xs, np.mean(setosa_X_train[:,feature_idx]), np.std(setosa_X_train[:,feature_idx])))
plt.plot(xs, normal_PDF(xs, np.mean(versicolor_X_train[:,feature_idx]), np.std(versicolor_X_train[:,feature_idx])))
plt.plot(xs, normal_PDF(xs, np.mean(virginica_X_train[:,feature_idx]), np.std(virginica_X_train[:,feature_idx])))
# END ANSWER

plt.xlabel(iris.feature_names[feature_idx])
plt.ylabel('Number of flowers / PDF')
plt.legend()
plt.show()
```



- The histogram shows the number of flowers that have a petal length within a certain window (bin). That means the values shown in the histogram are absolute counts.

Posterior probabilities

- To get the posterior probability, we can use Bayes' rule (and the sum rule):

$$\circ \quad (C_i|x) = \frac{p(x|C_i)P(C_i)}{p(x)} = \frac{p(x|C_i)P(C_i)}{\sum_{k=1}^K p(x|C_k)P(C_k)}$$

- We will use the good ol' $P(C_i|x)$ posterior probability as a class classifier such that we can also know the probability of the classification being correct (rather than just "most likely")

among all other classes")

- Hint, since we assumed that the gaussian model ($p(x|\mu, \sigma)$) for the class conditional probability, $p(x|C_k) = p(x|\mu, \sigma)$ with μ and σ being the parameters of the objects that belong to class k

```
def posterior(x, means, sds, priors, i):
    """
    Compute the posterior probability P(C_i | x).
    :param x: the sample to compute the posterior probability for.
    :param means: an array of means for each class.
    :param sds: an array of standard deviation values for each class.
    :param priors: an array of frequencies for each class.
    :param i: the index of the class to compute the posterior probability for.
    """
    posterior = 0

    # START ANSWER
    p_x = 0.0
    for k in range(len(priors)):
        p_x += normal_PDF(x, means[k], sds[k])*priors[k]

    posterior = normal_PDF(x, means[i], sds[i])*priors[i]/p_x
    #END ANSWER

    return posterior

means = [mean_setosa, mean_versicolor, mean_virginica]
sds = [sd_setosa, sd_versicolor, sd_virginica]
priors = [
    setosa_X_train.shape[0]/X_train.shape[0],
    versicolor_X_train.shape[0]/X_train.shape[0],
    virginica_X_train.shape[0]/X_train.shape[0]
]

# Test out the code
flower_idx = 6
print("Flower belongs to class", iris.target_names[Y_train[flower_idx]])

# iterate over all classes
for i in range(3):
    x_post = posterior(X_train[flower_idx, feature_idx], means, sds, priors, i)
    print("Posterior probability for class", iris.target_names[i], ": ", x_post)

post_setosa = posterior(X_train[flower_idx, feature_idx], means, sds, priors, 0)
post_versicolor = posterior(X_train[flower_idx, feature_idx], means, sds, priors, 1)
post_virginica = posterior(X_train[flower_idx, feature_idx], means, sds, priors, 2)

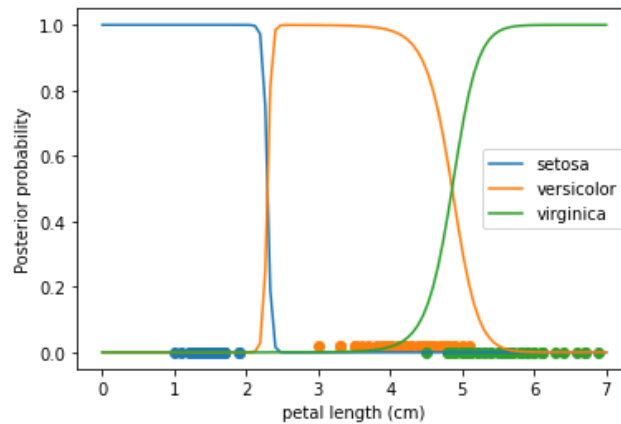
assert np.isclose(post_setosa, 1.1048294835009998e-107, rtol = 0.0001, atol = 0.), "Expect"
assert np.isclose(post_versicolor, 0.03817178391547811, rtol = 0.0001, atol = 0.), "Expect"
assert np.isclose(post_virginica, 0.9618282160845218, rtol = 0.0001, atol = 0.), "Expect"
```

- Let's plot the posterior probabilities and compare them with the data

```
xs = np.linspace(0, 7, 100)
# START ANSWER
plt.plot(xs, posterior(xs, means, sds, priors, 0), label=iris.target_names[0])
plt.plot(xs, posterior(xs, means, sds, priors, 1), label=iris.target_names[1])
plt.plot(xs, posterior(xs, means, sds, priors, 2), label=iris.target_names[2])

plt.scatter(iris.data[np.where(iris.target == 0), 2], iris.target[np.where(iris.target == 0)])
plt.scatter(iris.data[np.where(iris.target == 1), 2], iris.target[np.where(iris.target == 1)])
plt.scatter(iris.data[np.where(iris.target == 2), 2], iris.target[np.where(iris.target == 2)])
```

```
# END ANSWER
plt.xlabel(iris.feature_names[feature_idx])
plt.ylabel('Posterior probability')
plt.legend()
plt.show()
```



- The posterior probabilities matches the data

Bayes Classifier

- We can compute the posteriors for every class such that an input x in `classify(x, params, priors)` returns a class C_i

```
def classify(x, means, sds, priors):
    classification = -1
    # START ANSWER
    post_0 = posterior(x, means, sds, priors, 0)
    post_1 = posterior(x, means, sds, priors, 1)
    post_2 = posterior(x, means, sds, priors, 2)

    if (post_0 > post_1) and (post_0 > post_2): classification = 0
    elif (post_1 > post_0) and (post_1 > post_2): classification = 1
    elif (post_2 > post_0) and (post_2 > post_1): classification = 2
    # END ANSWER
    return classification

# Test out the code
flower_idx = [5, 20, 30]
predicted_classes = np.zeros(3, dtype=np.int64)
for i, flower_idx in enumerate(flower_idx):
    predicted_classes[i] = classify(X_train[flower_idx, feature_idx], means, sds, priors)

print("Predicted class", iris.target_names[predicted_classes])
print("Flower belongs to class", iris.target_names[Y_train[flower_idx]])
assert (predicted_classes == Y_train[flower_idx]).all()
```

```
Predicted class ['virginica' 'versicolor' 'setosa']
Flower belongs to class ['virginica' 'versicolor' 'setosa']
```

- Let's now use the test set to **evaluate** how good the classifier was by returning the percentage of elements in the test set that were classified correctly

```
def evaluate(X_test, Y_test, means, sds, priors):
    accuracy = 0
    # START ANSWER
```



```

total = len(Y_test)
predicted_classes = np.arange(total)
correct = 0

for i in range(total):
    predicted_classes[i] = classify(X_test[i], means, sds, priors)
    if (predicted_classes[i] == Y_test[i]): correct+= 1

accuracy = correct/total
# END ANSWER
return accuracy

accuracy = evaluate(X_test[:, feature_idx], Y_test, means, sds, priors)

print(accuracy)
assert accuracy > 0.9, "Expected a higher accuracy"

```

Decision boundary

- The function to create the decision boundaries does this:
 - For each class, compute the posterior for 1000 points between 1 and 7
 - If for any two classes the posteriors are as good as equal (and not very close to 0) at a point, add that point to the list of decision boundaries
 - Plot vertical lines at these points

```

def decision_boundary(means, sds, priors):
    decision_boundaries = []
    # START ANSWER
    decision_boundaries = np.zeros(1000)
    bcount = 0

    posterior_0 = np.zeros(1000)
    posterior_1 = np.zeros(1000)
    posterior_2 = np.zeros(1000)

    xs = np.linspace(1, 7, 1000)

    for i in range(1000):
        posterior_0[i] = posterior(xs[i], means, sds, priors, 0)
        posterior_1[i] = posterior(xs[i], means, sds, priors, 1)
        posterior_2[i] = posterior(xs[i], means, sds, priors, 2)

        if np.isclose(posterior_0[i],posterior_1[i], atol=0.05) and posterior_0[i] > 0.1:
            decision_boundaries[bcount] = xs[i]
            bcount += 1

        elif np.isclose(posterior_0[i],posterior_2[i], atol=0.05) and posterior_0[i] > 0:
            decision_boundaries[bcount] = xs[i]
            bcount += 1

        elif np.isclose(posterior_1[i],posterior_2[i], atol=0.05) and posterior_1[i] > 0:
            decision_boundaries[bcount] = xs[i]
            bcount += 1

    # END ANSWER
    return decision_boundaries

# Create a scatterplot of the third feature.
feature_idx2 = 3

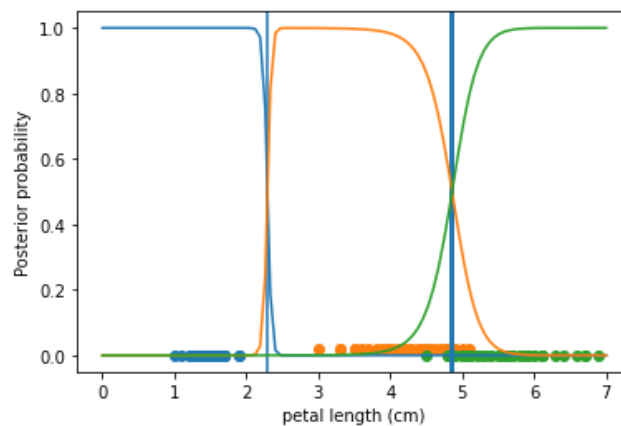
plt.scatter(iris.data[np.where(iris.target == 0), 2], iris.target[np.where(iris.target =
plt.scatter(iris.data[np.where(iris.target == 1), 2], iris.target[np.where(iris.target =
plt.scatter(iris.data[np.where(iris.target == 2), 2], iris.target[np.where(iris.target =

```

```
plt.xlabel(iris.feature_names[feature_idx])
plt.ylabel('Posterior probability')
decision_boundaries = decision_boundary(means, sds, priors)
for boundary in decision_boundaries:
    if boundary != 0:
        plt.axvline(x=boundary)

# Add the posterior probabilities
plt.plot(xs, posterior(xs, means, sds, priors, 0),label=iris.target_names[0])
plt.plot(xs, posterior(xs, means, sds, priors, 1),label=iris.target_names[1])
plt.plot(xs, posterior(xs, means, sds, priors, 2),label=iris.target_names[2])
plt.show()
```

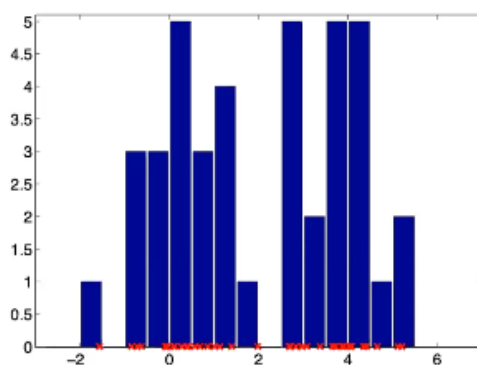
- This method was complicated, it's easier to just solve for the x where the class probabilities are equal



Non-parametric density estimation

- No knowledge of the distribution
- Need to manage the smoothness of the distribution

Histogram



- Split the data points in bins of width h
- Count the number of objects in each bin: k
- Probability density estimate for a sample x :
 - fraction of samples in a bin (containing sample x) normalized by the size of the bin

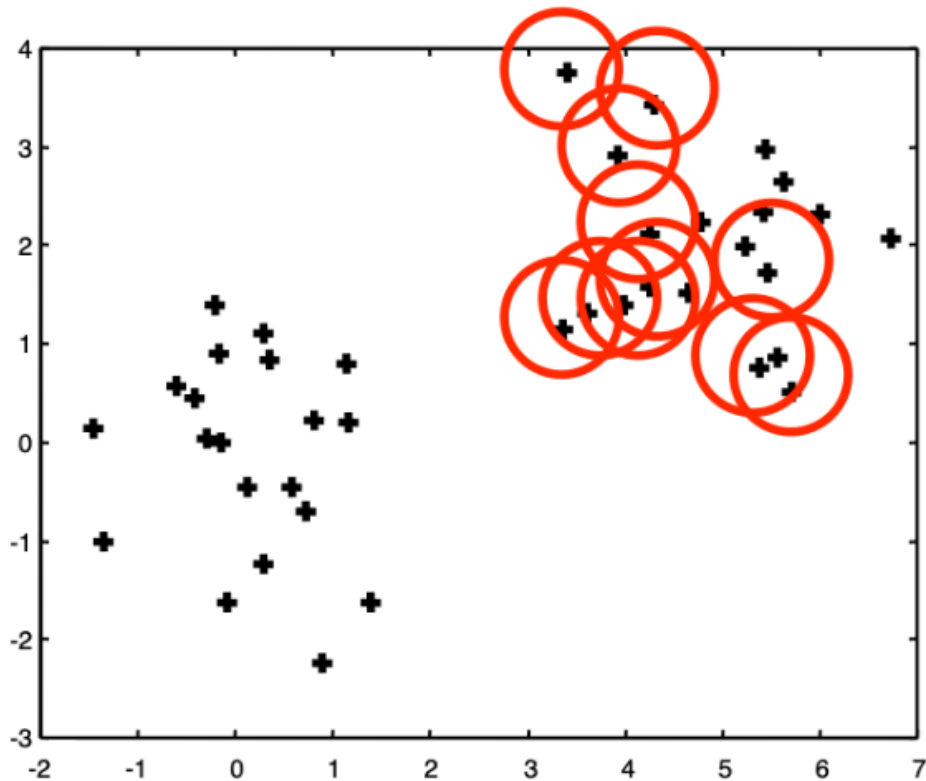
$$\hat{p}(x) = \frac{1}{h} \frac{k_x}{N}$$

- For a reasonably precise approximation the bin size (h) can't be too large
- For a stable estimate the bin size cannot be too small (otherwise it overfits to training data and it's too affected by noise)
- The final value will depend on the number of training examples
- From the histogram there are two deriving techniques:

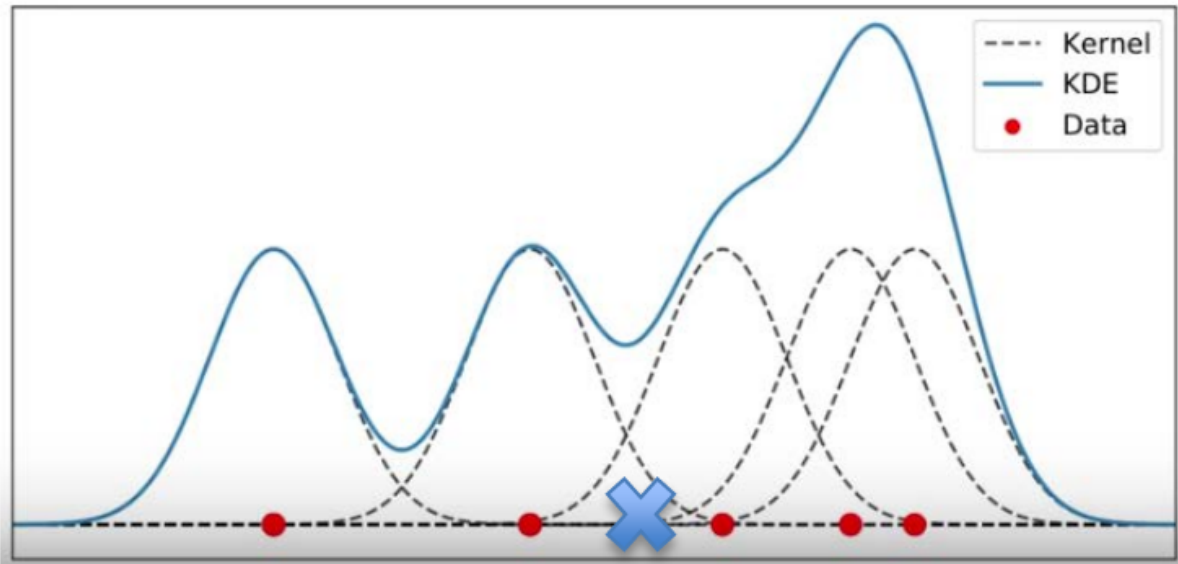
- Parzen (window/kernel) density estimate
- K-nearest-neighbor density estimate

Parzen (window/kernel) density estimate

- Goal: Estimate the class conditional distribution $p(x|y)$ (which is easily derived from the training data where both x and y hold) without assuming a known model (such as gaussian)
- We apply a "kernel" function to each point
 - This function does have a defined shape (gaussian (round), histogram boxes (squared), etc.)
 - Gaussian shape for the kernel function is the most popular choice
 - We also fix the size (h) of the kernel function, such as the width of the bin box or the variance of the gaussian



- Then we merge all the shapes together (and apply some smoothing) to get the final "kernel" probability density function that we'll use for the class conditional probability $p(x|y)$ (although in this isolated kernel context it is just regarded as $\hat{p}(x)$, aka probability of a point x given the kernel distribution)



- The “smoothing” and “merging” (Parzen probability density) function is:

$$\hat{p}(x) = \frac{1}{nh} \sum_{i=1}^n K(x, x_i, h)$$

- K = kernel function (gaussian, box, triangle, etc)
- h is the fixed size of the K kernel function
- x is the object/point/feature vector which we are interested about knowing his class conditional probability (aka parzen probability)
- x_i are all the other points in the training set (with the same class).
- $\frac{1}{nh}$ is just a normalization constant
- In the event of a gaussian kernel function, $x_i = \mu$ and $h = \sigma^2$

$$\hat{p}(x) = \frac{1}{nh} \sum_{i=1}^n \left(\frac{1}{\sqrt{2\pi h^2}} e^{-\frac{(x-\mu)^2}{2h^2}} \right)$$

- This is simply the average of n gaussian functions with each data point as a center.
- For non-gaussian kernels it is implicitly assumed that “x” in $K(x)$ is actually the result of $\frac{x-x_i}{h}$, to which the final outcome might be determined by a series of if else statements:

Given a set of four data points:

$$x_1 = 2, x_2 = 2.5, x_3 = 3.5, x_4 = 0.5$$

find Parzen probability density function (pdf)

estimates at $x=3$ using the kernel function with width $h=1$:

$$K(x) = \begin{cases} 0.5 & \text{if } |x| < 1 \\ 0 & \text{otherwise} \end{cases}$$

Parzen pfd:

$$\hat{p}(x) = \frac{1}{nh} \sum_{i=1}^n K\left(\frac{x - x_i}{h}\right)$$

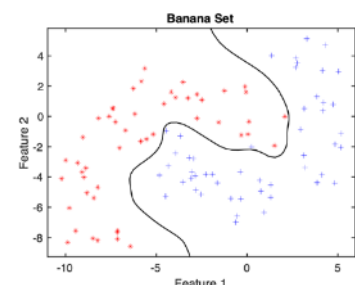
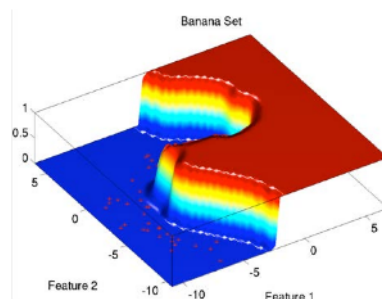
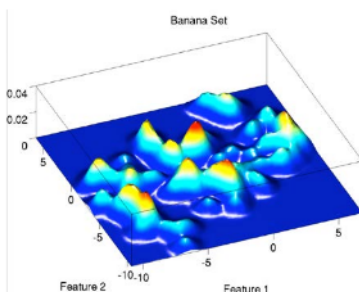
$$\begin{aligned} \hat{p}(x) &= \frac{1}{nh} \sum_{i=1}^n K\left(\frac{x - x_i}{h}\right) = \frac{1}{4} \left(K\left(\frac{3-2}{1}\right) + K\left(\frac{3-2.5}{1}\right) + \right. \\ &\left. K\left(\frac{3-3.5}{1}\right) + K\left(\frac{3-0.5}{1}\right) \right) = \frac{1}{4} (0 + 0.5 + 0.5 + 0) = \frac{1}{4} \end{aligned}$$

Consistent notation for higher dimensions

- In the context of ML of trying to estimate the class conditional probabilities $\hat{p}(x|y)$ (with hat denoting estimation), we'll express a kernel density function that uses Gauss (normal) as kernel model as:

$$\hat{p}(x|y_i) = \frac{1}{n_i} \sum_{j=1}^{n_i} N(x|x_j^{(i)}, hI)$$

- With I as the identity matrix used to generate a covariance matrix for higher dimension models

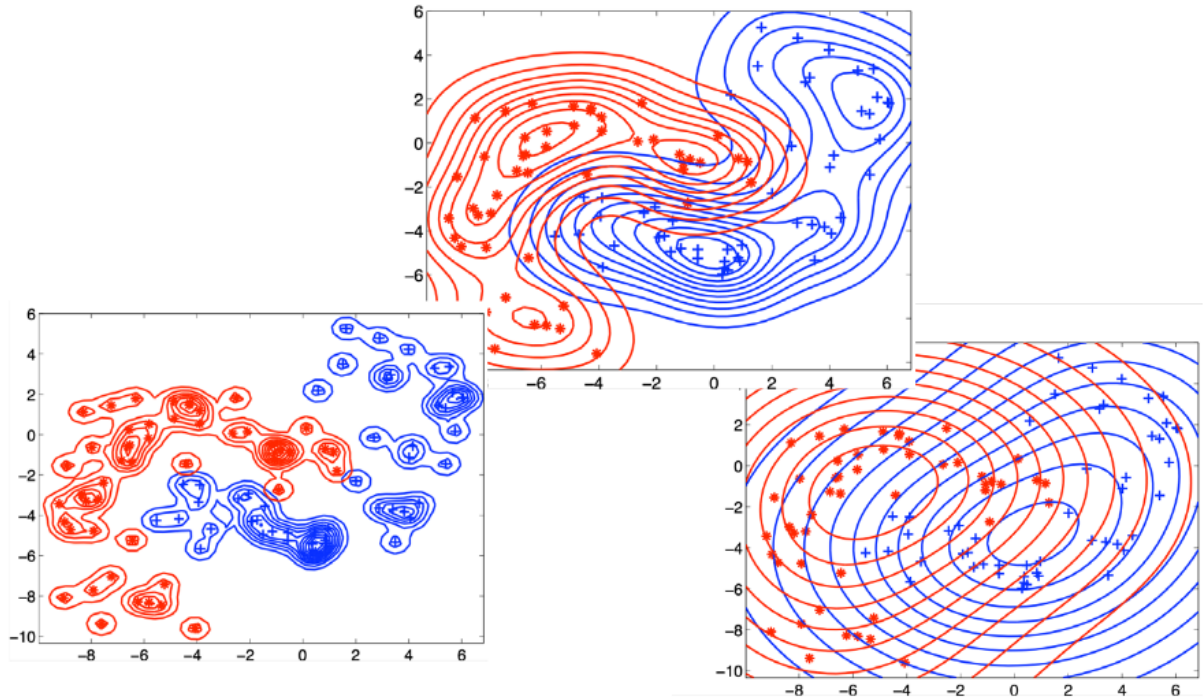


$$\text{Bayes: } \hat{p}(x|y_i)\hat{p}(y_i) > \hat{p}(x|y_j)\hat{p}(y_j)$$

25

Parzen width (h size) parameter optimization

- If width is too small, you overfit to the training points
- If width is too big, you don't distinguish well between classes anymore
- The sweet spot lies in between

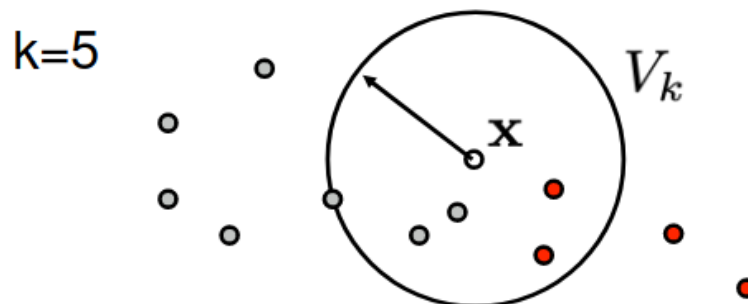


- There are multiple approaches to optimize it:
 - Trial and error for each h size and choose the one that performs the best
 - Optimize the "likelihood" (maximize "log-likelihood") = $\sum \log(\hat{p}(x))$
 - Use the average 10-nearest neighbor distance
 - Use a heuristic

K-nearest neighbors

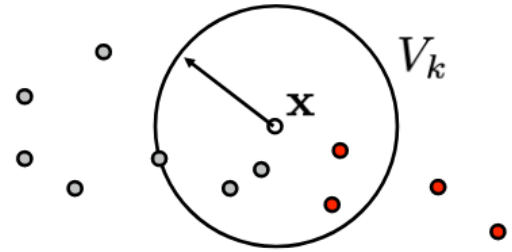
Use the intuition to classify a new point x :

- Locate the cell on the new point x
- Do **not** fix the volume of the cell:
grow the cell until it covers k objects:
find the k -th neighbours
- predict the class y of new point x



- k should be odd to avoid draws

K-nn density estimation



- *Data points:* x_1, x_2, \dots, x_n
- *Classes:* i and j
- $k = 5$

$$\hat{p}(x|y_i) = \frac{k_i}{n_i V_k}$$

- Where V_k is the volume of the sphere centered at x with radius r , being the distance to the k -th nearest neighbor; k_i is the number of neighbours of class i within V_k ; n_i is the number of data points belonging to class i .
- Class priors: $\hat{p}(y_i) = \frac{n_i}{n}$
- Bayes: $\hat{p}(x|y_i)\hat{p}(y_i) > \hat{p}(x|y_j)\hat{p}(y_j) \rightarrow k_i > k_j$
- In practice we won't be using K-nn density estimation for the conditional probability nor the posterior one, but just for the classifier, and hence we'll just be interested in $k_i > k_j$

Optimizing k

- The largest k value is the number of observations in the entire data set
 - You'd end up classifying everything to the class that already had more observations
- The smallest k value is 1
 - You'd end up overfitting to the training data
- You can do trial and error for different values of k and test which value gives the best performance with the test set

Breaking ties

- Use odd k, but with more than 2 classes it won't necessarily solve it
- Assign randomly to a tied class
- Assign to the class that has the largest prior probability
- Use first nearest neighbor to decide

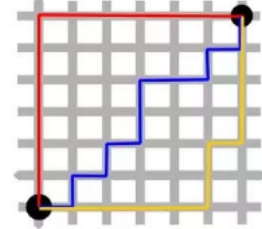
Distance measures

- Euclidean (numeric features):

$$D(x, x') = \sqrt{\sum_d |x_d - x'_d|^2}$$

- Manhattan distance

$$D(x, x') = \sum_d |x_d - x'_d|$$



- Hamming (categorical features):
 - number of features where x and x' differ

$$D(x, x') = \sum_d 1_{x_d \neq x'_d}$$

K-nn pros and cons

- Simple and flexible classifier
- Often a very good classification performance
- It is simple to adapt the complexity of the classifier
- relatively large training sets are needed
- the complete training set has to be stored
- distances to all training objects have to be computed
- the features have to be scaled sensibly
- the value for k has to be optimized

Naive bayes

- Works well for small datasets
- We make strong assumption that all features are independent "given y " such that $p(x|y)$ is the joint product of all $p(x_i|y)$
- With this assumption we get rid of the curse of dimensionality
- In our context, the class value explains all the dependence between features
- To find the individual $p(x_i|y)$ we can either use parametric or non-parametric approaches

Classifier evaluation

- There are many machine learning classifier algorithms:
 - generative classifiers (based on a model such as gaussian, and decision boundary based on bayes rule):
 - parametric densities (features are spread like the model):
 - Two-Class case:
 - Quadratic Density classifier: Uses the log of the conditionals as classifier, which is a quadratic function (line or hyperbola).
 - Linear Discriminant Analysis (LDA): for 2D and more dimensions, the covariance matrix of each class gaussian distribution might not be invertible. Therefore covariance matrix of all classes are the same (just identity times average variance of all features), which inherently makes the decision boundary a straight line. Although covariance is the same each class gaussian can have different mean.

- Nearest mean classifier: When even the average variance of the entire feature space is hard to compute (i.e. curse of dimensionality). We assume all features have the same variance and all features are uncorrelated. It's also a linear decision boundary but just uses the shortest distance to the mean of a class as classification criteria.
- nonparametric densities:
 - Parzen (it uses a model but it is only applied in the "kernel" function, which is applied at individual feature vectors, and the only hyperparameter is the size of the window (i.e. bin width, gaussian width (variance)) and then average the distributions of all individual points)
 - i.e. simply the average of n gaussian functions with each data point as a center (and all gaussians having the same variance h).
 - k-nearest neighbour: classifier purely based on distance. (could be euclidian aka normal, "manhattan", amount of matches, etc.). Class is assigned to the majority class among the nearest k neighbours.]
- discriminative densities:
 - logistic classifier
 - support vector
 - decision trees
 - perceptio
 - neural networks
- To select which one to use we could either test each of them on cross-validated training/test sets and select the one that performs best on average
- Alternatively we could take the nature of the characteristics of data sets to chose a model
 - When the data is scarce, gaussian based classifiers perform poor
- For high dimensional data sets (more than 2D) it is common to bruteforce and try all classifiers and rank them by classification performance/error.
 - Don't overfit to the training despite the training set being a representative sample of the true distribution
 - Split data into train and test set
 - Apply bootstrapping, cross-validation or leave-one-out
 - Analyze the learning curve
 - Consider classifier complexity
 - Analyze the bias-variance tradeoff
 - Consider confusion matrices
 - Analyze ROC curve
 - Consider reject curve

Error estimation by Test Set

- Typically all the data that you start with is called the design set
- How to test the classifier? -> Use test data separated (sacrificed) from the training data
- If you change the training set, you get a different classifier with a different error estimate

Evaluating correctness

- Just apply the test object (of which we know the class) to the classifier function and assess whether the output matches the known class.
- Each of these evaluations returns a boolean: correct or incorrect, 1 or 0. Therefore the "correctness" or it's antagonist "error" are the sum of "Bernoulli" random variables wherer error is the average of Z_i :

- $$\hat{\epsilon} = \frac{1}{N} \sum_{i=1}^N Z_i$$

- $$Z_i = \begin{cases} 0 & \text{if } x_i \text{ is correctly classified} \\ 1 & \text{if } x_i \text{ is incorrectly classified} \end{cases}$$

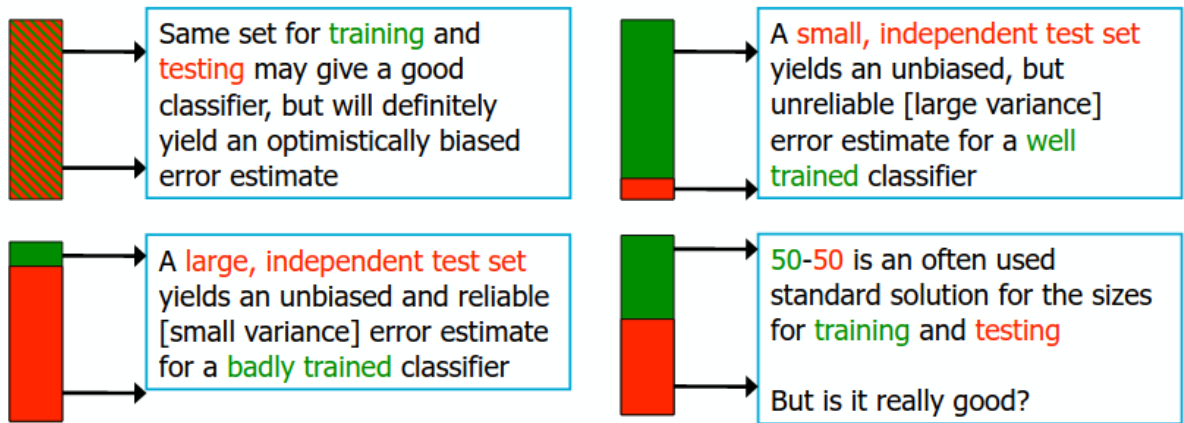
- Bernoulli error random distribution has the variance:

$$\sigma^2 = \text{Var}(\hat{\epsilon} \mid \text{test size } N) = \frac{\epsilon(1 - \epsilon)}{N}$$

- We can see that as the test size (N) increases, the variance of the error will decrease.
 - Which means that the estimate of the error that you got might be way off the real error.
 - Therefore even though we might want to use the smallest average error classifier, if that error has high variance we might have to discard it to be sure
- We can also see that the variance is a quadratic formula with error 0% and 100% giving the minimum (zero) variance and error 50% giving the highest variance

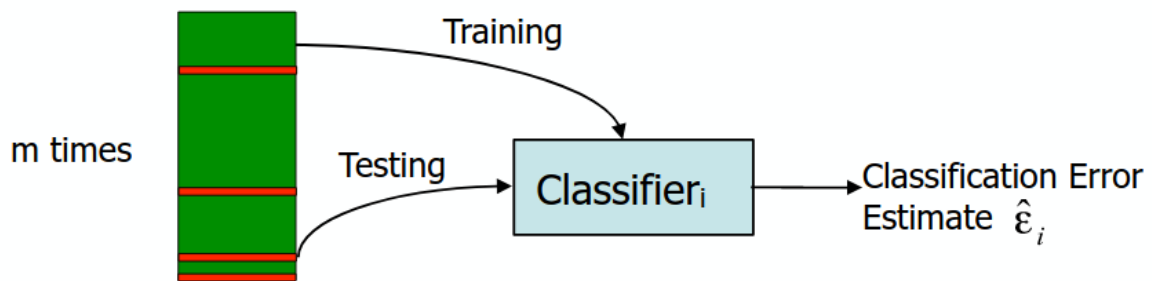
Training set size vs test set size

- There is a trade off between being having a large enough training set to fit the training parameters of the model and having a large enough test set to have a small classifier error variance.
 - Large training set = good classifier
 - Large test set = reliable, unbiased error estimate



Bootstrapping

- Randomly draw N objects from N objects (**with replacement**)



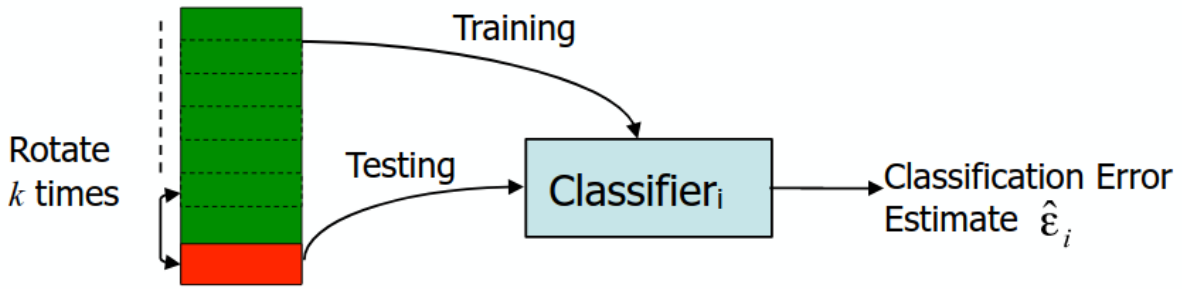
$$\hat{\epsilon} = \frac{1}{m} \sum_i \hat{\epsilon}_i$$

- Left-over objects are used for testing
- Repeat m times
- in the context of a 40 deck card set, taking a card, writing its value on a piece of paper, then putting the card back to the deck (replacement) and repeat the same process 40 times, will

probable make some cards to be withdrawn more than once and some cards to not have ben withdrawn at all (left-over).

- In the context of the design set, the left-over objects are sent to the test set, and the withdrawn objects are sent to the training set.
 - Some classifiers perform bad when there are duplicates in the training set (i.e. non-invertible covariance matrix?)
- Then the classifier is trained an evaluated under those sets and it's error is annotated. However this error belongs to a single random training/test setting.
- You repeat the bootstrapping process m times and take the average error as a measure of the classifier performance.

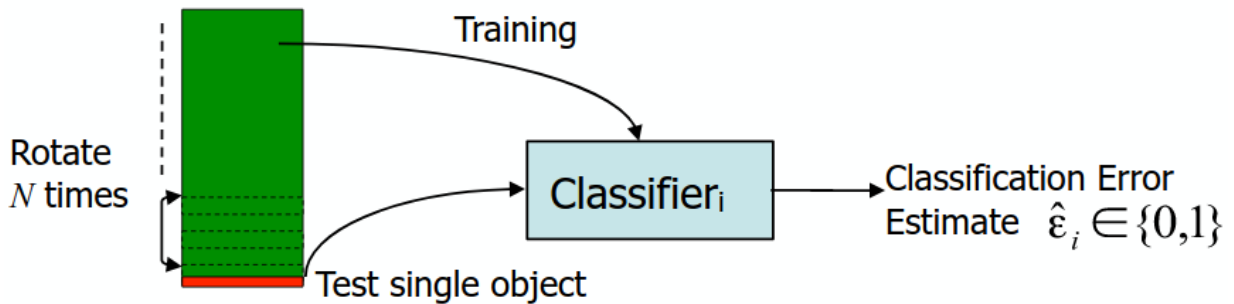
K-fold cross validation



$$\hat{\epsilon} = \frac{1}{k} \sum_i \hat{\epsilon}_i$$

- Splits the design set in k separate parts
- Then use all fractions but 1 for training and the remaining for testing
- You can then train and test the classifier 10 times for each possible test set
- Take the average the error estimate of those k tests
- If you like the average error, then train once more but with the whole design set as the training set and keep that as the final classifier (but it cannot be tested anymore as there's no partition left for testing)

Leave-one-out procedure



$$\hat{\epsilon} = \frac{1}{N} \sum_i \hat{\epsilon}_i$$

- This is good because eventhough the single object test is very small, we average it over all possible permutations.

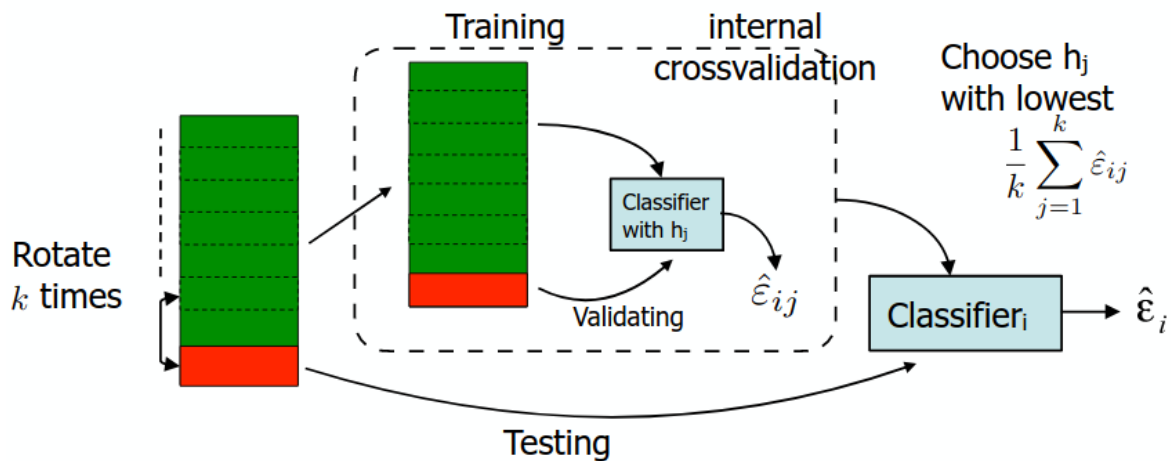
- However, this is computationally very expensive
- We generally chose the biggest number of folds we can afford (with design set len being the max)
 - This lies between 5 and 10 folds

Optimising hyperparameters

- Hyperparameters are those values used in kernel functions such as bin size, number of neighbours k , width parameter h (gaussian variance), etc.
- **DON'T** optimise them by choosing the number that performs best on the tests.
 - If you do it you are likely overfitting to the test set and you won't have any left-over samples to test the choice of hyperparameters on an independent set
- **To optimise hyperparameters apply double cross-validation**

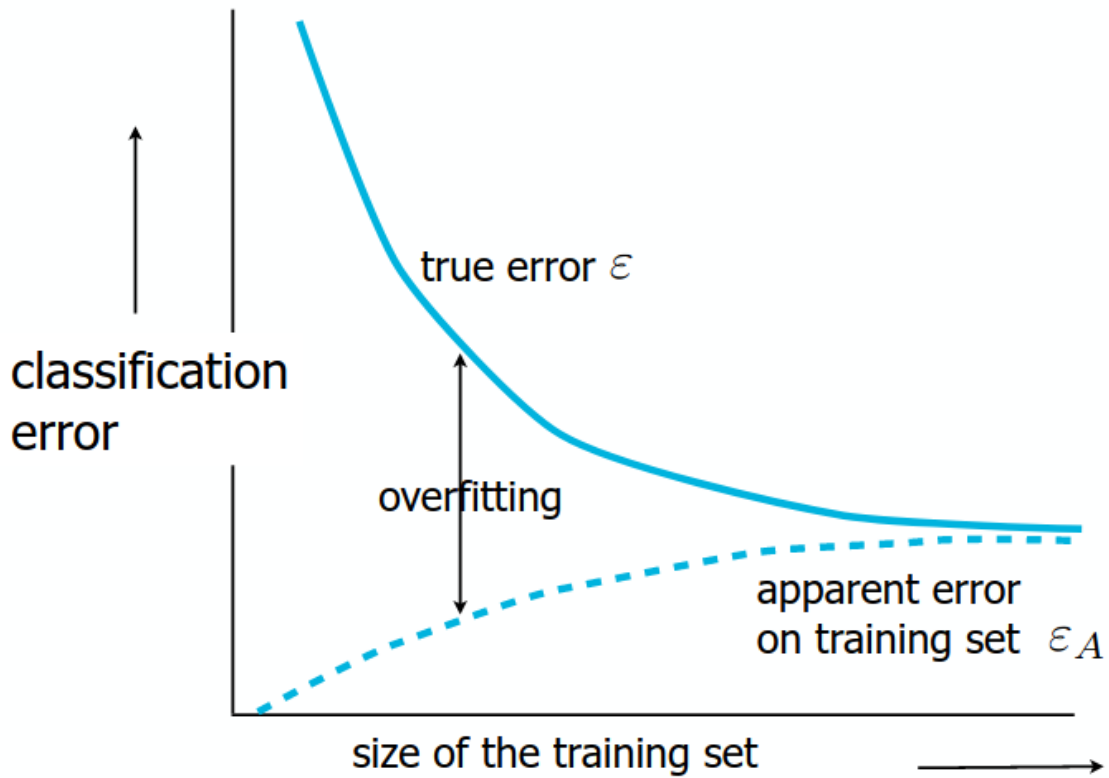
Double cross-validation

- To optimise hyperparameters, do cross-validation **inside** another cross-validation:



- The test set of the hyperparameter (nested) cross-validation is called the validation set
- You start with the regular training/test k fold crossvalidation to test different classifiers.
 - within each fold you test each value of the hyperparameter, then write the average error for that fold
- Repeat for all folds and chose the classifier whose average hyperparameter error is the smallest

Learning curve



- For all models, the **apparent** classification error of the classifier **on the training set itself** increases as the size of the training set increases, because as more data points are added, there's more room for overlapping features with different classes.
- However, when testing the model on an **independent test set**, (which aims to measure the **true error**), a small training sample is very likely to be irrepresentative of the true population, so as the training set size increases, it's distribution resembles the independent test set, and thus it can be fit closer to the test set.
- Overfitting is the error distance between true error (from test set) and apparenet error (from training set)
 - Aim for a small gap or perhaps chose a less flexible/complex classifier (which perform better on smaller sets)

Different classifier complexity



- Complex classifiers perform better with larger training sets
 - Same holds for larger feature sets
- Simple classifiers perform better with smaller training set

- Same holds for feature sets
- Fundamentally best classifier depends on the size of the training set
- As the training size increases it reaches an horizontal asymptote
- Larger training sets yield better classifiers
- Independent test sets needed for unbiased error estimates
- Larger test sets yield more accurate error estimates
- LOO cross validation "optimal", but might be infeasible
- 10-fold crossvalidation is often used

Bias-variance tradeoff

Squared Error

$$L(w) = E[|g(x) - y|^2]$$

- We have to deal with the fact that we can encounter an arbitrarily complex distribution
- The squared error is used to measure the difference between the predicted output and the real value
- E means expected value

Bias-variance dilemma

- The classifier function could be expressed as $f(x)$ with x as the feature vector/object we want to predict its class. But technically $f(x)$ depends on the training set D it was trained, so we express the classifier function as $f(x;D)$
- We can re-write the square error as:

$$\circ \quad E_D[(g(x; D) - y)^2]$$

- This can be rewritten as:

$$\circ \quad E_D[(g(x; D) - E_D[g(x; D)])^2] + E_D[(E_D[g(x; D)] - y)^2]$$

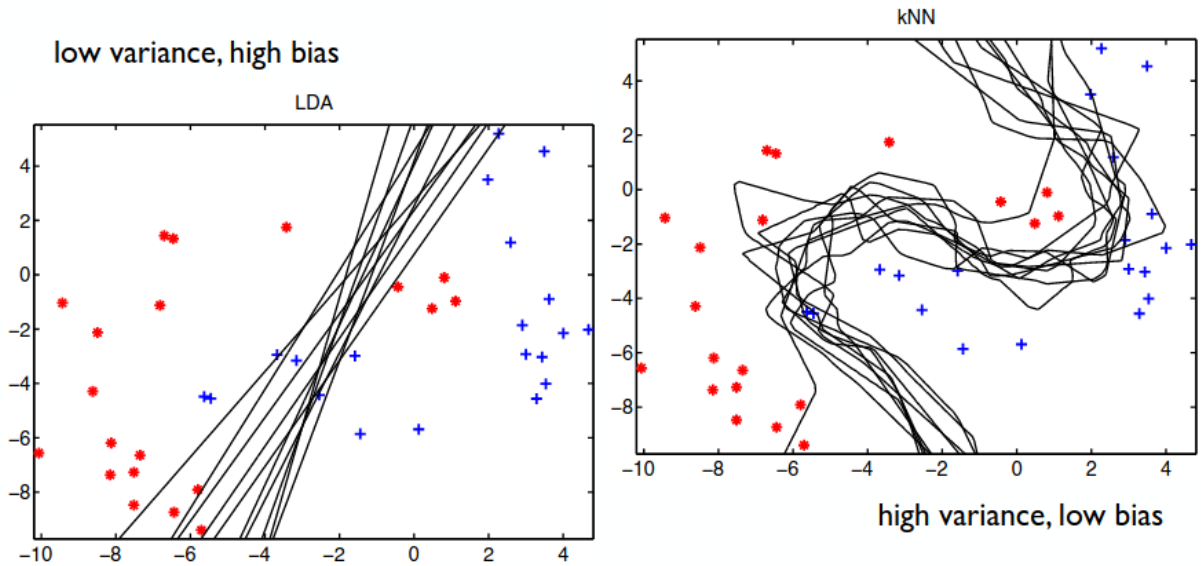
$$\circ \quad = \textit{variance} + \textit{bias}$$

- The take away is that the error of any given classifier can be decomposed in two terms that depend on the design set
 - variance indicates how stable the error value is if we change the design set (if we change the train/test partitioning)
 - bias = how good on average my model is compared to the true labels

Bias-variance trade-off

- Usually the more complex the model, the lower the bias but the highest the variance

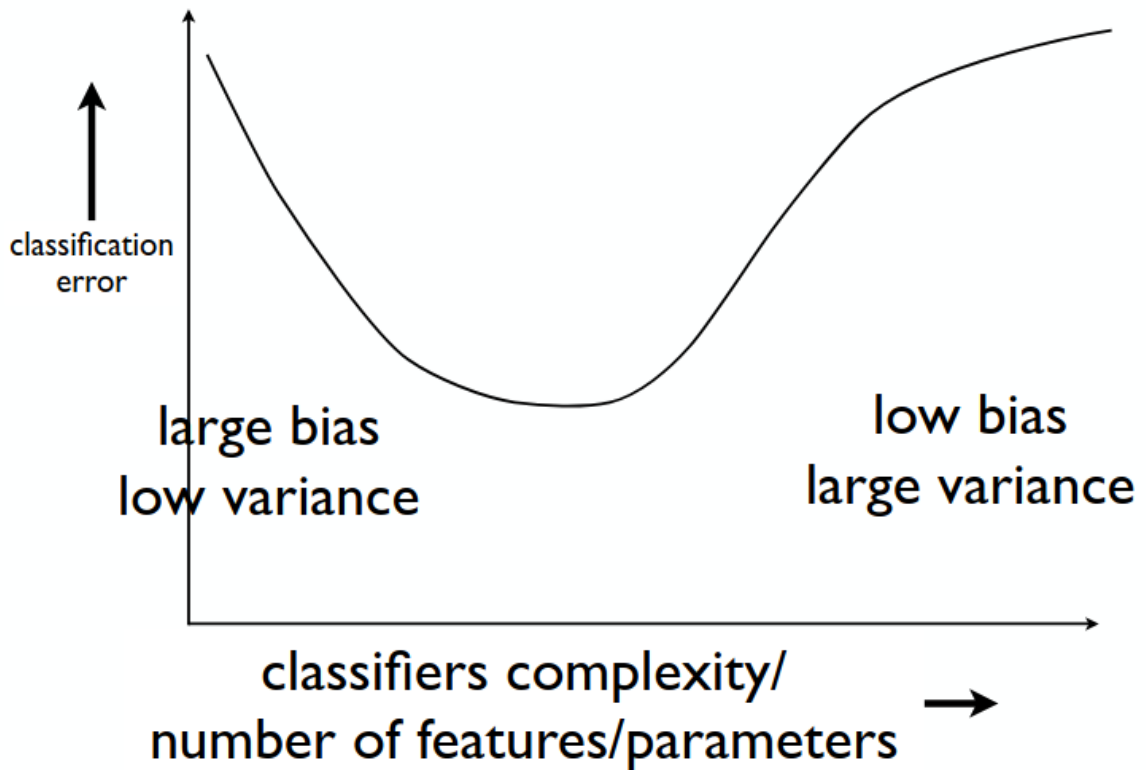
• Compare LDA with kNN:



- Variance, bias, in relation to changing the arrangement of the design test (into train/test sets)
- More simple classifier is more stable (and need less data to train)
- More complex classifier only works when you have sufficient number of training data

Feature curve

- While keeping the training set size constant:



Error/Performance measures

- Error: probability of erroneous classifications
- Performance / accuracy: 1 - error

- Sensitivity of a target class [e.g. diseased patients]: performance for objects from that target class
- Specificity: performance for all objects outside target class
- Precision of a target class: fraction of correct objects among all objects assigned to that class.
- Recall: fraction of correctly classified objects; identical to sensitivity when related to particular class
- True positive rate: identical to sensitivity
- False positive rate: error for all objects outside targ

Two-class classification errors

- There are tons of different performance measures
- Standard classification error: the weighted average (in terms of class frequency) of the classifier error for each class $\epsilon = \epsilon_1 p(y_1) + \epsilon_2 p(y_2)$
- Weighted classification error: we apply an additional weight to each type of missclassification error λ_{12} means classifying 1 as 2: $\epsilon = \lambda_{12} \epsilon_1 p(y_1) + \lambda_{21} \epsilon_2 p(y_2)$
- ...

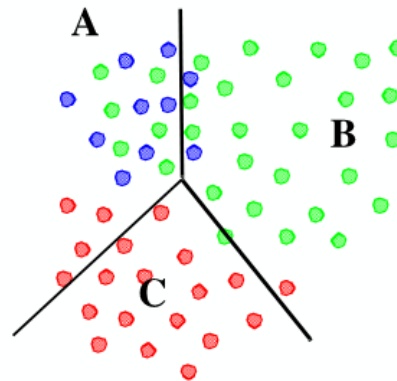
Confusion Matrices

- Gives you more insight on where things go wrong
 - Which classes and objects are troublesome by producing counts of class-dependent errors (How many objects have been classified as A that should have been B?)

$$N_A = 10, N_B = 30, N_C = 20$$

$$E = \frac{c_{12} + c_{13} + c_{21} + c_{23} + c_{31} + c_{32}}{N_A + N_B + N_C}$$

$$E = 14 / 60 = 0.2333$$



		classified to			
		A	B	C	
objects from	class A	8	2	0	0.20 error in class A
	class B	6	23	1	0.23 error in class B
	class C	4	1	15	0.25 error in class C
					0.228 averaged error

$$C = \text{confmat}(\Lambda, L)$$

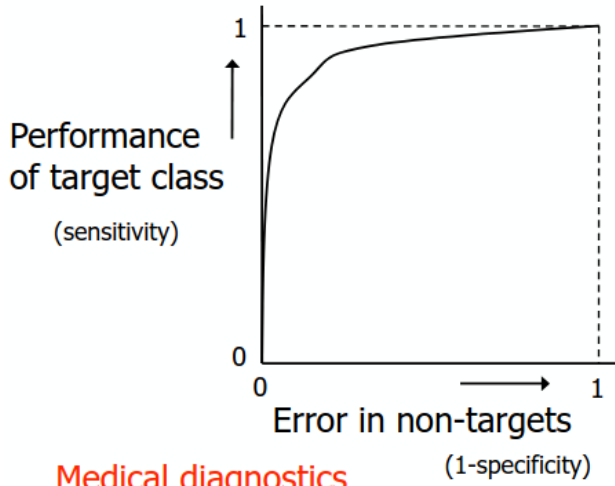
Λ real labels

L obtained labels

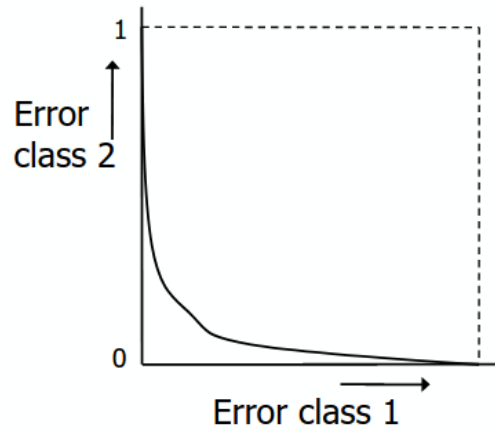
Reciever-Operator Analysis

- Depending on where you draw the threshold, increasing the number of errors on one class will naturally decrease the number of missclassified errors on the other class (in a two-class case)

ROC: Receiver-Operator Characteristic (from communication theory)



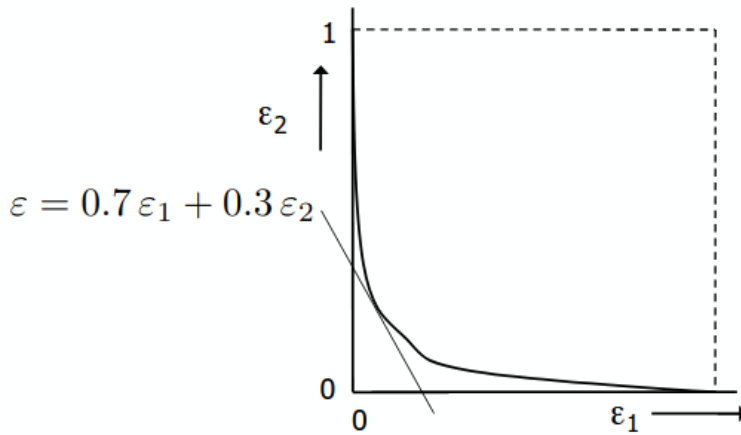
Medical diagnostics
Database retrieval



2-class pattern recognition

- The optimal location for the trade-off of a two class classifier with different errors for each class is found when the line touches the curve

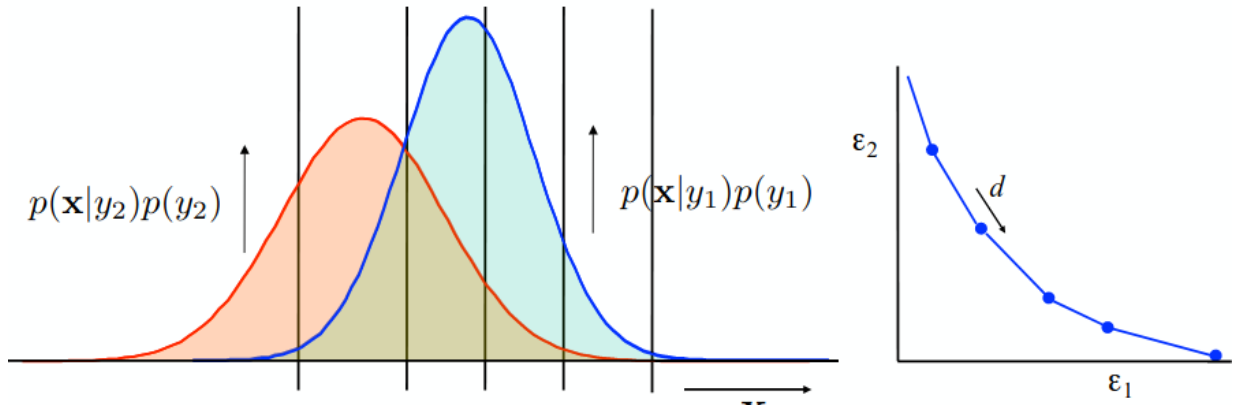
Effect of Changing Priors/Costs



$$\epsilon = p(y_1) \epsilon_1 + p(y_2) \epsilon_2$$

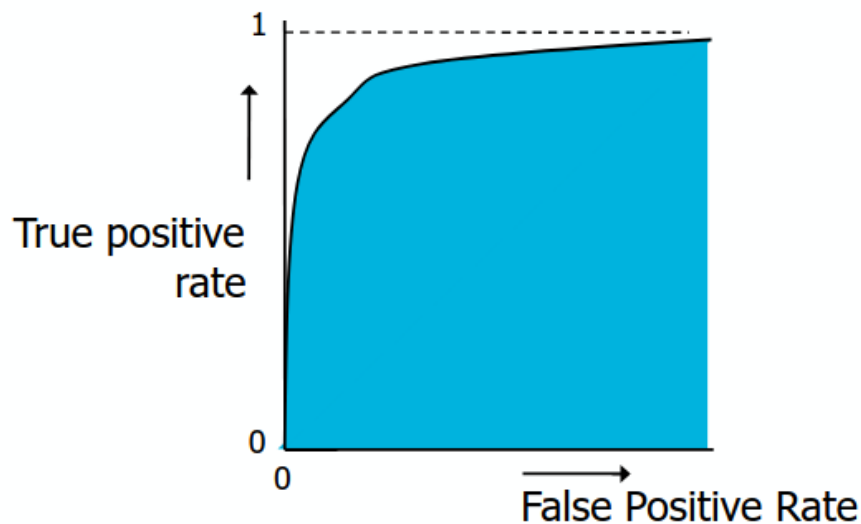
ROC Curve

- Curve is obtained by varying classifier threshold d



Area under the ROC curve (AUC)

- Integrate the area under the roc curve (in terms of true vs false rate of the same class) and the higher the better, ideally it should be 1
- A random classifier (that is correct half the time for a two class problem) would give 0.5
- This performance measures is insensitive to class priors (frequency of each class doesnt affect)



Lab: K-Nearest Neighbors in python

Load Data

- Load the iris dataset

```
import numpy as np
from sklearn import datasets # To Load the dataset
from sklearn.model_selection import train_test_split # To split in train and test set

seed = 20
# Load the data and create the training and test sets
iris = datasets.load_iris()
```

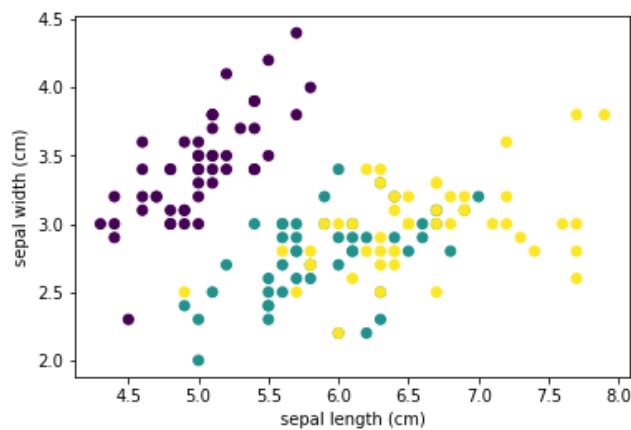
```
# X is the feature vectors of the data points, and Y is the target (ground truth) class
X_train, X_test, Y_train, Y_test = train_test_split(iris.data, iris.target, test_size=0.2)
```

- Plot the data (of the 1st and 2nd features) to verify that they can be separated by non-linear/parametric boundaries

```
from matplotlib import pyplot as plt

# START ANSWER
from matplotlib import pyplot as plt

# START ANSWER
plt.scatter(iris.data[:, 0], iris.data[:, 1], c=iris.target)
plt.xlabel(iris.feature_names[0])
plt.ylabel(iris.feature_names[1])
plt.show()
# END ANSWER
```



Distance

- Next, we will create a function to compute distance between two points \mathbf{p} and \mathbf{q} . We will employ the often used Euclidean distance to find the nearest neighbours of a point

Note: As we are working with feature vectors, the "." depicts a dotproduct:

$$d(\mathbf{p}, \mathbf{q}) = \sqrt{(\mathbf{p} - \mathbf{q}) \cdot (\mathbf{p} - \mathbf{q})}$$

Hint: You might know a more specific formulation of this as $|\mathbf{p}| = \sqrt{p_1^2 + p_2^2}$

```
from scipy.spatial import distance

def euclidean(p, q):
    """
    Computes the Euclidean distance between point p and q.
    :param p: point p as a numpy array.
    :param q: point q as a numpy array.
    :return: distance as float.
    """

    dist = 0
    # START ANSWER
    dist = np.sqrt((p-q) @ (p-q))
    # END ANSWER
    return dist
```

```
# Check whether your algorithm is correct
a = np.array([2, 4, 8])
b = np.array([3, 5, 9])

print('The output of your algorithm:', euclidean(a, b))
assert np.isclose(euclidean(a, b), distance.euclidean(a, b))
```

Nearest neighbours

```
def get_neighbours(training_set, test_instance, k):
    """
    Calculate distances from test_instance to all training points.
    :param training_set: [n x d] numpy array of training samples (n: number of samples,
    :param test_instance: [d x 1] numpy array of test instance features.
    :param k: number of neighbours to return.
    :return: list of length k with neighbour indices, with increasing distance of the ne
    """

    neighbours = []
    # START ANSWER
    neighbours = np.zeros(len(training_set))

    i = 0
    for q in training_set:
        neighbours[i] = euclidean(test_instance, q)
        i += 1

    neighbours = np.argsort(neighbours)[0:k].tolist() # argsort returns the indices of t
    # END ANSWER
    return neighbours

neighbours = get_neighbours(X_train, X_test[0], 5)

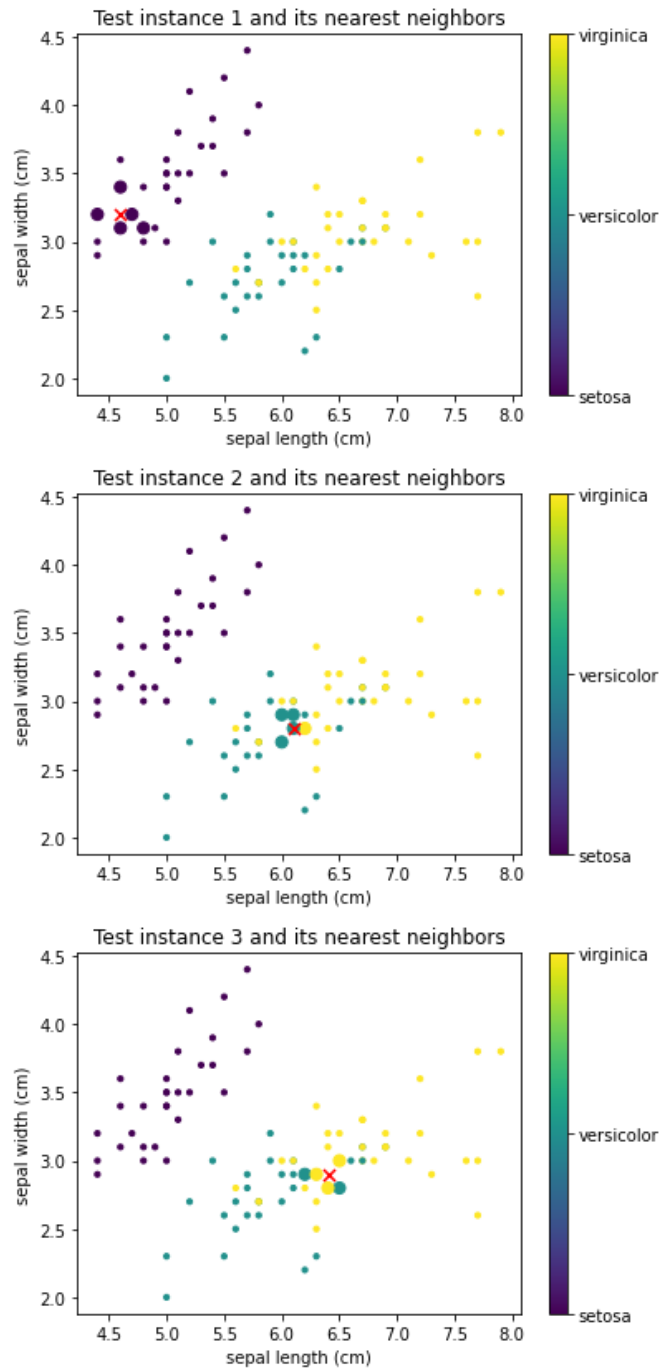
# Check whether your algorithm is correct
print('The indices returned by your algorithm are:', neighbours)
assert neighbours == [63, 41, 76, 51, 10]
```

- Use the provided plot code to show the nearest neighbours for a couple of different values for `k` and a number of test samples.

```
def plot_neighbours(X_train, Y_train, test_instance, k):
    """
    Plots all points in the dataset and shows the neighbours of a given test instance.
    """

    neighbours = get_neighbours(X_train, test_instance, k)
    # Initialization of the sizes of the points to be plotted, size 10
    neigh_sizes = np.ones((len(Y_train), 1)) * 10
    neigh_sizes[neighbours] = 50
    plt.scatter(X_train[:, 0], X_train[:, 1], c=Y_train, s=neigh_sizes)
    plt.xlabel(iris.feature_names[0])
    plt.ylabel(iris.feature_names[1])
    plt.colorbar(ticks = [0, 1, 2], format = plt.FuncFormatter(lambda i, *args: iris.target_names[i]))
    plt.scatter(test_instance[0], test_instance[1], c='r', s=50, marker='x')
    plt.show()

for i in range(3):
    test_instance = X_test[i, [0, 1]]
    k = 5
    plt.title('Test instance %s and its nearest neighbors' % (i+1))
    plot_neighbours(X_train[:, [0,1]], Y_train, test_instance, k)
```



Majority vote

```

from collections import Counter # To count unique occurrences of items in array, for maj

def get_majority_vote(neighbour_indices, training_labels):
    """
    Given an array of nearest neighbours indices for a given test case,
    tally up their classes to vote on the correct class for the test instance.
    :param neighbours: list of nearest neighbour indices.
    :param training_labels: the list of labels for each training instance.
    :return: the label of most common class.
    """

    most_common = -1
    # START ANSWER
    labels = np.array(training_labels)
    most_common = Counter(labels[neighbour_indices]).most_common(1)[0][0]

    # END ANSWER
    return most_common
  
```

```

predicted_label = get_majority_vote(neighbours, Y_train)
print('Your predicted label:', predicted_label)

assert predicted_label == 0
assert get_majority_vote([0,1,2,3,4], [0,2,2,1,3]) == 2
assert get_majority_vote([0,1,2,3,4], [3,1,1,3,0]) == 3

```

Accuracy

- Compute the accuracy of the k-nn model with the the training set itself (which uses the training set to calculate the neighbours)
 - Below we use scikit-learn

```

from sklearn.metrics import accuracy_score

def predict(X_train, X_test, Y_test, k=5):
    """
    Predicts all labels for the test set, using k-nn on the training set.
    :param X_train: the training set features.
    :param X_test: the test set features.
    :param Y_test: the training set labels.
    :return: list of predictions.
    """

    # Generate predictions
    predictions = []
    # For each instance in the test set, get nearest neighbours and majority vote on pred
    # START ANSWER
    predictions = np.zeros(len(X_test))

    for i in range(len(X_test)):
        predictions[i] = get_majority_vote(get_neighbours(X_train, X_test[i], k), Y_test)

    predictions = predictions.tolist()
    # END ANSWER
    return predictions

k = 5
predictions = predict(X_train, X_test, Y_train, k)

# Summarise performance of the classification using scikit-learn
accuracy = accuracy_score(Y_test, predictions)
print('The overall accuracy of the model using scikit-learn is:', accuracy)

assert predictions == [0, 1, 1, 2, 1, 1, 2, 0, 2, 0, 2, 1, 2, 0, 0, 2, 0, 1, 2, 1, 1, 2,
assert np.isclose(accuracy, 0.9666666666666667)

```

- Manually check accuracy

```

def accuracy_score_self(Y_test, predictions):
    """
    Computes the accuracy of a test set as the fraction of items that was classified cor
    :param y_test: the list of true labels for the test set.
    :param y_pred: the list of predicted labels for the test set.
    :return: accuracy as a floating point.
    """

    accuracy = 0
    # START ANSWER

    correct = 0

```

```

total = len(Y_test)

for i in range(total):
    if (Y_test[i] == predictions[i]):
        correct += 1

accuracy = correct/total

# END ANSWER
return accuracy

# Summarise performance of the classification
accuracy_self = accuracy_score_self(Y_test, predictions)
print('The overall accuracy of the model using your implementation of accuracy:', accuracy)
assert np.isclose(accuracy, accuracy_self)

```

- Complete the plot_errors function to get a better understanding of why some points are misclassified.

```

def plot_errors(X_train, X_test, Y_train, Y_test, predictions, k):
    """
    Plots the test points that were misclassified and their nearest neighbours using plot
    """

    # START ANSWER
    predictions = predict(X_train, X_test, Y_train, k)
    accuracy_self = accuracy_score_self(Y_test, predictions)

    total_errors = int(np.round((1-accuracy_self)*len(Y_test)))
    errors = np.zeros((total_errors, len(X_test[0])))

    count = 0

    for i in range(len(Y_test)):
        if (Y_test[i] != predictions[i]):
            errors[count] = X_test[i]
            count += 1

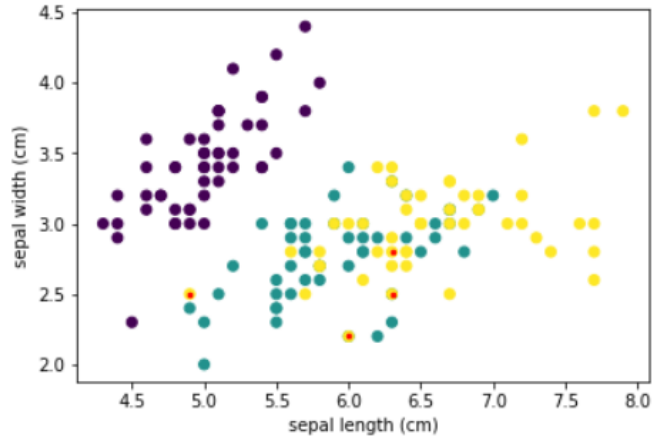
    print("k =", k)
    print(errors)
    # Complete set
    plt.scatter(iris.data[:, 0], iris.data[:, 1], c=iris.target)
    # Mark errors
    plt.scatter(errors[:, 0], errors[:, 1], c='r', s=25, marker='.')
    plt.xlabel(iris.feature_names[0])
    plt.ylabel(iris.feature_names[1])
    plt.show()
    # END ANSWER
    return

for i in range(1,10,2):
    plot_errors(X_train, X_test, Y_train, Y_test, predictions, i)

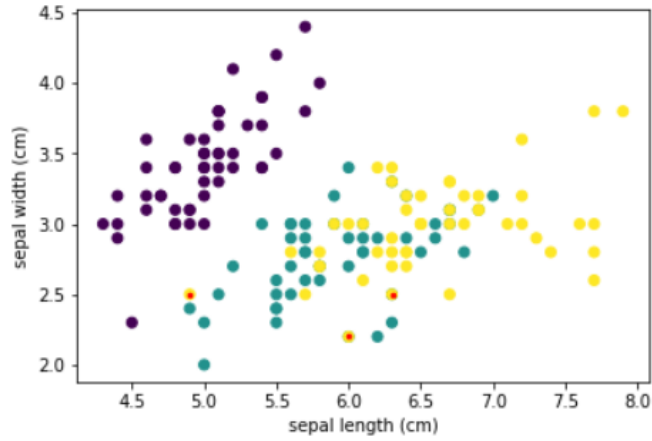
```

- See below that the misclassified points are those that are merged with other classes.

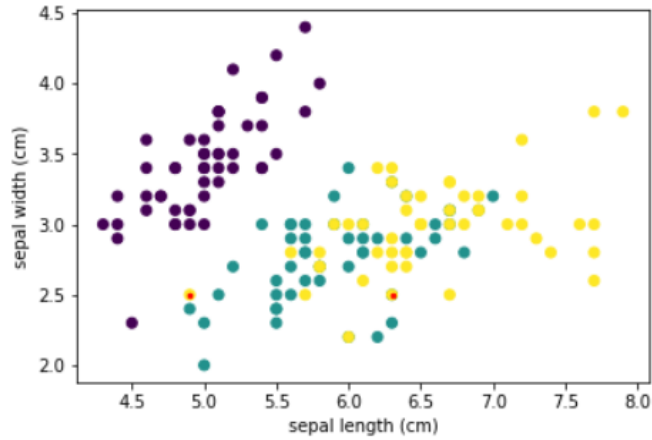
k = 1
[[6. 2.2 5. 1.5]
[6.3 2.5 4.9 1.5]
[6.3 2.8 5.1 1.5]
[4.9 2.5 4.5 1.7]]



k = 3
[[6. 2.2 5. 1.5]
[6.3 2.5 4.9 1.5]
[4.9 2.5 4.5 1.7]]

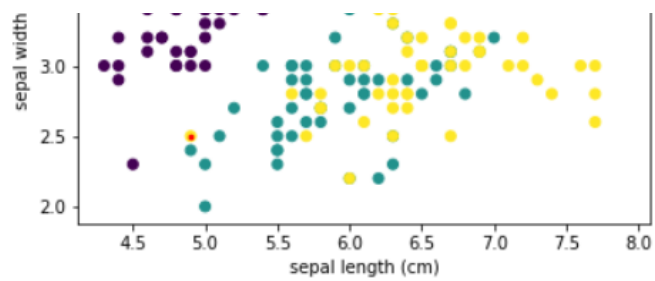


k = 5
[[6.3 2.5 4.9 1.5]
[4.9 2.5 4.5 1.7]]



k = 7
[[4.9 2.5 4.5 1.7]]



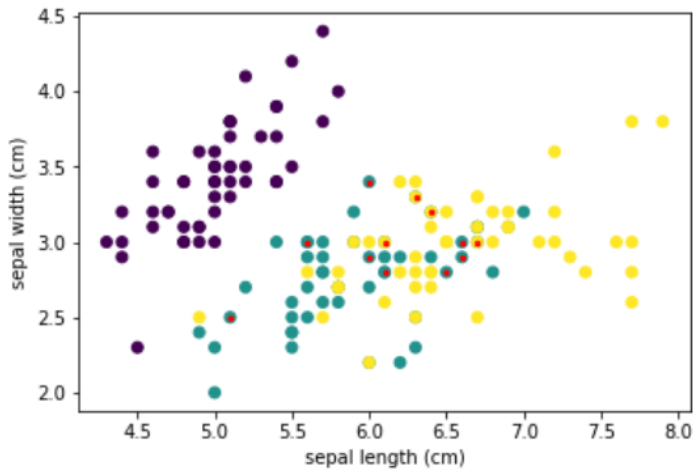


- However that was only the result from 1 random split of the data in training and test. To test the optimal k value properly we must calculate multiple accuracies of k over multiple train/test splits (and average each k accuracy over the different random splits). This is known as cross-validation
 - Look below the difference in random seeds for the same k

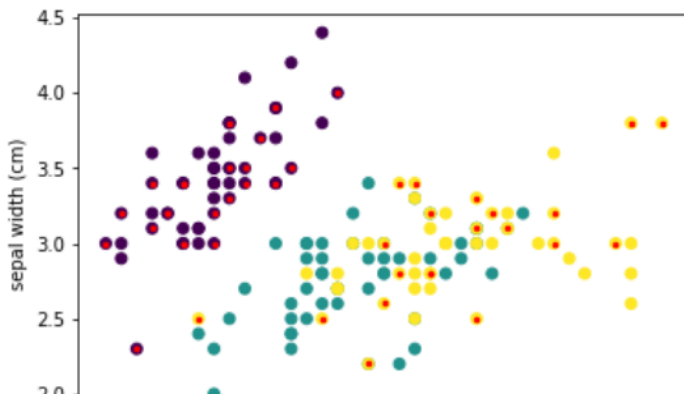
```
_test_split(iris.data, iris.target, test_size=0.4, random_state=seeds[0])  
test, predictions, len(Y_test))
```

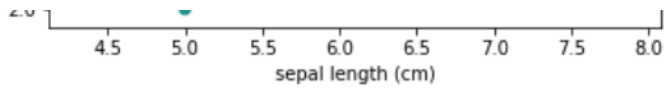
```
_test_split(iris.data, iris.target, test_size=0.4, random_state=seeds[1])  
test, predictions, len(Y_test))
```

```
k = 60  
[[5.1 2.5 3. 1.1]  
 [6.6 3. 4.4 1.4]  
 [6.3 3.3 4.7 1.6]  
 [6.4 3.2 4.5 1.5]  
 [6. 2.9 4.5 1.5]  
 [5.6 3. 4.5 1.5]  
 [6.7 3. 5. 1.7]  
 [6.1 2.8 4.7 1.2]  
 [6.1 3. 4.6 1.4]  
 [6.5 2.8 4.6 1.5]  
 [6.6 2.9 4.6 1.3]  
 [6. 3.4 4.5 1.6]]
```



```
[6.9 3.1 5.1 2.3]  
 [6.8 3.2 5.9 2.3]  
 [4.8 3.4 1.6 0.2]  
 [6.1 2.6 5.6 1.4]  
 [5.2 3.4 1.4 0.2]  
 [5.1 3.5 1.4 0.2]  
 [5.2 3.5 1.5 0.2]  
 [5.5 3.5 1.3 0.2]  
 [4.9 2.5 4.5 1.7]  
 [6.2 3.4 5.4 2.3]  
 [7.9 3.8 6.4 2. ]  
 [5.4 3.4 1.7 0.2]  
 [6.7 3.1 5.6 2.4]  
 [6.3 3.4 5.6 2.4]  
 [7.6 3. 6.6 2.1]  
 [6. 2.2 5. 1.5]  
 [4.3 3. 1.1 0.1]]
```





```

n_repetitions = 10
max_neighbours = 20
accuracies = np.zeros((max_neighbours, n_repetitions))
mean accuracies = np.zeros(max_neighbours)
seeds = [x for x in range(n_repetitions)]

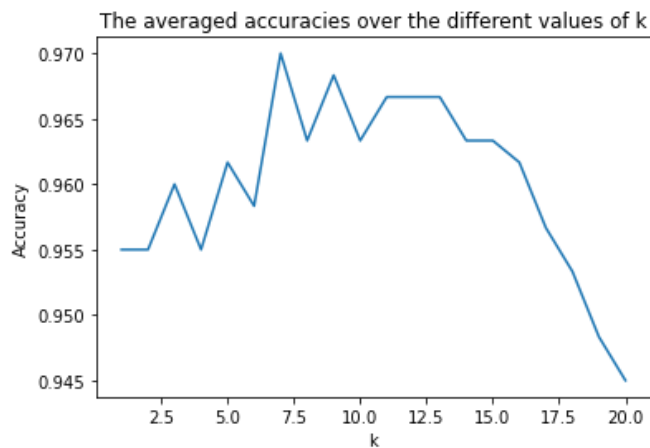
for i in range(n_repetitions):
    # Generate a new split of train and testset
    X_train, X_test, Y_train, Y_test = train_test_split(iris.data, iris.target, test_size=0.2, random_state=seeds[i])

    for k in range(1, max_neighbours + 1):
        # START ANSWER
        accuracies[k-1][i] = accuracy_score_self(Y_test, predict(X_train, X_test, Y_train, k))

mean accuracies = np.mean(accuracies, axis=1)
# END ANSWER

plt.plot(range(1, 21), mean accuracies)
plt.title('The averaged accuracies over the different values of k')
plt.xlabel('k')
plt.ylabel('Accuracy')
plt.show()

```



Learning curve

- For a learning curve, we plot the number of samples (x-axis) in the train set against the accuracy (y-axis).

```

k = 9

X_train, X_test, Y_train, Y_test = train_test_split(iris.data, iris.target, test_size=0.2, random_state=seeds[i])

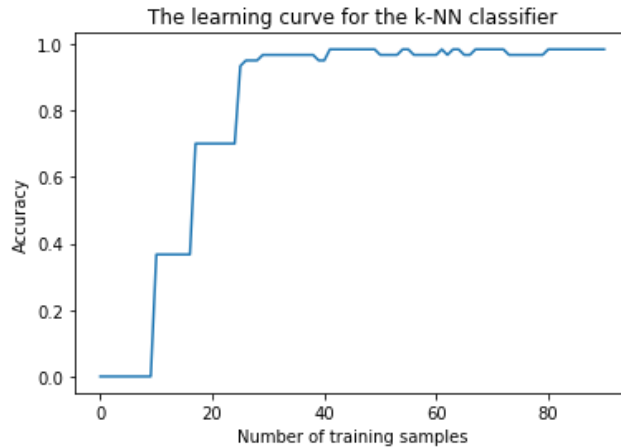
total_samples = X_train.shape[0]
# Set up array to store accuracies
accuracies = np.zeros(total_samples + 1)

# We want to learn with at least k samples and up to the size of the train set
for i in range(k, total_samples):
    predictions = predict(X_train[:i], X_test, Y_train[:i], k)
    accuracies[i + 1] = accuracy_score(Y_test, predictions)

# Plot learning curve
plt.plot(range(k, total_samples + 1), accuracies)
plt.title('The learning curve for the k-NN classifier')

```

```
plt.xlabel('Number of training samples')
plt.ylabel('Accuracy')
plt.show()
```

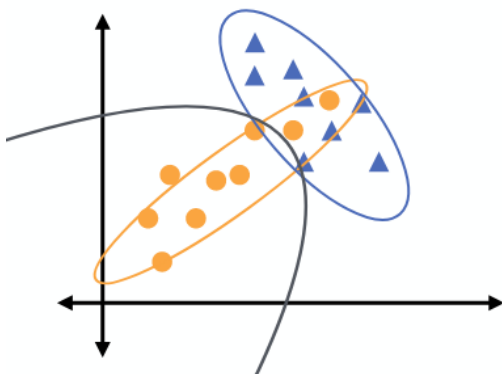


Linear classifiers

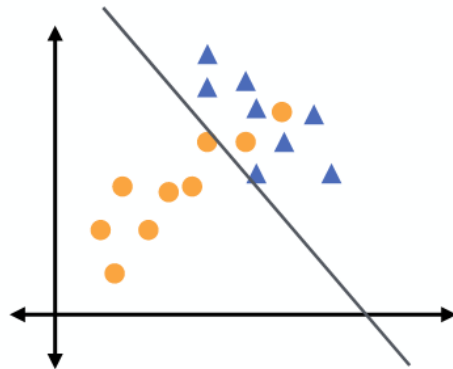
- Linear (Discriminative) as opposed to bayesian (generative) cannot “generate”/simulate new data points as we don’t know nor haven’t modeled the class distributions($P(X|Y)$).
- Instead linear models just model $P(Y|X)$ directly (and linearly) as all we care about is the decision boundary
 - The first decision boundary we make might be arbitrary, but it can be easily improved with the gradient descent algorithm

Generative vs. Discriminative

Model $P(X|Y)$ and $P(Y)$, or $P(X,Y)$ to derive $P(Y|X)$



Model $P(Y|X)$ or $h(X)$ directly



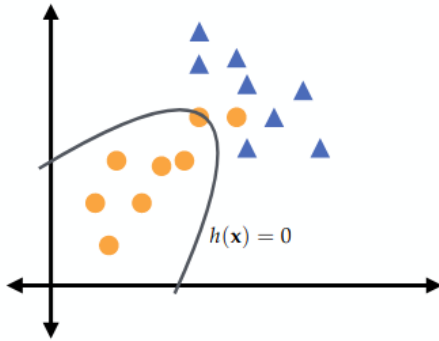
- Why linear?
 - Simple
 - Optimization (minimization) is possible and fast
 - Interpretable
 - Often a reasonable local approximation given limited data

Minimizing the risk

- Let $h(x)$ be the function that determines the class of input vector x
 - also known as “hypothesis function” or “discriminant function” (that maps input to a decision value)

- Let $L(h(x), Y)$ be the Loss/cost function that measures how well the predicted decision value matches the real outcome (Y)
- In theory minimizing that loss function would require us to know the actual class distributions (and calculate their averages (expectect values)), which in linear models we don't know.
 - Instead we just manually check the average value of all loss functions of all "versions" of the hypothesis function
 - Then select the hypothesis version (h) function with the minimum loss

The Empirical Risk



$$\min_{h \in \mathcal{H}} \frac{1}{N} \sum_{i=1}^N L(h(\mathbf{x}_i), y_i)$$

x, y : observed values corresponding to the input features and outcome of interest

h : hypothesis function that maps input to a decision value

L : loss/cost function that measures how well the predicted outcome matches the real outcome

- Breakdown of the minimization algorithm steps:

Classifier Construction using Empirical Risk Minimisation

$$\min_{h \in \mathcal{H}} \frac{1}{N} \sum_{i=1}^N L(h(\mathbf{x}_i), y_i)$$

1. Define the class of possible functions

2. Define a cost (loss) function to measure the "quality" of each hypothesis

3. Measure the average cost on the training data

4. Find the function that minimises this cost

Hypotheses

- When the outcome is continuous (regression), $h(x)$ can directly correspond to the function we want to find (i.e. the expected price).
 - We don't map input x to a class but to a continuous value.
- In classification $h(x)$ is the discriminant function which gives a real-valued output (such as the bayes probability)

- To go from this output to a class decision we still need to set a cut off, i.e.:

- $$\begin{cases} c_1 & \text{if } h(x) > 0 \\ c_2 & \text{if } h(x) \leq 0 \end{cases}$$

Hypothesis class: Linear

- $$h(x) = x^T w + w_0$$
 - Decision function is a weighted linear combination of the input features plus a constant (intercept/bias/threshold)
- The bias term may be added to the weight vector by adding a constant feature, (just a notational trick):

- $$h(x) = x^T w + w_0 = \begin{bmatrix} x \\ 1 \end{bmatrix}^T \begin{bmatrix} w \\ w_0 \end{bmatrix}$$

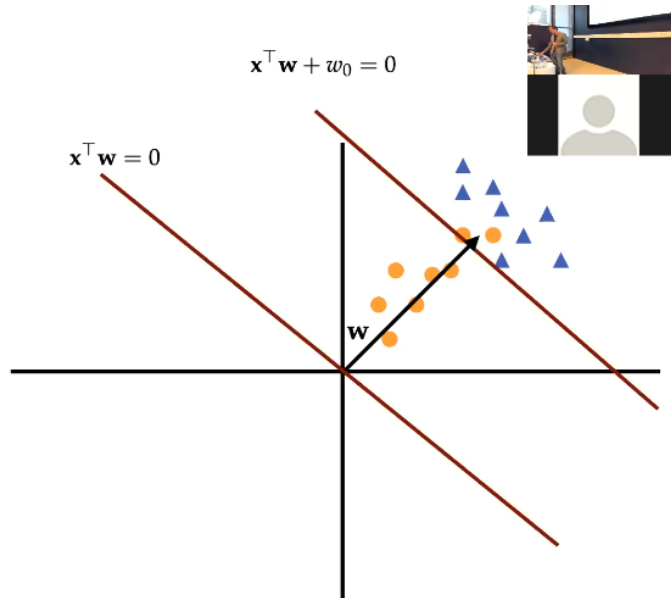
- A linear hypothesis class is a unique combination of different weights entries for the weight vector

Linear decision boundaries (2 feature vectorspace case)

- The weights vector is perpendicular to the decision boundary hyperplane (hyperplane as a subspace of 1 dimension smaller than it's container)
- The bias constant shifts the decision boundary

Linear Decision Boundaries

Decision boundary is where $x^T w + w_0 = 0$. This means w is orthogonal to every vector "within the decision boundary". For a 2D problem, this means it has to be a hyperplane of dimension 1 (a line).



- All the $h \in H$ are all the different weight vector (w) that we consider to search among them the one that has the minimum cost

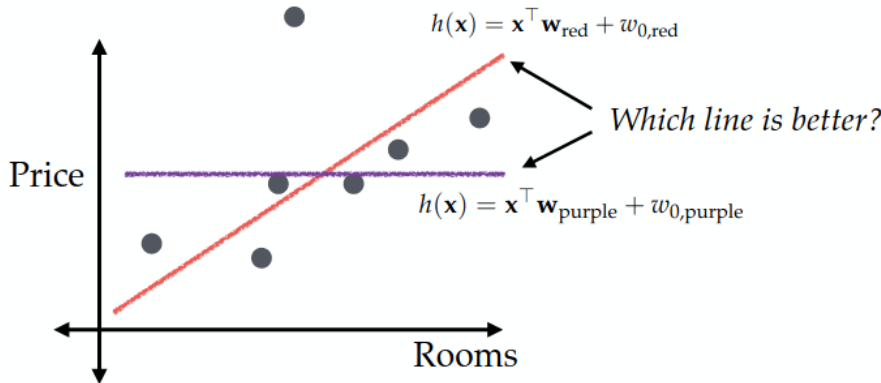
Linear regression

Regression Problem

We need to define a loss:

$$\min_{\mathbf{w}, w_0 \in \mathbb{R}^{D+1}} \frac{1}{N} \sum_{i=1}^N L(\mathbf{x}_i^T \mathbf{w} + w_0, y_i)$$

Task: predict the price of a house, given the number of rooms (note: continuous outcome, not discrete classes)



Regression Losses

We need to define a loss:

$$\min_{\mathbf{w}, w_0 \in \mathbb{R}^{D+1}} \frac{1}{N} \sum_{i=1}^N L(\mathbf{x}_i^T \mathbf{w} + w_0, y_i)$$

Consider the difference between the predicted value and the true value:

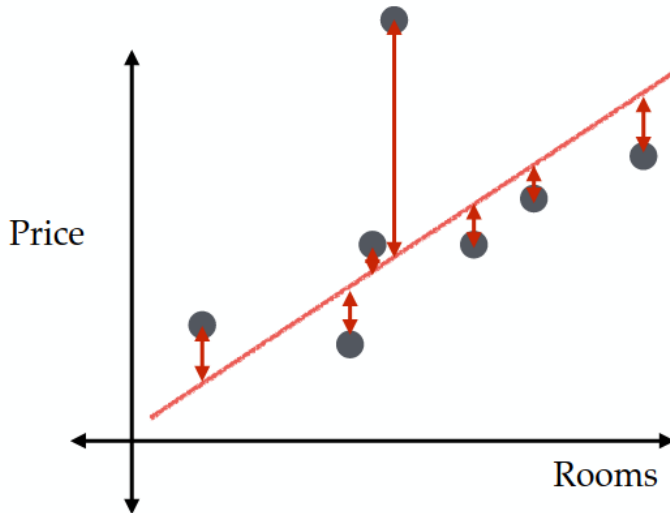
$$(\mathbf{x}_i^T \mathbf{w} + w_0 - y_i)$$

We can consider the absolute difference:

$$|\mathbf{x}_i^T \mathbf{w} + w_0 - y_i|$$

Or penalise large differences more strongly by taking the squared difference:

$$(\mathbf{x}_i^T \mathbf{w} + w_0 - y_i)^2$$



- Mean squared error, mean absolute error etc are all examples of loss functions

Minimizing risk

- We want to find the parameters (weight vector \mathbf{w}) that minimize the risk (that have the lowest cost)
- We could bruteforce all values of \mathbf{w} and choose the smallest cost (not practical)
- We could take the derivative of the loss function and equal it to 0 (might be a bit of a hussle)
- Alternatively we can start at some \mathbf{w} and keep making small changes to improve it (gradient descent)
 - It's like a combination of bruteforcing all possible values (but only those surrounding your last position) and taking the derivative. So instead of taking all values around you take the "derivative" (just the difference with the current position) of all values around you and choose the one with the deepest decrease.
- For simple linear regressions we can just use the "normal equation" (linear algebra) which also finds the minimum:

$$\circ \quad X^T X \mathbf{w} = X^T \mathbf{y}$$

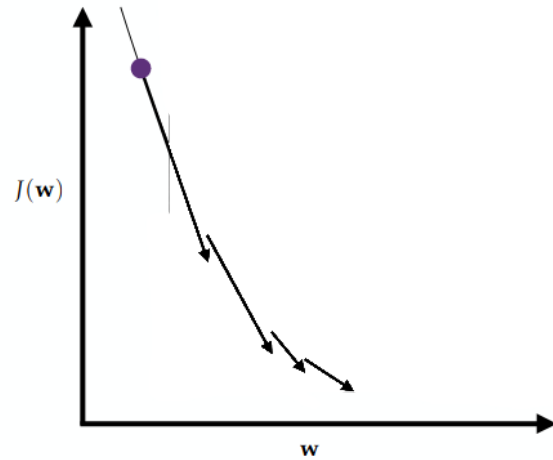
Gradient descent

- Used in procedures where the normal equation cannot be used.

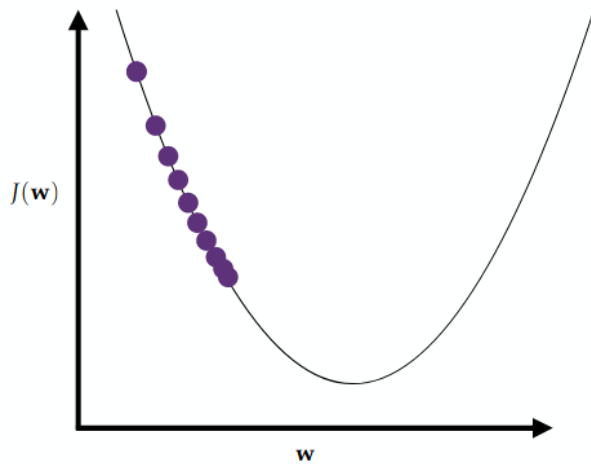
Gradient Descent

$$w_j^{t+1} = w_j^t - \alpha \left. \frac{\partial J(\mathbf{w}, w_0)}{\partial w_j} \right|_{\mathbf{w}, w_0 = \mathbf{w}^t, w_0^t}$$

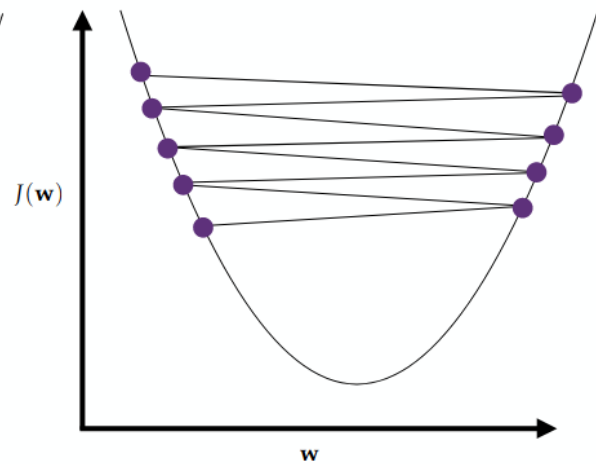
- Look at the slope, and follow the direction of the steepest slope
- How far do I go once I choose the direction (stepsize)? How many "steps" should I take?



Stepsize α



Too Small?



Too Large?

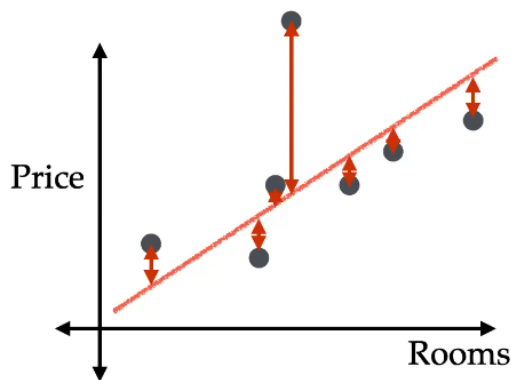
Gradient Descent Procedure

Given an objective function J , learning rate (step size) α , and number of iterations T .

1. Pick a starting value (for instance, random) for \mathbf{w} and w_0
2. For T iterations, for all $t = 0, 1, \dots, D$, do:

$$w_j^{t+1} = w_j^t - \alpha \left. \frac{\partial J(\mathbf{w}, w_0)}{\partial w_j} \right|_{\mathbf{w}, w_0 = \mathbf{w}^t, w_0^t}$$

Gradient descent in our Regression problem



$$J(\mathbf{w}, w_0) = \frac{1}{N} \sum_{i=1}^N (\mathbf{x}_i^\top \mathbf{w} + w_0 - y_i)^2$$

Derive the gradient:

$$\frac{\partial J(\mathbf{w}, w_0)}{\partial w_j} = \frac{1}{N} \sum_{i=1}^N 2(\mathbf{x}_i^\top \mathbf{w} + w_0 - y_i) x_i^{(j)}$$

$$\frac{\partial J(\mathbf{w}, w_0)}{\partial w_0} = \frac{1}{N} \sum_{i=1}^N 2(\mathbf{x}_i^\top \mathbf{w} + w_0 - y_i)$$

Stochastic Gradient Descent

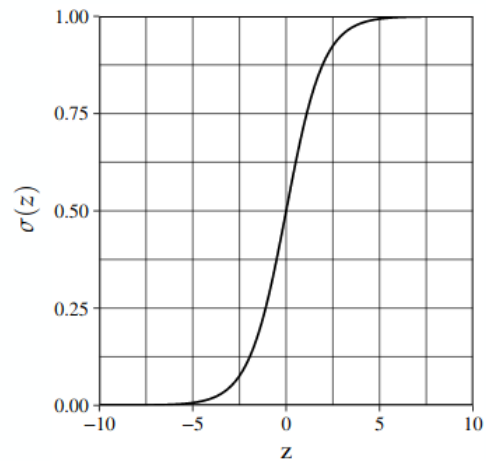
- To calculate the full gradient, we need to **sum over all** objects, before we take a step
- Instead we could estimate the gradient using one, or a few objects, and take a step using this estimate of the gradient
- The step is less "precise", but we can take many more steps in the same amount of time
- *Epoch*: visiting all the data once
- So in stochastic gradient descent, we do updates within an epoch, while regular gradient descent does only one update per epoch.

Logistic regression (logistic classifier)

- It's actually not a regression problem, it tries to solve a classification problem
- It emerges from the fact that the classical loss function of % of wrong classes assigned does not fit well with the gradient descent algorithm
 - Slightly modifying the discriminant function in any direction will keep providing the same number of correct/incorrect predictions, which makes the algorithm stall on that position.
 - The logistic regression therefore has a different approach: find a function that approximates the likelihood (of a correct assignment), (which does not need to calculate the class conditional, the priors and then bayes, for all predictions) using a given classifier (a given weight vector and bias constant)
 - The change in likelihood does react to slight modifications of the discriminant function! (which fits well with the gradient descent algorithm)
 - $h(x) = x^T w + w_0 \neq p(y|x)$ but it does return a continuous value between [0,1]

Logistic function

$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$



This is known as the logistic function, an example of a sigmoid function

- With $z = h(x) = h(x, w, w_0)$
N Bernoulli trials. The estimated probability of observing label y_i for object i :

$$\begin{cases} \sigma(h(\mathbf{x}_i)), & \text{if } y_i = 1 \\ 1 - \sigma(h(\mathbf{x}_i)), & \text{if } y_i = 0 \end{cases}$$

Combining this, the likelihood is

$$L(h) = \prod_{i=1}^N \sigma(h(\mathbf{x}_i))^{y_i} (1 - \sigma(h(\mathbf{x}_i)))^{1-y_i}$$

- We could see that $p(y|x) \approx \sigma(h(x, w, w_0)) = \frac{1}{1 + \exp(-h(x))} = \frac{1}{1 + \exp(-x^T w - w_0)}$
 - For class predictions, we eventually need to have a cut-off minimum probability of $h(x)$ to assign x to the class
- The reason we are transforming $h(x)$ to a logistic function even though $h(x)$ already works as a classifier is because $h(x)$ as a classifier is discrete and $h(x)$ raw could be any number and we just want to have a continuous output between 0 and 1 (so it resembles the posterior probability). Furthermore, the logistic function also has a nice property that allows us to get a density function and likelihood function
 - The likelihood function penalizes gray assignments and rewards black and white assignments. This is very sensitive to gradient descent tweaks in the weight vector (which

is what we're looking for) and this enable us to move towards the best weight vector.

- Instead of having a loss function (negative we want to minimize), we have a likelihood function (positive we want to maximize) (likelihood that the objects are assigned to the correct classes)

Logistic regression loss function (Objective function)

$$L(h) = \prod_{i=1}^N \sigma(h(\mathbf{x}_i))^{y_i} (1 - \sigma(h(\mathbf{x}_i)))^{1-y_i}$$

The likelihood is a measure of how well the estimated probabilities explain the observed labels, so we want to maximise this likelihood

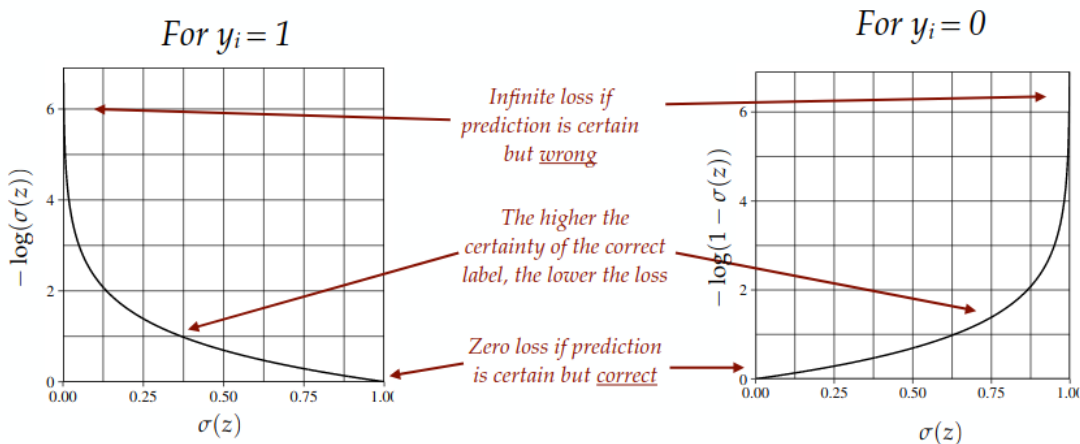
This gives the same optimum as minimizing the negative log likelihood:

$$J(h) = - \sum_{i=1}^N y_i \log[\sigma(h(\mathbf{x}_i))] + (1 - y_i) \log[1 - \sigma(h(\mathbf{x}_i))]$$

- The adaption with the log is to get easier to compute numbers and the minus sign is to turn it back to a minimization problem
- So while L(h) is a think we want to maximize J(h) is a cost that we want to minimize

Logistic Loss, Visualized

$$J(h) = - \sum_{i=1}^N y_i \log[\sigma(h(\mathbf{x}_i))] + (1 - y_i) \log[1 - \sigma(h(\mathbf{x}_i))]$$



Logistic regression hypothesis class

- Same as with the linear regression, we consider all possibilites of weight vectors for $h(x) = x^T W + w_0$

Gradient Descent Procedure (to minimize loss function)

- The derivative of $J(w) = - \sum_{i=1}^N y_i \log(\sigma(x_i^T w)) + (1 - y_i) \log(1 - \sigma(x_i^T w))$ is:

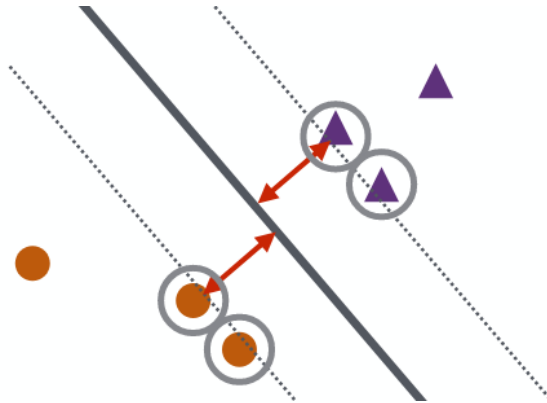
o

$$\nabla_w J(w) = - \sum_{i=1}^N (y_i - \sigma(x_i^T w)) x_i$$

- Which is equivalent to the difference between the model prediction and the actual class times the vector object

Support vector machines

- We only cover the linear version (classifier)
 - We assume that classes are linearly separable
- Beautiful math and popular in the 90's but computationally expensive and hard to optimize



As long as all objects are correctly classified, maximize the size of the “margin”: the distance of the closest points to the decision boundary

The objects on the margin are the **support vectors**: they completely define the decision boundary. The other objects can be moved (outside the margin) without changing the classifier.

- We are trying to find a decision boundary that is robust to changes in the data set (that is generalizable), and hence why we try to maximize the space between the support vectors and the decision boundary
- The classification is based on the sign of:

- $x_i^T w + w_0$

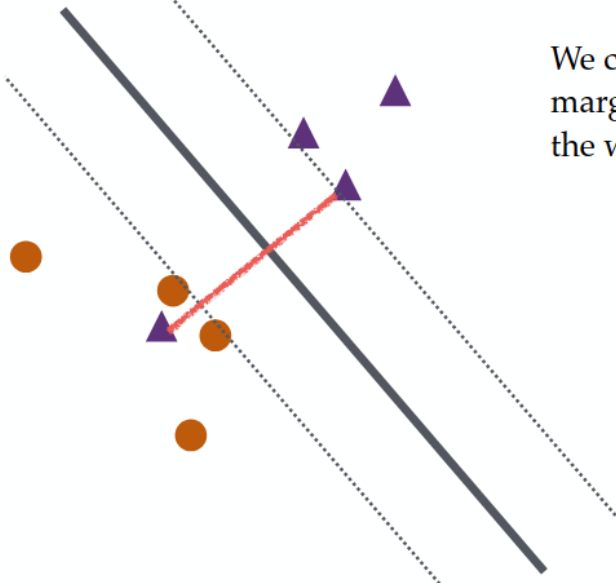
- However, we are gonna also include a margin threshold M such that the ones that are too close to the decision boundary do not get assigned to either class, and the ones that get classified follow:
 - $x_i^T w + w_0 \geq M$ if $y_i = +1$
 - $x_i^T w + w_0 \leq -M$ if $y_i = -1$
- Since we a priori had the freedom to scale w to reach M , we have to enforce the constraint to fix w scale such that the closest object has decision value -1 or 1 (which means $M = 1$)
- Remember that we want to maximize the margin (to be robust). The complete margin (thus from one antagonist support vector to the other instead of just to the decision boundary) is $2 * M$
 - Remember that $M=1$, but also remember that we need to make sure that regardless of the scale of w , the distance between the support vector and the decision boundary remains one. To do so we divided M over the length of w
- We end up with maximizing $\frac{2}{\|w\|}$. Which is equivalent to:

- $$\min_w \frac{1}{2} \|w\|^2$$

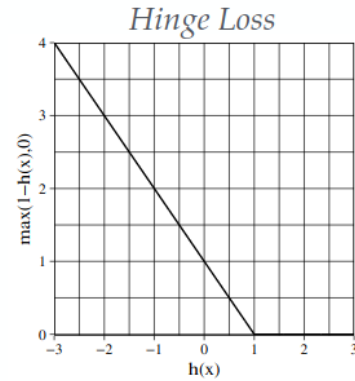
- Subject to the priorly established margin constraints

Soft margin support vector machine and (hinge) loss function

What if the data is not linearly separable?



We could allow for “violations” of the margin: penalize how far the object is on the wrong side of the margin.



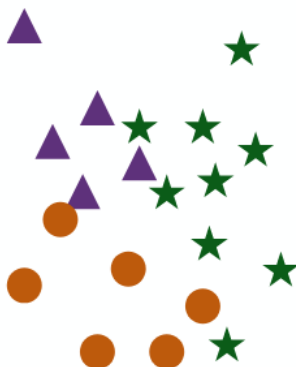
$$\min_{\mathbf{w}, w_0} \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^N [1 - y_i (\mathbf{x}_i^T \mathbf{w} + w_0)]_+$$

Where $[\cdot]_+$ indicates the value of the function if the value is positive, and 0 otherwise.

- Where C is the penalty constant and $\min_w \frac{1}{2} \|w\|^2$ is a loss term without data also known as a regularization term

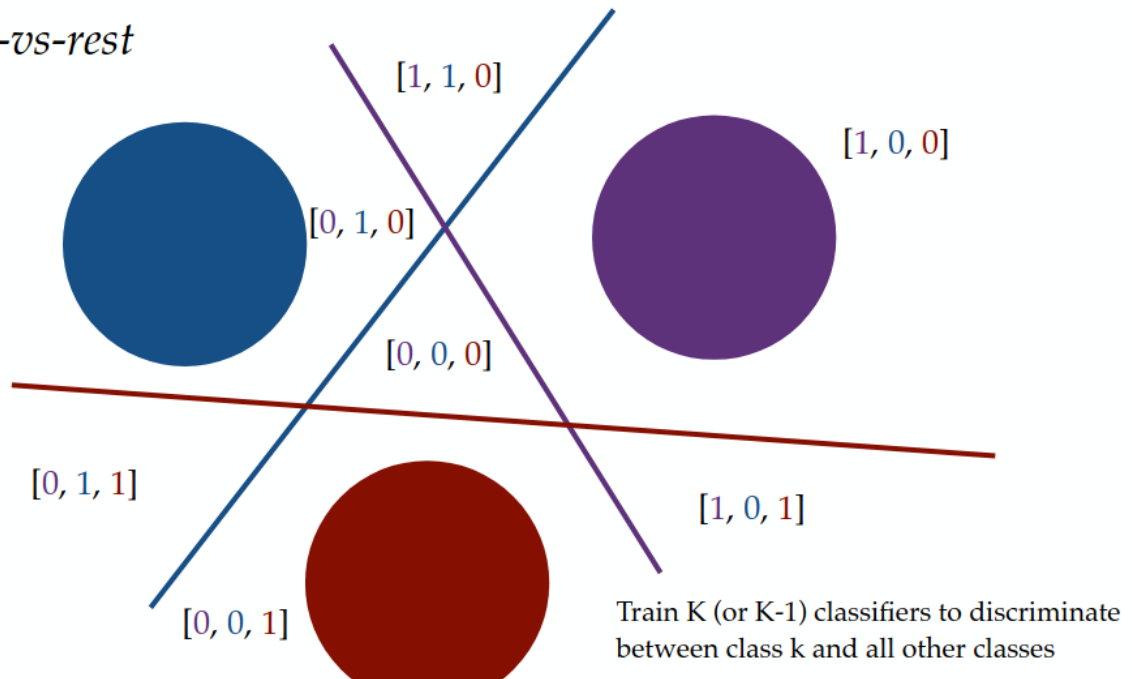
Multiclass classification

Multiclass Classification

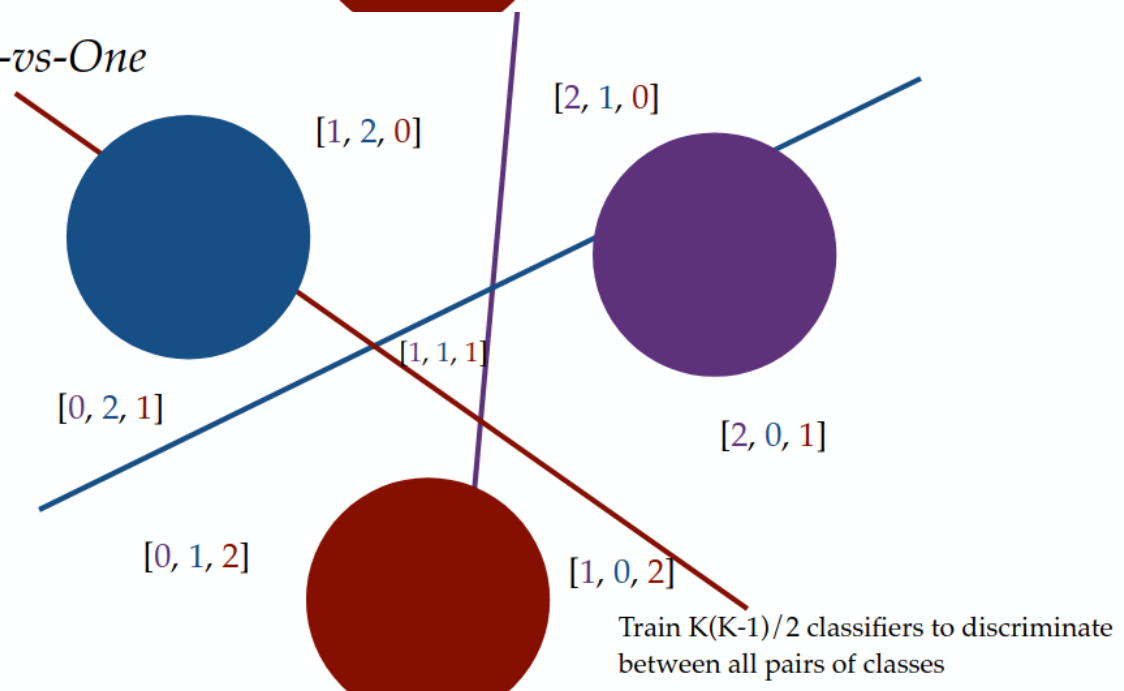


- More than 2 classes
- For generative models: model the extra $P(X | Y=k)$'s
- Discriminative models: two approaches:
 - Algorithm specific adaptations: directly construct a multi-class classifier
 - General techniques: combining multiple 2-class classifiers

One-vs-rest



One-vs-One



- Instead of using the hard classifiers (discrete class = this or class = that), which for multi class lead to awkward overlaps, just use the highest posterior conditional probability to determine the class

Lab 4: Logistic regression classifier

- We have n training images, each x_i is a 64×1 vector representing the grayscale values for each pixel in the 8×8 px image.
- Each image is associated with a discrete label $y_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- We want to derive an hypothesis function that can predict the label of any new image x
- We want to measure the (log) likelihood of how good the classifier is
- We want to find the weight vector θ that maximizes the log likelihood

Setup

```
import scipy
import sklearn
import numpy as np
import matplotlib.pyplot as plt
```

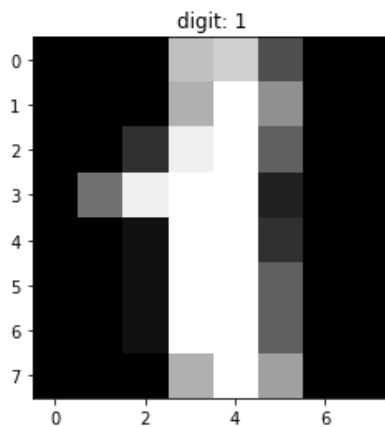
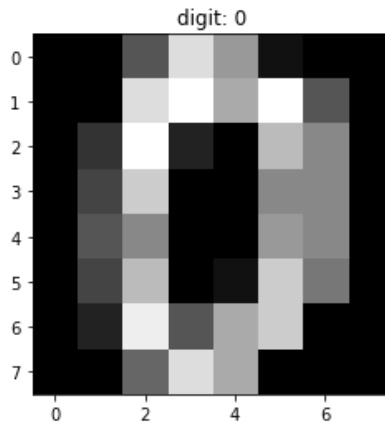
MNIST database

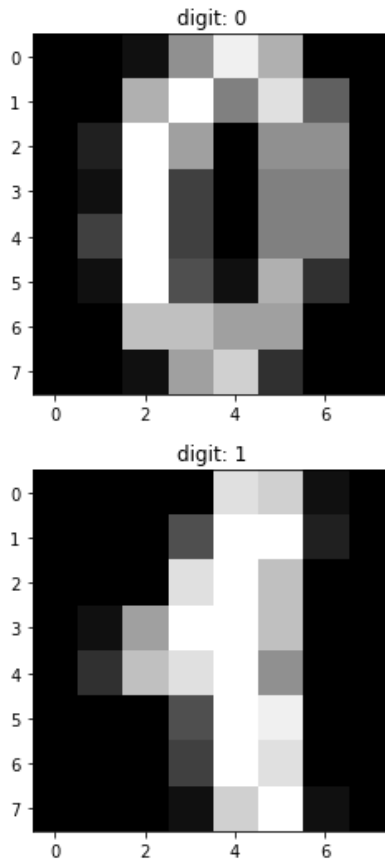
- Snippet below shows multiple versions of digits 0 and 1 of the MNIST database included in Scikit-learn
 - Although we have have 10 different classes (digits), we will first focus on the binary classification of the digits 0 and 1

```
# Load the digits with 2 classes (0 and 1)
binary_digits = datasets.load_digits(n_class=2)
binary_digits_images = binary_digits.images
binary_digits_labels = binary_digits.target

'''
all_digits_images is a numpy array where:
- the first index is the index of individual images
- the second index corresponds to the row of the pixel
- the third index corresponds to the column of the pixel
i.e.: all_digits_images[image_index,row,column]
the values of the pixels are values between 0 (black) and 16 (white)
'''

for i in range(4):
    digit_image = binary_digits_images[i,:,:]
    plt.figure()
    plt.gray()
    plt.title("digit: " + str(binary_digits_labels[i]))
    plt.imshow(digit_image)
```





Basic features

- We start with two basic features: crude measures of the length and width of the digit.
 - For widths, we measure this by taking the maximum values for every column and then taking the average of these values.
 - For lengths, we take the maximum value of each row and then we average over these values.
- Once we can classify based on simple features, we will extend our classifier to include more features.

```
# width: average of the column-wise max values
widths = np.zeros(len(binary_digits_images))
# START ANSWER
for i in range(len(binary_digits_images)):
    widths[i] = np.mean(np.amax(binary_digits_images[i], axis=0))
# END ANSWER

# length: average of the row-wise max values
lengths = np.zeros(len(binary_digits_images))
# START ANSWER
for i in range(len(binary_digits_images)):
    lengths[i] = np.mean(np.amax(binary_digits_images[i], axis=1))
# END ANSWER

assert (widths[:5] == np.array([8.5, 8.0, 9.25, 8.125, 9.5])).all()
assert (lengths[:5] == np.array([12.875, 15.625, 15.0, 15.75, 14.875])).all()
```

- Let's combine these two arrays into one numpy array called `binary_digits_features`:

```
binary_digits_features = np.vstack((widths, lengths)).T
print(binary_digits_features[:10])
```

- Plot the features in a scatterplot and see what the data looks like:


```

def plot_scatter(features, labels, db_x = None, db_y = None):
    widths = features[:,0]
    lengths = features[:,1]

    # Separate the 2 classes
    widths_0 = widths[labels == 0]
    lengths_0 = lengths[labels == 0]
    widths_1 = widths[labels == 1]
    lengths_1 = lengths[labels == 1]

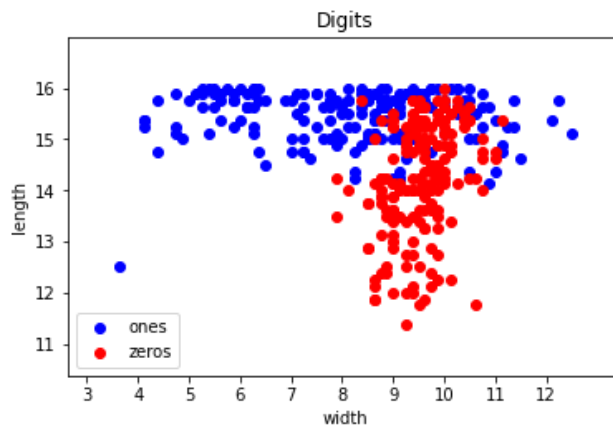
    # Plot
    plt.scatter(widths_1, lengths_1, c='blue', label='ones')
    plt.scatter(widths_0, lengths_0, c='red', label='zeros')

    # Extra code to plot the decision boundary
    # You won't be using this right away
    if not(db_x is None or db_y is None):
        plt.plot(db_x, db_y, label = "Decision_Boundary")

    plt.title('Digits')
    plt.xlabel('width')
    plt.ylabel('length')
    plt.axis('square')
    plt.xticks(np.arange(widths.min(), widths.max()+1, 1, dtype=int))
    plt.yticks(np.arange(lengths.min()-1, lengths.max()+ 1, 1, dtype=int))
    plt.xlim((widths.min()-1, widths.max()+1))
    plt.ylim((lengths.min()-1, lengths.max()+1))
    plt.legend(loc=3)
    plt.show()

plot_scatter(binary_digits_features, binary_digits_labels)

```



Linear classifier $h(x)$

We want to construct a classifier which, given digit features, decides whether the digit is a 1 or 0. To achieve this, we want to find a "decision function" $f(x)$, that, given an object represented by the vector x , returns a high value if the object belongs to class 1 and a small value if it belongs to class 0. To decide which class to assign an object to, we can set a threshold informing whether a given decision value is large enough. A linear classifier makes a particular choice for what the decision function $f(x)$ can look like: the function is a linear combination of the values of the features. For a one-dimensional dataset the decision takes the following form:

- $\text{if } \theta_1 \cdot x + \text{bias} > \text{threshold} \text{ classify as 1 else classify as 0}$
 - Here θ_1 , the bias and the threshold are the parameters that need to be tuned for the classifier to work properly.
- In case of the two features we have now, this classifier will look like
 - $\text{if } \theta^T x + \text{bias} > \text{threshold} \text{ return 1 else return 0}$

- with theta having each weight for each feature of x (thus theta and x having the same length)
- The classifier takes the dot product of these vectors and checks whether the obtained value exceeds the threshold value for a positive classification.
- We can include the threshold into the bias term
 - $\theta^T x + (bias - threshold) > 0$
 - We can also get rid of the bias - threshold notion and just call it bias such as $\theta^T x + bias > 0$

Logistic function $\sigma(z)$

- Instead of finding the θ that minimizes the classification error, we are going to construct a linear classifier that returns probabilities of objects belonging to the different classes ($p(y|x)$) and find the θ that maximizes *how well* these probabilities reflect the data we have observed.
- To do this, we first have to convert the values of our decision function $\theta^T x + bias$, into values between 0 and 1. These values will reflect the probability of the object to belong to class 1. For this conversion, we will use the logistic function:

- $$\sigma(\theta^T x + bias) = \frac{1}{1 + e^{-(\theta^T x + bias)}}$$

- See that when the bias is infinity large not only the classifier is obviously larger than 0 but sigma = 1
- When bias is + dot product are 0 (kissing the boundary), sigma = 0.5
- When bias is minus infinity, sigma = 0
- Another useful property of the derivative of the logistic function that we will use later on is the following:

- $$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

Posterior probabilities

- Indeed the sigma function that models the probability of the object belonging to class 1 or 0 is used to model the posterior probability:
 - $P(Y = 1|x) = \sigma(\theta^T x + bias)$
 - $P(Y = 0|x) = 1 - \sigma(\theta^T x + bias)$
- Using only the width feature, below the logistic function that encapsulates the probability of $P(Y=1|x)=h(x)=\sigma(\theta x+bias)$:

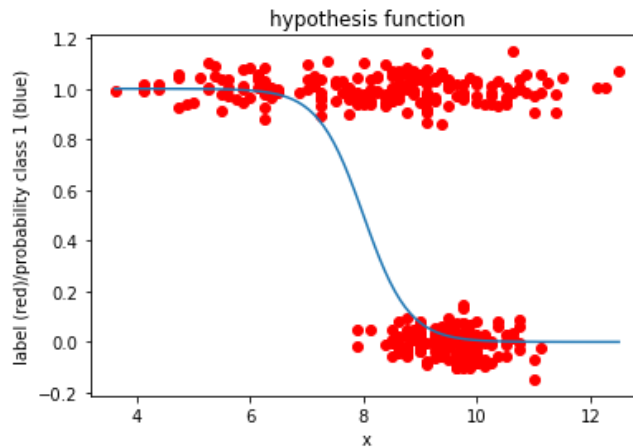
```
def plot_hypothesis(features, labels, theta, bias):
    # Some noise is added to better visualize the labels of the datapoints
    labels_noise = labels + np.random.normal(0, .05, labels.shape)

    widths = features[:,0]
    plt.scatter(widths, labels_noise, c='red')
    x = np.linspace(np.min(widths), np.max(widths), 100)

    sigmoid_1D = 1 / (1 + np.exp(-(theta*x + bias)))
    plt.title('hypothesis function')
    plt.xlabel('x')
    plt.ylabel('label (red)/probability class 1 (blue)')
    plt.plot(x, sigmoid_1D)
    plt.show()

# Try to find proper values for theta and bias
# Such that the sigmoid properly goes through both the datapoints with label 0 and 1
theta = -2.5
bias = 20
# START ANSWER
# END ANSWER
```

```
plot_hypothesis(binary_digits_features, binary_digits_labels, theta, bias)
```



- The temporary hypothesis function has $\theta = -2.5$ and $bias = 20$.
 - It will be optimized in the next sections

Simplifying notation

- Instead of having a separate bias term, we will incorporate into the weights and add an extra feature to x with 1

```
# This function adds an extra 1.0 to every feature vector
def add_one_features(data):
    return np.vstack((data.T, np.ones(len(data))))).T

binary_digits_features_prime = add_one_features(binary_digits_features)
print(binary_digits_features_prime[:10])
```

- The logistic function ($p(y|x)$) with the updated vectors:

```
# Implement the hypothesis function so that it works for thetas/features of arbitrary length
def hypothesis(x, theta):
    """
    Calculate the hypothesis function for every datapoint in x
    :param x: numpy array of size (n, d) where n is the number of samples
    and d is the number of features per sample including the 1 extra feature
    :param theta: numpy array of size (d,)
    :return: predicted probability.
    """
    # START ANSWER
    sigmoid = np.zeros(len(x))
    for i in range(len(x)):
        sigmoid[i] = 1/(1 + np.exp(-(x[i] @ theta)))
    # END ANSWER
    return sigmoid

x = binary_digits_features_prime
# To test our hypothesis function, we set three different theta vectors
# ALL 1
theta_ones = np.ones(3)
# ALL 0
theta_zeros = np.zeros(3)
# ALL -1
theta_min_ones = -5 * np.ones(3)

# And apply the prediction
hypothesis_ones = hypothesis(x, theta_ones)
hypothesis_zeros = hypothesis(x, theta_zeros)
hypothesis_min_fives = hypothesis(x, theta_min_ones)
```

```
# Output for each theta vector
# expected = 1.0
print("Prediction ones: {}".format(hypothesis_ones[:5]))
# expected = 0.5
print("Prediction zeros: {}".format(hypothesis_zeros[:5]))
# expected = ~0
print("Prediction minus fives: {}".format(hypothesis_min_fives[:5]))

assert np.isclose(hypothesis_ones, 1).all()
assert np.isclose(hypothesis_zeros, 0.5).all()
assert np.isclose(hypothesis_min_fives, 0).all()
```

Likelihood function

- We can compute the likelihood $L(\theta)$ for the entire dataset by computing the product of the likelihood of all samples:

- $$L(\theta) = \prod_{i=1}^N \sigma_{\theta}(h(x_i))^{y_i} (1 - \sigma_{\theta}(h(x_i)))^{1-y_i}$$

- Or we could compute an equivalent but more computation friendly likelihood function, the log likelihood (recall that $\log(ab) = \log a + \log b$ and $\log(a^b) = b \log a$):

- $$\log L(\theta) = \sum_{i=1}^N y_i \log(\sigma_{\theta}(h(x_i))) + (1 - y_i) \log(1 - \sigma_{\theta}(h(x_i)))$$

- As the log function is an monotonic increasing function, this approach will also lead to maximizing the likelihood itself

```
near_0 = 1e-16
near_1 = 1.0 - near_0

def log_likelihood(h_x, y):
    """
    Computes the log likelihood of your classifier.
    :param h_x: numpy array of predicted probabilities.
    :param y: numpy array of actual labels (positive (1) or negative (0)).
    :return: The log likelihood.
    """
    log_likelihood = 0
    # START ANSWER
    log_likelihood = np.sum(
        y*np.log(np.where(h_x == 0, near_0, h_x))
        +(1-y)*np.log(1-np.where(h_x == 1, near_1, h_x)))
    # END ANSWER
    return log_likelihood

# These predictions should do very well
h_x1 = np.array([0.01, 0.01, 0.99, 0.99])
y1 = np.array([0, 0, 1, 1])
l11 = log_likelihood(h_x1, y1)
print(l11)

# These predictions should do ok
h_x2 = np.array([0.2, 0.1, 0.9, 0.8])
y2 = np.array([0, 0, 1, 1])
l12 = log_likelihood(h_x2, y2)
print(l12)

# These predictions should do bad
```

```

h_x3 = np.array([0.9, 0.8, 0.99, 0.3, 0.1])
y3 = np.array([0, 0, 1, 1, 1])
l13 = log_likelihood(h_x3, y3)
print(l13)

assert np.isclose(l11, -0.040201)
assert np.isclose(l12, -0.657008)
assert np.isclose(l13, -7.428631)

# There might be warnings from numpy regarding division by zero and invalid value.
# You can solve this by replacing 0/1 values with near_0/near_1 values with the np.where
h_x4 = np.array([0.0, 0.1, 1.0, 0.95])
y4 = np.array([0, 0, 1, 1])
l14 = log_likelihood(h_x4, y4)
print(l14)
# h_x5 0.0 will become 1e-16 so you can divide with that.
h_x5 = np.array([1.0, 0.99, 0.0, 0.01])
y5 = np.array([0, 0, 1, 1])
l15 = log_likelihood(h_x5, y5)
print(l15)

assert np.isclose(l14, -0.156653, rtol=0.5)
# Due to the wrong predictions, this likelihood is very low
assert l15 < -10.0

```

- To visualise the effect of θ on the likelihood function, you can run the code below. Here only the width is used for visualisation purposes.

```

# Use only the width feature
width_features = binary_digits_features[:,0]
width_features_prime = add_one_features(width_features)
binary_digits_labels

# Axis limits to plot
min_theta_0 = -20.0
max_theta_0 = 5.0
min_bias = -30.0
max_bias = 150.0

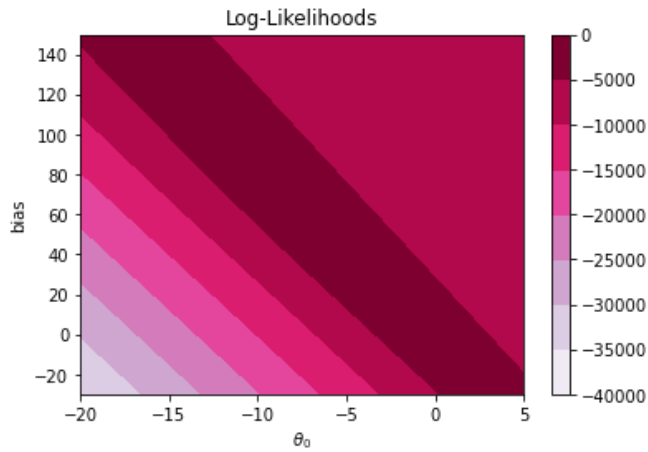
# Resolution for both axis
N = 50
thetas_0 = np.linspace(min_theta_0, max_theta_0, N)
biases = np.linspace(min_bias, max_bias, N)

# 2D array with Log Likelihoods to be filled
log_likelihoods = np.zeros(shape=(len(biases), len(thetas_0)))
# Fill the 2D array
for i_theta_0, theta_0 in enumerate(thetas_0):
    for i_bias, bias in enumerate(biases):
        # Construct theta
        theta = np.array([theta_0, bias])
        h_x = hypothesis(width_features_prime, theta)
        ll = log_likelihood(h_x, binary_digits_labels)
        log_likelihoods[i_bias, i_theta_0] = ll

# Plot Log Likelihoods
X, Y = np.meshgrid(thetas_0, biases)
cs = plt.contourf(X, Y, log_likelihoods, cmap="PuRd")
plt.title('Log-Likelihoods')
plt.xlabel(r'$\theta_0$')
plt.ylabel('bias')
plt.colorbar(cs)

plt.show()

```



Gradient ascent

- We will reformulate the gradient descent algorithm and simply apply it in the opposite direction. This is what that looks like:
- Set θ to a vector of random values.
- For each training epoch:
 - Compute the gradient of the *log likelihood* function for θ and training samples.
 - The derivative of $L(\theta_j)$ for a a specific θ_j configuration (while keeping other variables as constants (y and x)) is:

$$\frac{\partial}{\partial \theta_j} \ell(\theta) = (y - h_{\theta}(x))x$$

```
def calculate_gradients(theta, x, y):
    """
    Calculate the gradient for every datapoint in x
    :param theta: numpy array of theta
    :param x: numpy array of the features
    :param y: the label (positive (1) or negative (0))
    :return: The gradients for every datapoint in x
    """
    gradients = np.zeros((len(x), len(theta)))
    # START ANSWER
    gradients = x * ((y-hypothesis(x, theta)).reshape(len(y),1))
    # END ANSWER
    return gradients

theta = np.array([1,1.5,2.5])
x = np.array([[ -10,5,1],[0.5,1,1]])
y = np.array([0,1])
gradients = calculate_gradients(theta, x, y)
print(gradients)

assert np.isclose(gradients[0], np.array([5.0, -2.5, -0.5])).all()
assert np.isclose(gradients[1], np.array([0.00549347, 0.01098694, 0.01098694]), atol= 0.
```

- Adjust θ in the direction of the gradient potentiated by the step size α

- $$\begin{aligned} \theta_j &:= \theta_j + \alpha \frac{\partial}{\partial \theta_j} \ell(\theta) \\ &= \theta_j + \alpha (y - h_{\theta}(x))x \end{aligned}$$

- With `:=` denoting assignment (of new value) rather than predicating that (the current) θ equals (the new) definition

- In this context $h(x)$ is actually $\sigma(h(x))$

```
def apply_gradient(theta, gradient, alpha):
    """
    Applies the gradient step to theta and returns an adjusted theta.
    :param theta: current theta array of size (d,)
    :param gradient: the gradient array of (d,)
    :param alpha: learning rate
    :return: the updated theta array of size (d,)
    """
    updated_theta = theta
    # START ANSWER
    updated_theta = theta + alpha * gradient
    # END ANSWER
    return updated_theta

theta = np.array([1,2,3])
gradient = np.array([10,-10,5])
alpha = 0.1
updated_theta = apply_gradient(theta, gradient, alpha)
print(updated_theta)

assert (updated_theta == np.array([2,1,3.5])).all()
```

- Repeat from step 2 until convergence or a set number of epochs.

```
def train_theta(features, labels, n_epochs=200, theta=None, alpha = 0.1):
    assert len(features) == len(labels)

    num_features = len(features[0])
    num_items = len(features)
    # Set theta to initial random values
    # Initialize theta randomly if it's not provided
    if theta is None:
        theta = np.random.normal(0, .05, num_features)

    # We go through the entire training set a number of times
    # Each of these iterations is called an epoch
    for epoch in range(n_epochs):
        # Calculate the average gradient for all items and apply gradient ascent to theta
        # START ANSWER
        gradient = np.mean(calculate_gradients(theta, features, labels),axis=0)
        theta = apply_gradient(theta, gradient, alpha)
        # END ANSWER

    return theta

# Train a theta vector for the features and labels of the binary digits:
theta = train_theta(binary_digits_features_prime, binary_digits_labels, n_epochs=100000,

print("theta vector: " + str(theta))
print("log likelihood: " + str(log_likelihood(hypothesis(binary_digits_features_prime, t
```

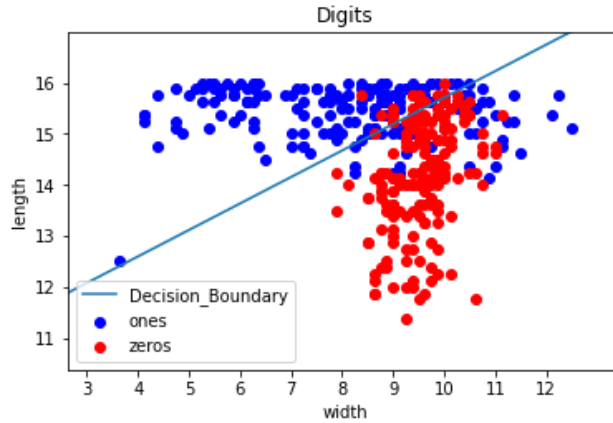
- Now we have obtained a trained theta vector, let's plot the decision boundary in the 2D scatterplot.

```
def decision_boundary(theta, plot_x):
    return (-1/theta[1]) * (theta[0] * plot_x + theta[2])

def plot_decision_boundary(theta, data, labels):
    db_x = np.array([data[:,0].min()-1, data[:,0].max()+1])
    db_y = decision_boundary(theta, db_x)
```

```
plot_scatter(data, labels, db_x = db_x, db_y = db_y)
```

```
plot_decision_boundary(theta, binary_digits_features_prime, binary_digits_labels)
```



Logistic regression Implementation on MNIST data

- Great! We now have implemented logistic regression to separate two classes based on 2 features. Let's extend this implementation to use a different representation of the object: the intensities of the 64 pixels!
- Initialize the data:

```
from sklearn.model_selection import train_test_split

# Flatten the data so all items are 1D and append an extra one feature to every item
binary_digits_pixels = add_one_features(binary_digits_images.reshape(binary_digits_image

# The shape should be (360, 65)
assert binary_digits_pixels.shape[0] == 360
assert binary_digits_pixels.shape[1] == 65

# Split dataset into train and test set
x_train_digits, x_test_digits, y_train_digits, y_test_digits = train_test_split(binary_d
```

- Train theta

```
# train a theta vector for the features and labels of the binary digits:
theta_digits = train_theta(x_train_digits, y_train_digits)
print("theta vector: " + str(theta_digits))
```

- Predict classes

```
def predict_binary(x_test, theta):
    """
    Predicts a label for each image in x_test using theta.
    :param x_test: an array of size (n, 65) of all test images.
    :param theta: a (65,) array of trained theta.
    :return: an integer array of size (n,) of labels for each test_image.
    """
    predictions = np.zeros(x_test.shape[0], dtype=int)
    # START ANSWER
    predictions = np.round(hypothesis(x_test, theta)).astype('int')
    # END ANSWER
    return predictions

x_test = np.array([[1,2,3,1], [-1,2,1.5,1], [4,-5,2,4]])
theta = np.array([1,-1,2,-2])
```



```

predictions = predict_binary(x_test, theta)
print(predictions)

assert (predictions == np.array([1, 0, 1])).all()
assert predictions.dtype == np.dtype('int')

```

- Accuracy function:

```

def compute_accuracy(predictions, y_true):
    """
    Computes the accuracy of the predictions based on the true labels.
    :param predictions: an array of size (n,) of the computed predictions for each image
    :param y_true: an array of size (n,) of the true labels of each image.
    :return: the accuracy of the predictions.
    """
    accuracy = -1
    # START ANSWER
    total = len(y_true)
    correct = 0

    for i in range(total):
        if (predictions[i] == y_true[i]): correct+= 1

    accuracy = correct/total
    # END ANSWER
    return accuracy

predictions = np.array([0,1,1,0,1])
y_true = np.array([0,1,0,1,1])

accuracy = compute_accuracy(predictions, y_true)
assert accuracy == 0.6

```

- Accuracy test with the data:

```

predictions = predict_binary(x_test_digits, theta_digits)
accuracy = compute_accuracy(predictions, y_test_digits)

print("accuracy: " + str(accuracy))
assert accuracy > 0.95

```

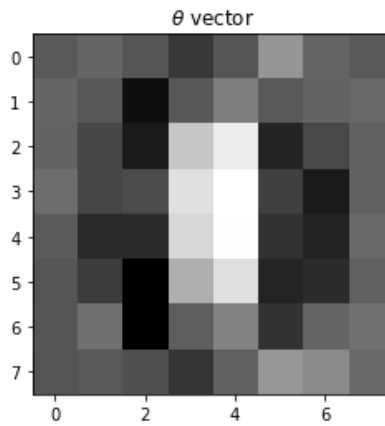
- With the pixel features we are able to obtain quite a good θ to create an accurate classifier. You can visualise the θ vector with the code below.

```

def plot_theta_image(theta, title=r"$\theta$ vector"):
    # remove bias from the image
    theta_no_bias = theta[:64].reshape(8,8)
    plt.figure()
    plt.gray()
    plt.title(title)
    plt.imshow(theta_no_bias)

plot_theta_image(theta_digits)
theta_digits

```



- We can see that the theta vector has high (brighter) values in the middle and dark in the edges
 - This because in the edges people don't often write
 - Most of the info regarding the shape of a number is contained in the middle of the image
- Learning curves

```
# Set Learning rate (try experimenting with this)
alpha = 0.001

# Set theta to initial value of None
theta_digits = None

# We go through the entire training set a number of times
# Each of these iterations is called an epoch
n_epochs = 50

accuracies_train = []
accuracies_test = []
log_likelihoods_train = []
log_likelihoods_test = []

for epoch in range(n_epochs):
    theta_digits = train_theta(x_train_digits, y_train_digits, n_epochs=1, theta=theta_d
    # Calculate accuracy
    accuracy_train = -1
    accuracy_test = -1
    # START ANSWER x_train_digits, x_test_digits, y_train_digits, y_test_digits
    predictions_train = predict_binary(x_train_digits, theta_digits)
    accuracy_train = compute_accuracy(predictions_train, y_train_digits)
    predictions_test = predict_binary(x_test_digits, theta_digits)
    accuracy_test = compute_accuracy(predictions_test, y_test_digits)
    # END ANSWER
    accuracies_train.append(accuracy_train)
    accuracies_test.append(accuracy_test)

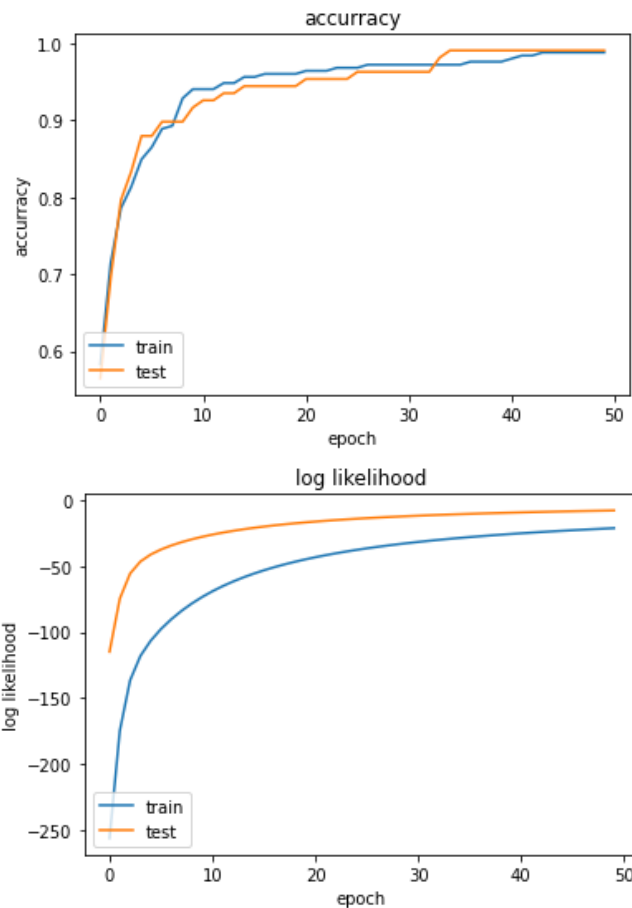
    # Calculate Log Likelihood
    ll_train = 0
    ll_test = 0
    # START ANSWER
    ll_train = log_likelihood(hypothesis(x_train_digits, theta_digits), y_train_digits)
    ll_test = log_likelihood(hypothesis(x_test_digits, theta_digits), y_test_digits)

    # END ANSWER
    log_likelihoods_train.append(ll_train)
    log_likelihoods_test.append(ll_test)

plt.plot(np.arange(len(accuracies_train)), accuracies_train, label='train')
plt.plot(np.arange(len(accuracies_test)), accuracies_test, label='test')
plt.title('accuracy')
```

```
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.legend(loc=3)
plt.show()

plt.plot(np.arange(len(log_likelihoods_train)), log_likelihoods_train, label='train')
plt.plot(np.arange(len(log_likelihoods_test)), log_likelihoods_test, label='test')
plt.title('log likelihood')
plt.xlabel('epoch')
plt.ylabel('log likelihood')
plt.legend(loc=3)
plt.show()
```



Multi-class logistic classifier

- Next, we will extend your code for binary classification to a multiclass classification for all 10 classes. First load the data with the code below

```
# Import the load function for the dataset
from sklearn import datasets
from sklearn.model_selection import train_test_split

n_classes = 10

# Load the digits with 10 classes (digits 0 - 9)
all_digits = datasets.load_digits(n_class=n_classes)
all_digits_images = all_digits.images
all_digits_labels = all_digits.target

# Flatten the data so they are 1D and append extra ones to the feature vectors
all_digits_pixels = add_one_features(all_digits_images.reshape(all_digits_images.shape[0]

# The shape should be (1797, 65)
assert all_digits_pixels.shape[0] == 1797
```

```

assert all_digits_pixels.shape[1] == 65

# Split dataset into train and test set
x_train_digits, x_test_digits, y_train_digits, y_test_digits = train_test_split(all_digi

```

- The strategy we will be using to construct a multiclass classifier from a binary classifier is known as “one-vs-all”, or “one-vs-rest”. The idea is to construct separate binary classifiers for each class that discriminate between that class and all other classes. We then combine the predictions of the resulting ten classifiers.
- We will create ten hypothesis functions (ten different theta’s θ_i): one for each class. Each hypothesis tells us the probability that a given image belongs to the corresponding class of θ_i . If the probability is low, it must belong to some other class.
 - We pretend for each class to be $y=1$ and the rest $y=0$

```

# Initialize a theta array, one for every class
multiclass_thetas = np.zeros((10,65))

for class_no in range(n_classes):
    current_label = class_no
    # Hint: convert the labels array to have only 1's at the current class_no
    multiclass_thetas[class_no] = train_theta(x_train_digits, np.where(y_train_digits ==
    # START ANSWER

    # END ANSWER
    print("class_no: " + str(class_no))

print("first 3 parameters of every theta")
print(multiclass_thetas[:, :3])

```

Responsible Machine Learning

- Bias is fast decision making without explicit thought
 - Formally “preference or inclination for or against something”
 - Can be positive, negative or neutral
 - Often accompanied by a refusal to consider the merits of other points of view
- Useful in real life
- Can cause problems in software
- Subconscious decision making of the programmer have an influence on developed software and machine learning
 - We will never be able to develop a system that is bias-free
 - But we can do things to identify them and to reduce them
- Bias can develop into prejudice:
 - Assumptions made without adequate knowledge
 - Most commonly used to refer to a preconceived judgement toward a person or a group of people because of a personal or specific characteristic
 - Usually resistant to rational influence
- Prejudice leads to discrimination:
 - Taking actions based on a prejudice
 - Treating a person or group of persons based solely on their membership of a certain group or category
 - The behaviour of excluding or restricting members of a group from opportunities that are available to people from another group
- Two types of bias
 - Implicit:

- Expectations based on learned coincidences which unknowingly affect everyday perceptions, judgment, memory and behaviour
- This is like bayes conditional probability
- Subconscious thought
- Explicit:
 - Is informed by our implicit bias but it also at least in part a conscious choice

Source of bias in Machine learning

- AI systems or ML techniques are not inherently bad nor turn bad by themselves
- Resolving data bias in ML projects means first determining where it is
- Important source for bias: training dataset
- The training database should represent the real world
- Skewed data is okay as long as you are aware of it (it may serve the purpose of analyzing that specific type of instances)
- The bad data comes from human biases:
 - Association bias: ML model reinforces and/or multiplies a cultural bias, which can create gender bias
 - Selection bias: dataset does not reflect realities of the environment in which a model will run
 - Racial bias: data skews in favor of particular demographics
- Lack of diversity in ML teams (think of facial recognition at amazon that didn't work with black people)

Cultural bias in language

- Word embeddings:
 - Tool to extract (semantic) associations between words/concepts
 - Each word is a vector in a vector space of 300 dimensions
 - Computed on the context it keeps in large text corpora
 - Influenced by culture (and gender)

Algorithmic bias

- Algorithm doesn't understand semantics and collects irrelevant information that corrupts the model

Evil programmers

- No ML/AI programming is self-learning
- All programs are implemented by people

Detecting bias

- There's no silver bullet solution
- Biases in ML remain a black box most of the time
- Bias is hard to quantify

Fairness

- Fairness is the bias or discrimination on specific realms
- Can be quantified in 3 ways (non of them are concrete):
 - It can be measured in stages (training data vs learned model)
 - It can be compared by demographic groups
 - they should be treated equally
 - We are all equal vs what you see is what you get
 - First one assume everybody is equally capable and should given the same treatment, the other one disputes it

Debiasing training set/model

- Check the distribution of class labels in training
 - Equalize the distribution training set (or make sure that it is representative of the population in which the model will be run on)

Debiasing group vs Individual level

- Similar outcomes for different groups
 - Smart algorithms
 - Carefully selecting the features used for the ML task e.g. zip codes are well-known proxies for race, are often eliminated
- Similar outcomes for similar individuals: much harder
 - 2 individuals on either side of the line are very similar but different outcome

WAE vs WYSIWYG

- Fix bias through smart implementation of the algorithm
 - Be aware of the bias
 - test the system
 - Choose the right features
 - make sure that they don't correlate with discriminating features
 - Work in a diverse team

Bias in IT decision making systems

- Decision making systems (i.e. getting a loan) don't leave room for human interpretation and are not sympathizing with the situation as a whole
 - Street-level bureaucracy: you can bend the rules
 - qualitative evaluation
 - System-level bureaucracy: you can't bend the rules
 - quantitative evaluation
 - programmers control the system
 - independent judge reviews individual cases only when there is an appeal
- Harmless bias in one domain (i.e. sport scouting) can be harmful in other domains (insurance fraud)
- ML does not scale well across domains for which it has not been specifically trained

Ethics

- Moral standards of "right" or "wrong" that prescribe how we must act
- There are two kinds:
 - Descriptive ethics: factual explanations about the moral systems we abide to (discussed by biology and psychology)
 - Normative ethics: prescriptive rules on how to behave (discussed by philosophy).
Examples:
 - Consequentialism (base your behaviour based on the consequences). It's utilitarian:
 - Actions must optimize "utility" (vague measure of "goodness" in terms of pleasure, happiness, well-being etc)
 - Leads to quantification and justification of difficult trade-offs between alternatives.
 - System-bureaucracy is like this.
 - Duty ethics (behaviour based on a backlog of tasks)
 - Rules for right action regardless of outcomes
 - The tasks are right/wrong because of their nature, regardless of the context
 - Satisfies the need for universal principles that won't admit exceptions
 - It's the opposite of consequentialism/utilitarianism
 - Virtue ethics (stoic virtue)

- Focuses on developing a virtuous individual, in the context of his own life, who is capable of deciding for himself
- It's a lifetime learning experience
- It's the opposite of both utilitarianism/consequentialism and duty ethics in terms of adopting a set of rules (utility score vs dogma activities) vs using common sense (developed as an individual through life experiences and rational judgement)...

Moral foundations theory

- There are 5 balances that determine moral virtue
 - Care vs harm
 - Fairness vs cheating
 - Loyalty vs betrayal
 - Authority vs subversion (legit leadership vs rebellious (without cause))
 - Sanctity vs degradation (religious virtue vs profane behaviour)

The proxy problem

- Often the source of a bias is impossible to determine
- Implicit bias has no clear explanation that can be targeted for mitigation techniques
- Deep neural networks are so blackboxed that it's impossible to mitigate its implicit biases
- proxy attributes are seemingly innocuous attributes that correlate with socially sensitive attributes, serving as proxies for the socially sensitive attributes themselves
 - I.e. kanalenland postcode correlates with muslim
- There is often a trade-off that the best available indicators for some classification task also serve as proxies for discriminative outcomes

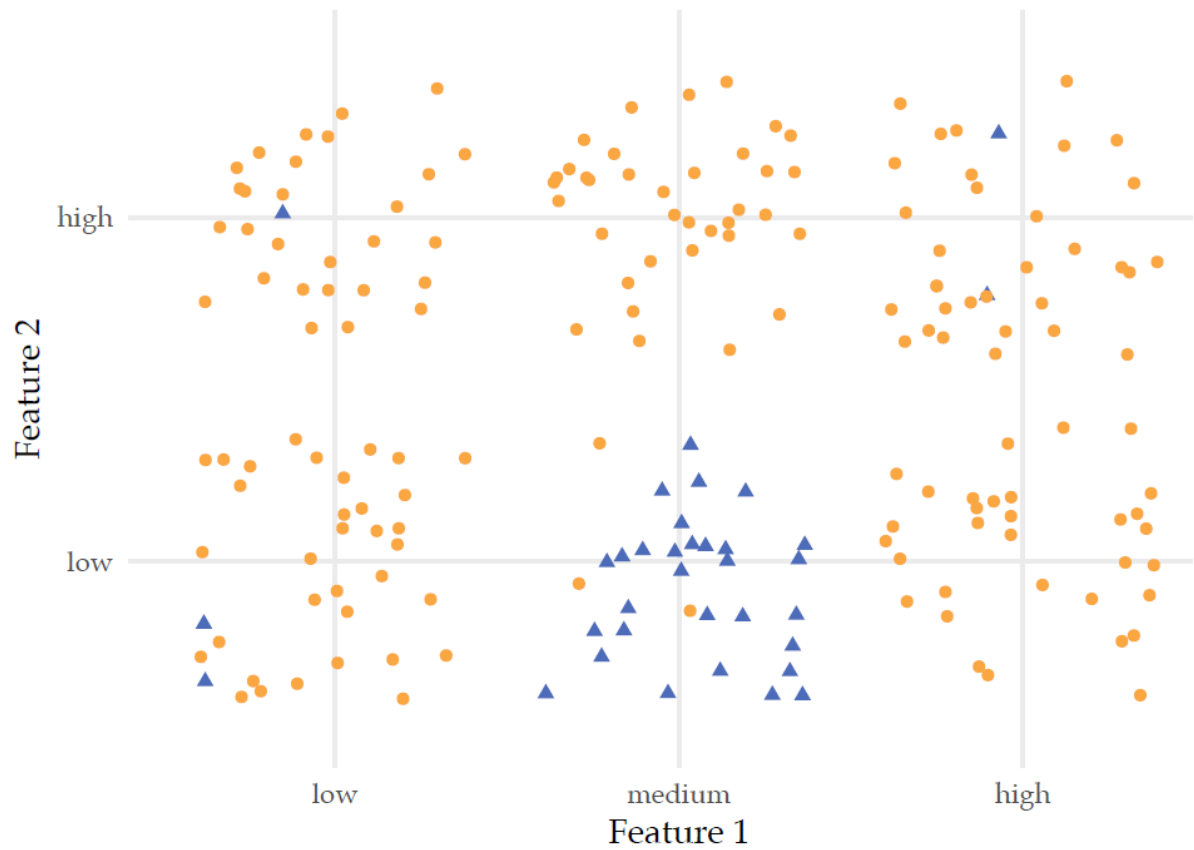
Concluding remarks

- Justifying the application of ML in the real world is a complex and often morally fraught process that should involve various stakeholders, adequate testing procedures, policy making, specific engineering expertise, variety of user involvement and a variety of expertise of the domain of application
- ML is never a shortcut solution for complex problems, they have to be integrated into complex practices and institutions with preexisting problems of justice and power

Non-linear classifiers

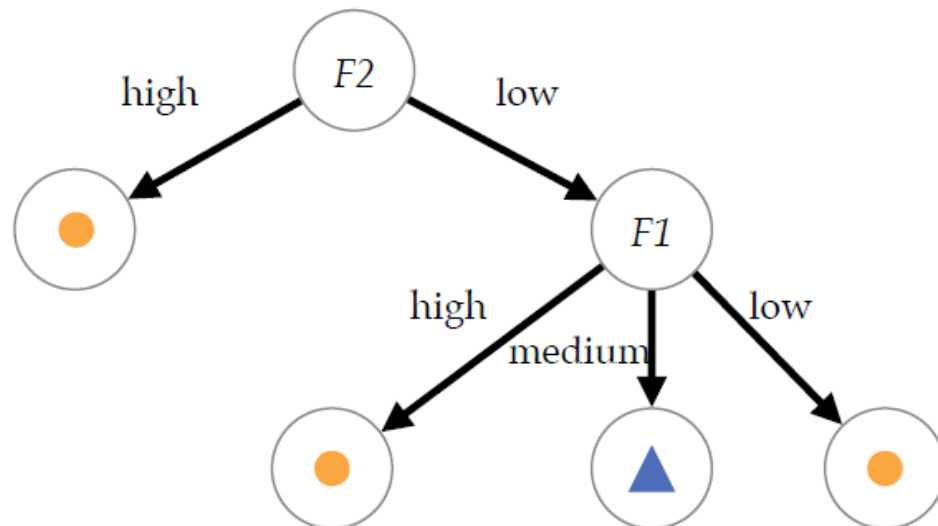
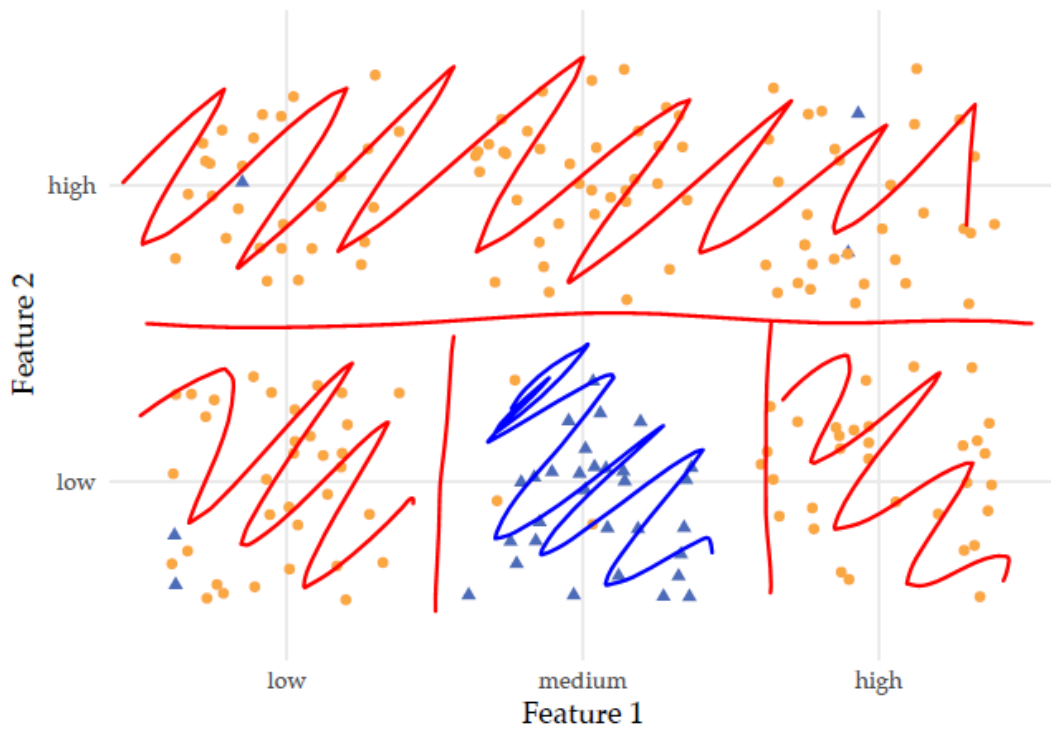
Decision trees

- Especially in high dimensional spaces linear classifiers might not work so well.



Defining the class of possible (hypothesis) functions

- Split the feature space using one feature at a time, recursively
 - Splitting forms a tree structure
 - Partitions the space in "rectangles"
 - In each rectangle/leaf, we assign a value



Defining the cost function (to measure the quality of the hypotheses)

$$h(\mathbf{x}) = \sum_{l \in \text{Leaves}} c_l \mathbb{I}(\mathbf{x} \in l)$$

$$\min_{c_l} \sum_{i=1}^N \mathbb{I}(h(\mathbf{x}_i) \neq y_i)$$

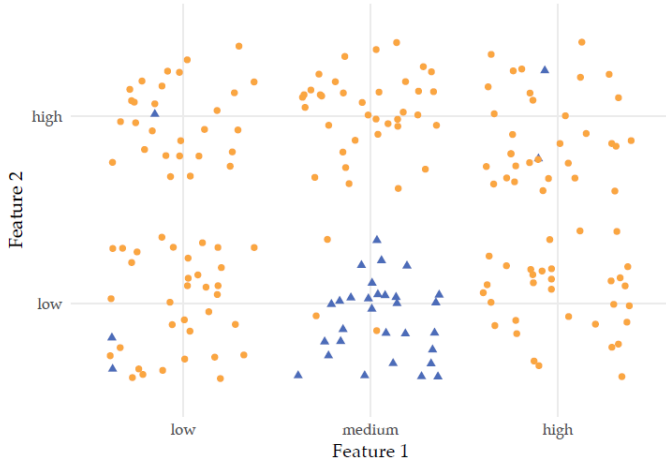
- Just count the number of mistakes...

- Technically means go through the function, measure the final leave node, and compare it with the actual value

Measuring/learning the average cost on the training data

- The most optimal solution is that one such that for a given tree node, the classification is equal the the most common class:

Consider the following "tree". What should the assignment in the node?



Orange

- If all objects in a node belong to one class: we are done
- If not, find a node-variable combination that increases the quality of the tree the most if we split it the node further

Measure of improvement

Set of objects in the node

$$I(S) - \sum_{V \in Values(F)} \frac{|S_V|}{|S|} I(S_V)$$

If $I(.)$ is the misclassification error in a set, this measures how much the 0-1 loss will decrease if we split the node into multiple nodes.

Other ways to measure information within a node $I(.)$?

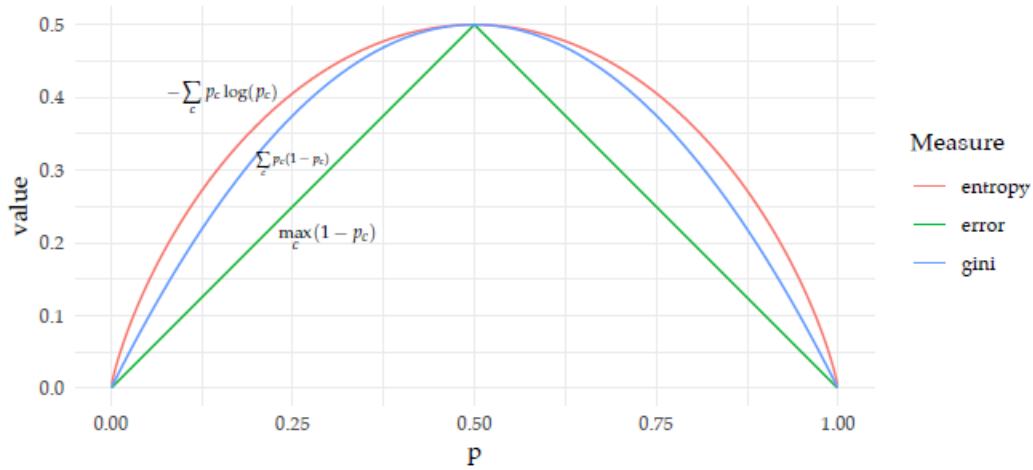
- The left term is the current misclassification error, the right term is the new misclassification error after applying the extra node-variable combination classification
 - There are multiple measurements for the misclassification error:
 - Missclassification (the normal one): $\max_c(1 - p_c)$
 - Entropy: $-\sum_c p_c \log(p_c)$
 - Gini Index: $-\sum_c p_c(1 - p_c)$

Splitting Criteria: Information Measures



p_{\bullet}	1.0	0.75	0.5	0.25	0.0
$\max_c(1 - p_c)$	0.0	0.25	0.5	0.25	0.0
$-\sum_c p_c \log(p_c)$	0.0 (!)	0.56	0.69	0.56	0.0 (!)
$\sum_c p_c(1 - p_c)$	0.0	0.375	0.5	0.375	0.0

Comparing Splitting Criteria



Note: Entropy scaled to have the same maximum

Example: Information Gain Calculation

$$I(S) = -\sum_c p_c \log(p_c) \quad I(S) = \sum_{i \in L} \frac{|S_i|}{|S|} I(S_i)$$

Node Entropy

Information Gain

Overall:
35/200

$$-\left(\frac{35}{200} \log\left(\frac{35}{200}\right) + \frac{165}{200} \log\left(\frac{165}{200}\right)\right) = 0.464$$

Feature 1:

High (3/99)

$$-\left(\frac{3}{99} \log\left(\frac{3}{99}\right) + \frac{93}{99} \log\left(\frac{93}{99}\right)\right) = 0.136$$

Low (32/101)

$$-\left(\frac{32}{101} \log\left(\frac{32}{101}\right) + \frac{69}{101} \log\left(\frac{69}{101}\right)\right) = 0.624$$

$$0.464 - \left(\frac{99}{200} \cdot 0.136 + \frac{101}{200} \cdot 0.624\right) = 0.081$$

Feature 2:

Low (3/65)

$$-\left(\frac{3}{65} \log\left(\frac{3}{65}\right) + \frac{62}{65} \log\left(\frac{62}{65}\right)\right) = 0.187$$

Medium (30/69)

$$-\left(\frac{30}{69} \log\left(\frac{30}{69}\right) + \frac{39}{69} \log\left(\frac{39}{69}\right)\right) = 0.685$$

High (2/66)

$$-\left(\frac{2}{66} \log\left(\frac{2}{66}\right) + \frac{64}{66} \log\left(\frac{64}{66}\right)\right) = 0.136$$

$$0.464 - \left(\frac{65}{200} \cdot 0.187 + \frac{69}{200} \cdot 0.685 + \frac{66}{200} \cdot 0.136\right) = 0.122$$

$$\text{Gain Ratio}(S) = \frac{IG(S)}{IV(S)}$$

Splitting continuous variables

- We set an arbitrary threshold
 - And based on this one assign to either side
- The number of thresholds is determined by measuring the information gain and choosing the amount that provides the highest information gain
- To tell when to stop growing a tree we can:
 - Put a minimum number of objects per node (partition) required
 - Put a maximum tree depth
 - Set a minimum information gain
- The alternative is to start with a given tree and "prune it"
 - Remove branches parts based on performance on validation set
- A *too* large tree is probably overfitted to the training data

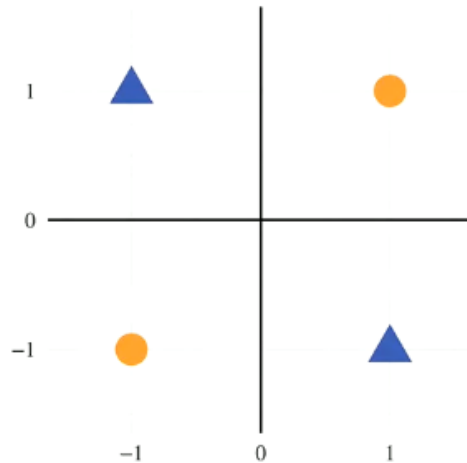
Overview

- What it is:
 - Non-linear classifier
 - Choices in learning:
 - Splitting criterion (information gain, gain ratio, misclassification error)
 - Stopping (minimum gain, node size, max depth)
 - Pruning
 - Condorcet's jury theorem:
 - When the individual jury has a probability of getting it right above 50%, then the more the better as the law of large numbers will ensure that the number of good decisions is larger than the bad ones, such that choosing the max class will be the right choice.
 - If it's below 50%, then the less the better as we might be lucky
 - Fixed rules:
 - Hard rules i.e. majority voting
 - Soft rules i.e. mean
 - Learned rules
 - Learn a classifier to output a decision based on the training set
- Good features:
 - Interpretable
 - Automatic feature selection
 - Easy to incorporate discrete features and missing values
 - Fast
- Bad features:
 - Unstable
 - Cannot model linear relationships efficiently
 - Greedy partitioning approach to non-linear classification that leads to an intuitive classifier, but typically has high variance

Neural net-works (Multi-layer perceptrons)

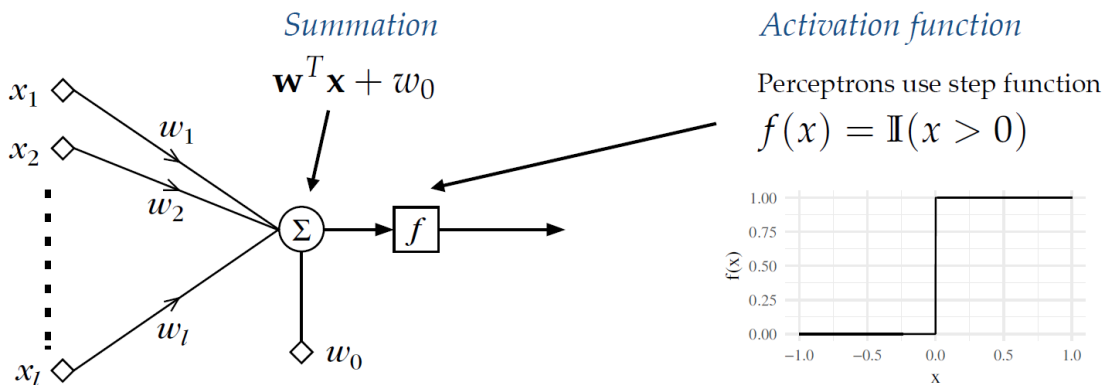
- There was a caveat with the information gain algorithm
 - If at a certain node, splitting the objects does not lead to information gain (like in the example above), the algorithm halts, although we can clearly see that we could have achieved a 100% classification rate if we had split the data with two lines! (namely the axes).
 - This problem happens because the logic tree approach first describes the possible classification functions, and then assigns them to the available nodes in an optimal way (which is halted in the first step)

- The solution to this particular problem is to transform the features into a new feature space where we have the product of y and x such that we can make a linear classifier into the new subspace (that splits positive against negative values), however, when its boundaries are displayed back to the original feature space note that the classifier is not longer a straight line



- Neural networks use a combination of classifiers in a "dynamic way"
- It is the common machine learning algorithm used today
- Relative of multi-layer perceptrons, artificial neural networks and feed-forward "deep" neural networks and connected to logistic regression, gradient descent and classifier combining
- A perceptron is an algorithm classifier that teaches itself without human training
 - The name originates from the Navy computer called Perceptron that did that

Perceptron



Inspired by a simplified model of neurons in the brain

- We take the inner product of the object features and the weights vector and then apply an "activation" function
 - It classifies the input to a binary solution (part of class or not)
 - Same as a linear classifier but uses a step function instead of a logistic regression

Perceptron loss function

Perceptron Math

$$\sum_{i=1}^N \max(-y_i(\mathbf{w}^T \mathbf{x}_i + w_0), 0)$$

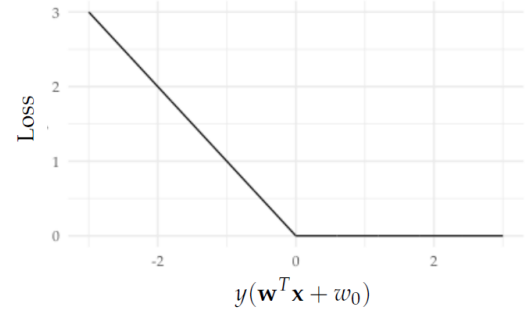
$y_i \in \{-1, +1\}$

↓

Loss: perceptron loss

Function class: linear

Optimizer: (stochastic) gradient descent



Perceptron training

- We train the perceptron with the gradient descent (like in the linear classifier logistic regression)

Perceptron Training

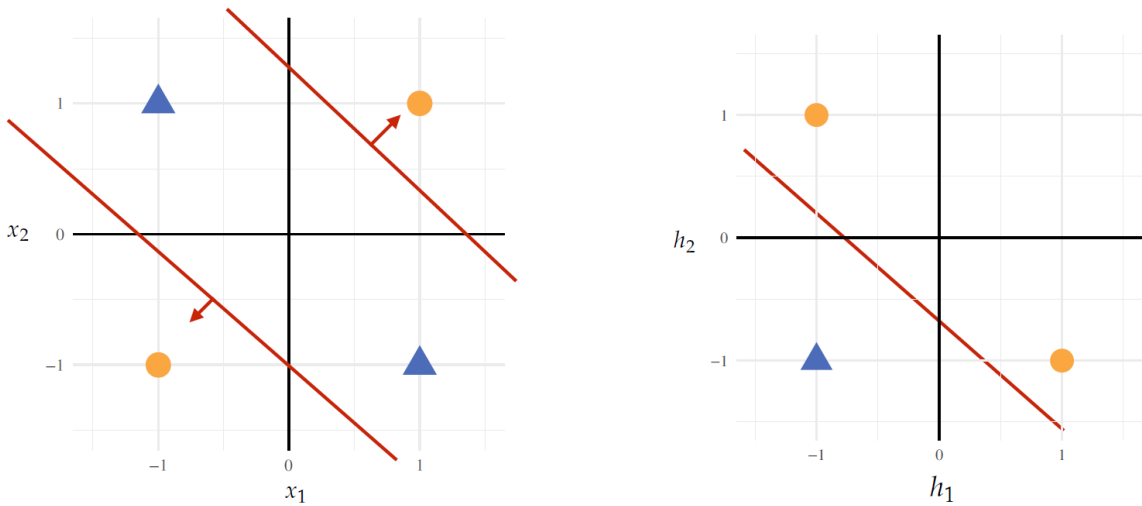
$$\sum_{i=1}^N \max(-y_i(\mathbf{w}^T \mathbf{x}_i + w_0), 0)$$

$$\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}} = \sum_{y_i(\mathbf{w}^T \mathbf{x}_i) < 0} -y_i \mathbf{x}_i$$

$$\mathbf{w}^{t+1} = \mathbf{w}^t + \alpha_t \sum_{y_i(\mathbf{w}^t \mathbf{x}_i) < 0} y_i \mathbf{x}_i$$

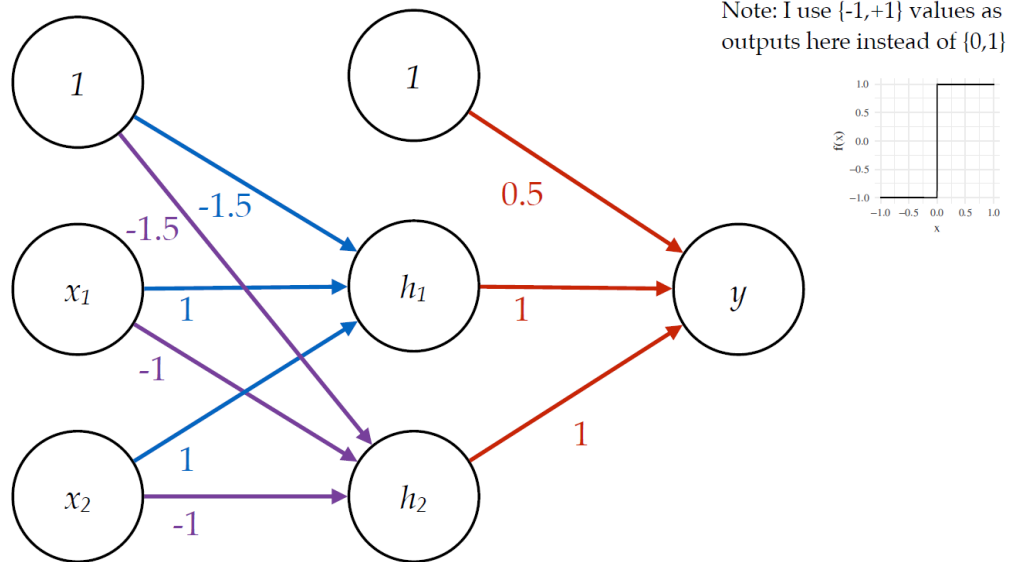
Guaranteed to converge if:

1. Problem is linearly separable
 2. We choose the right, decreasing, linear rate
- The perceptron itself remains a linear classifier and remains incapable of solving the XOR logic gate problem
 - But the beauty of the perceptron is that it is used in combination with other perceptron such that their outputs can be the inputs of others



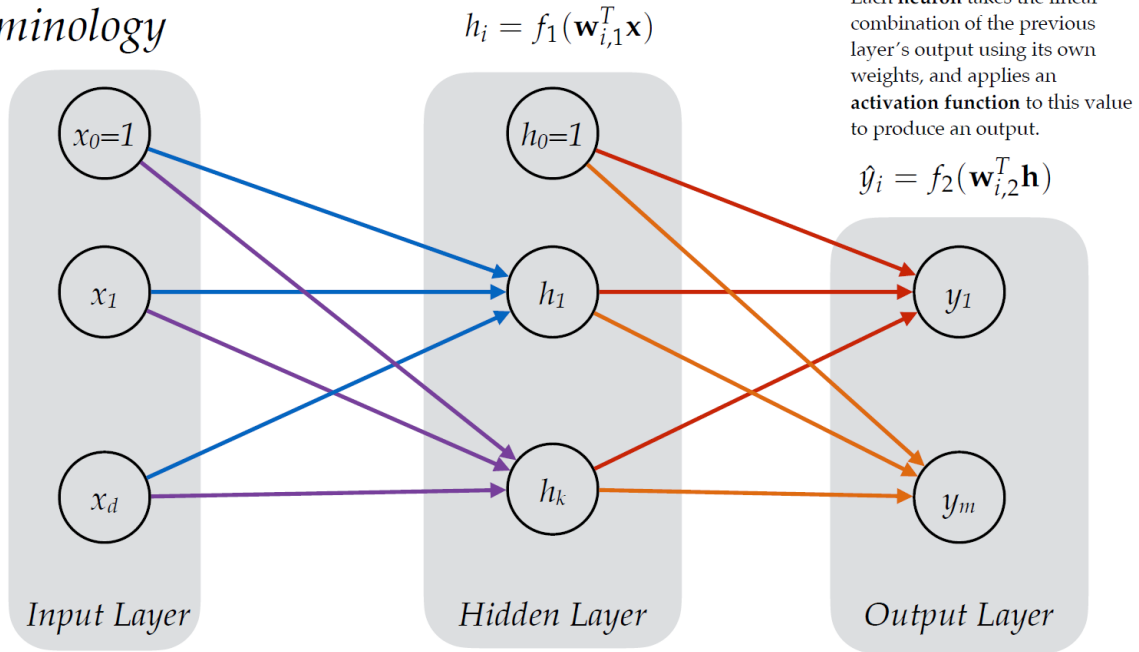
Use two classifiers and consider their outputs as the inputs to a third classifier

- Doing so creates a network of classifiers outputs linked to classifier inputs



- See that the first node is the bias and that the lines represent the weights of the weight vector
 - At each step, these lines may not only have different weights, but also may be forwarded differently to nodes (formally each of the nodes points to all of the nodes of the next layer, but if the weight is 0 you might remove it from the chart)
 - Then see how the last layer feeds back into a single node, namely the final output that classifies the object into a category
- The beauty of this approach is that we can combine linear classifiers (each representing a layer in the network) that can eventually classify problems that a priori were not solvable by a single linear classifier

Terminology



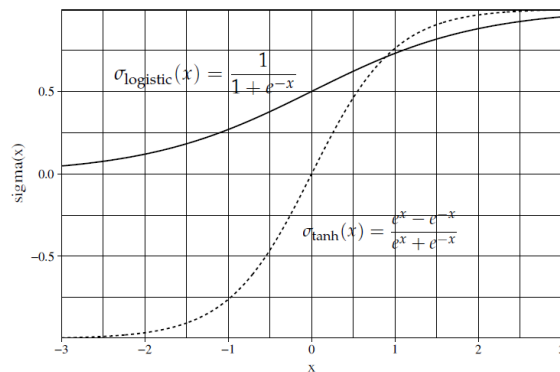
- It's called a feed-forward network because all the arrows point to the right (information flows only in one direction)
- See how we can implement the output layer in way that it can have a value for different classes (thus a multiclass classifier)
- The computations happen in the hidden layer and output layer
 - The fundamental difference between the output layer and the hidden layer is that in the output layer you can compare the results with the actual labels of the training set, however, with the hidden layer we are not able to tell whether the hidden layer vector has the right values (as they can't be compared to any previously collected data about the object)
 - You can also add as many hidden layers as you want/need

Activation function

- To have a more useful cost function, one that allows you to use the gradient descent more efficiently, we need to move from the step function (which does not show any (mis)improvements other than when operating near the boundary) to other activation function (such as the logistic one, where we can see (mis)improvements from tweakments everywhere)

Multi-layer Perceptron

- Confusing naming: MLP as a general term
- Other activation functions:
 - Sigmoid (logistic)
 - Tanh
 - Modern alternatives: ReLU and its variants



Note: tanh is a rescaled logistic

- Nowadays “multi-layer perceptron” stands for combination of layers (which may have different activation functions each) rather than the original one from the 60’s where it exclusively regarded the step up function

Learning is hard

- Since the object input has its weights linked to the hidden layers, the effects of changing the weights in the first layer do not necessarily have direct impact on the final output, which makes the gradient descent algorithm more clumsy but the implementation remains mostly the same
 - However the gradient descent can only fix the weights at the input layer, not the hidden. This is called “propagation of error”
- Backpropagation suggests us to start tweaking the highest weights first, as they are most likely contributing to the error the most, and then move onto the smaller ones

Challenges, advantages and disadvantages

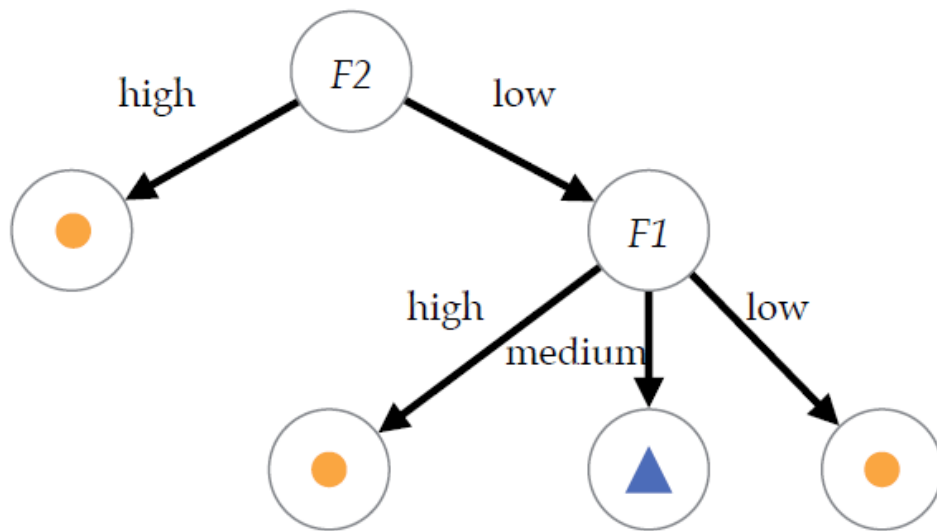
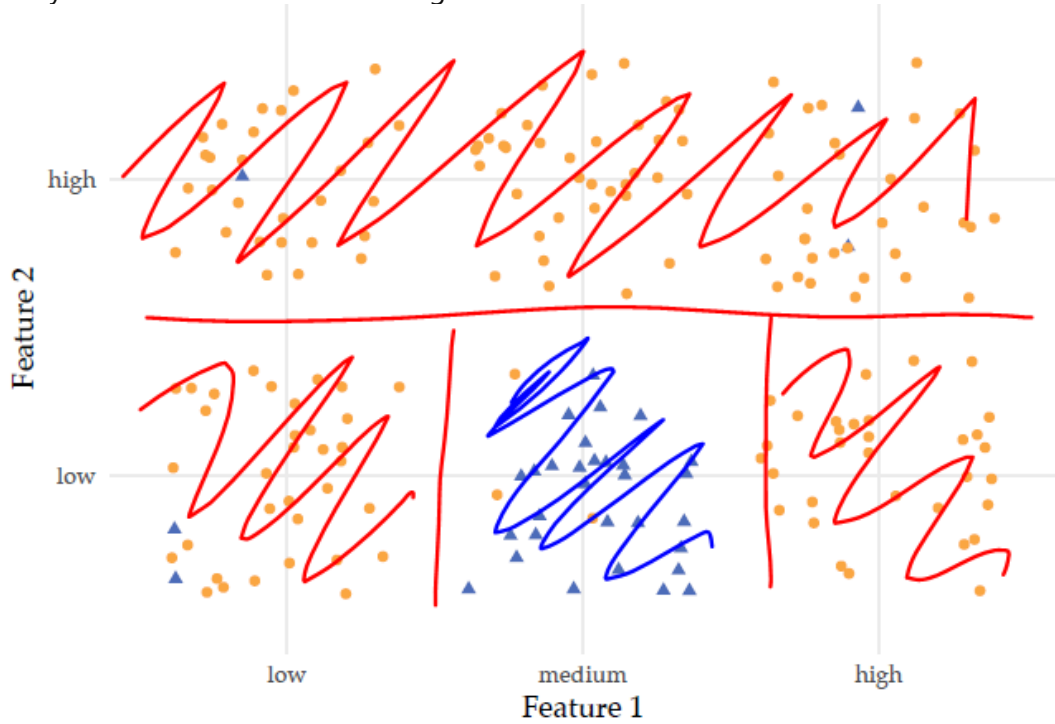
- Challenges:
 - Non-convex risk function may get stuck in local optima, convergence might be slow
 - Many architecture choices, hard to choose
 - Flexible model: risk of overfitting
 - Difficult to interpret the black box part of the model
 - With large parameter size it becomes computationally demanding
- Advantages
 - Flexible model class
 - Good empirical performance on many structured problems
 - Can be easily adapted to different learning settings
- Disadvantages
 - Computationally expensive
 - Lots of hyperparameters
 - Does not converge to a unique optimum
 - Optimizing is hard
 - Hard to interpret
- Recap
 - Perceptrons are simple linear binary classifiers
 - Multi-layer perceptrons are connected architectures of perceptron-like nodes, inspired by a simplified model of the brain

- They are trained using stochastic gradient descent, efficiently calculating the gradient using back propagation

Lab: non-linear classifier

Decisions trees

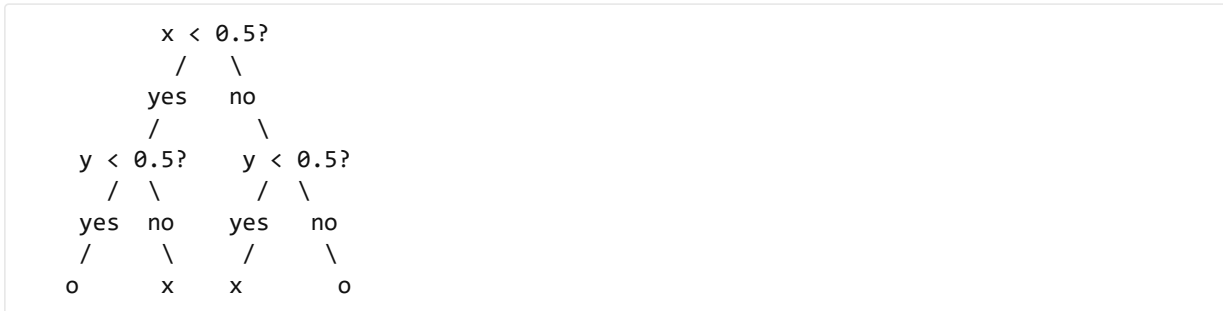
- Decision trees are non-linear classifiers. In other words: we can separate data with a decision boundary that does not resemble a single line.



- Think of the XOR problem given by the following points:
 - X = (0,1), (1,0)
 - O = (0,0),(1,1)
- Which resembles

o	x
x	o

- A priori there's no single line that can split the two classes
- But we can implement a series of true/false statements to classify the class of the objects in question:



Dataset: Heart Disease

- We will use decision trees to predict whether a patient has a heart disease using a dataset containing symptoms, prescriptions, and diagnoses from four different hospitals.

Relevant Information:

This database contains 76 attributes, but all published experiments refer to using a subset of 14 of them. In particular, the Cleveland database is the only one that has been used by ML researchers to this date. The "goal" field refers to the presence of heart disease in the patient. It is integer valued from 0 (no presence) to 4. Experiments with the Cleveland database have concentrated on simply attempting to distinguish presence (values 1,2,3,4) from absence (value 0).

Attribute Information:

```

-- Only 14 used
-- 1. #3  (age)
-- 2. #4  (sex)
-- 3. #9  (cp)
-- 4. #10 (trestbps)
-- 5. #12 (chol)
-- 6. #16 (fbs)
-- 7. #19 (restecg)
-- 8. #32 (thalach)
-- 9. #38 (exang)
-- 10. #40 (oldpeak)
-- 11. #41 (slope)
-- 12. #44 (ca)
-- 13. #51 (thal)
-- 14. #58 (num)          (the predicted attribute)

```

- The dataset contains both discrete and continuous variables
 - An example of a discrete variable is attribute #9, chest pain type, with four different possible labels for the chest pain type.
 - An example of a continuous variable is attribute #12, serum cholesterol in mg/dl, with the concentration of cholesterol in mg/dl
- Decision trees are particularly good at handling both discrete and continuous variables, so they could be a good classifier for this dataset.

Cleaning up the data

- The heart disease directory contains four datasets from different hospitals. We have created a cleaned-up version of the dataset where patient records from all four hospitals are aggregated together. You can load it in using NumPy:

```
import numpy as np
```

```
data = np.load('data/heart_disease.npy')
data.shape
```

(299,14) (299 patient records with 14 features)

- We can easily find out which ones are discrete and which ones are continuous by counting the number of unique values for each of them:

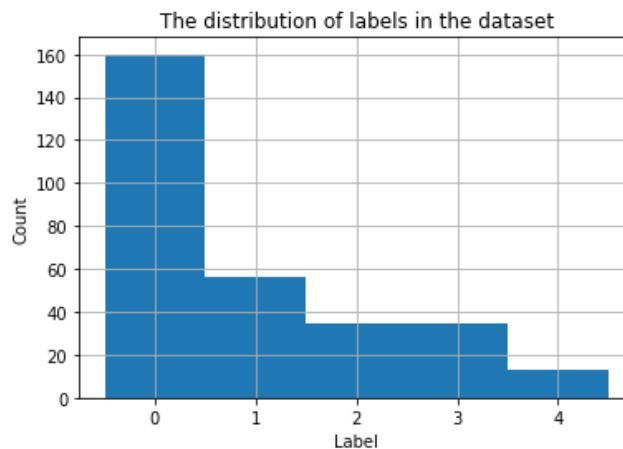
```
sorted_data = np.sort(data, axis=0)
frequencies = (sorted_data[1:,:] != sorted_data[:-1,:]).sum(axis=0) + 1
frequencies
```

array([41, 2, 4, 50, 153, 2, 3, 92, 2, 40, 3, 4, 3, 5])

- We can see that index 0,3,4,7,9 have significantly more unique values, and we're inclined to assume that these are the continuous variables.
- The last value is the diagnosis (outcome, output, y, object class label)
- We can visualize how these are distributed:

```
import matplotlib.pyplot as plt

plt.hist(data[:, 13], np.arange(0, 4 + 1.5) - 0.5)
plt.title('The distribution of labels in the dataset')
plt.ylabel('Count')
plt.xlabel('Label')
plt.grid()
plt.show()
```



- We will simplify the outcome such that it is either belonging to class absence (0) or to class presence (1,2,3,4).

Splitting the dataset

- In order to train and validate the decision trees, we split the dataset into a training and validation set and separate each of these into three arrays:
 1. `x_discrete`, a 2d-array of integers containing the discrete variables for each patient.
 2. `x_numeric`, a 2d-array of floats containing the numeric variables for each patient.
 3. `y`, a 1d-array of booleans indicating for each patient whether the diagnosis was class 0 or not. This array contains the labels.

```
# Separate the array into features and labels
x = data[:, :13]
y = data[:, 13]

# Transform classes to booleans
```

```

# y = (y == 0), Numpy will repeat this equality check for each entry in the array
# and return an array of booleans.
y = y == np.zeros(len(y))

def split_dataset(x, y, random_state):
    # Split data into train and validation
    from sklearn.model_selection import train_test_split

    # For this assignment, we state the random_state variable.
    # This variable will be used as the seed for the random number generation so that th
    # Therefore, all exercises will give the same results every run.
    x_train, x_validation, y_train, y_validation = train_test_split(x, y, test_size=0.3,

    # Separate features into discrete and numeric arrays.
    # You can verify that the split (with a boundary of 5) is correct by looking at the
    x_train_discrete = x_train[:, np.where(frequencies < 5)[0]].astype(int)
    x_train_numeric = x_train[:, np.where(frequencies > 5)[0]]
    x_validation_discrete = x_validation[:, np.where(frequencies < 5)[0]].astype(int)
    x_validation_numeric = x_validation[:, np.where(frequencies > 5)[0]]

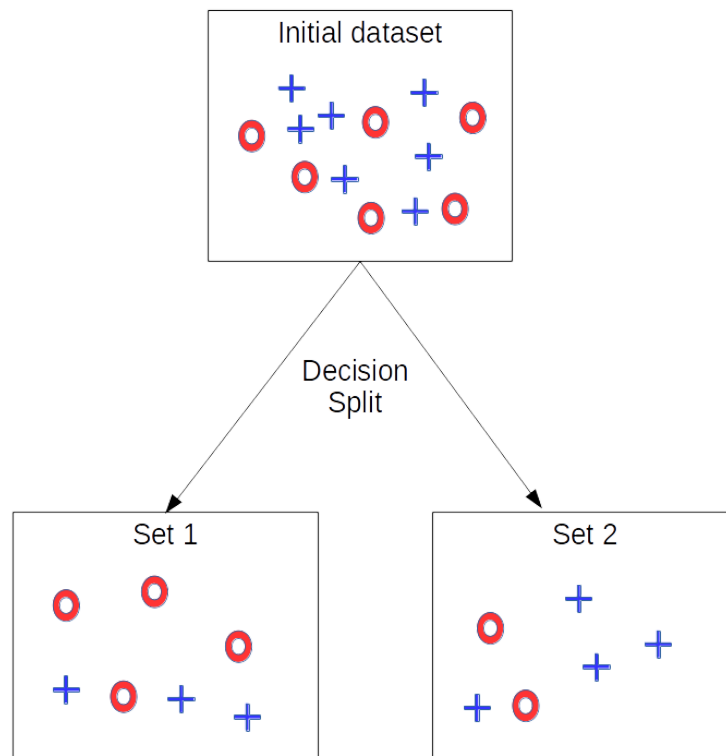
    return x_train_discrete, x_train_numeric, x_validation_discrete, x_validation_numeri

x_train_disc, x_train_num, x_validation_disc, x_validation_num, y_train, y_validation =

```

Entropy and information gain

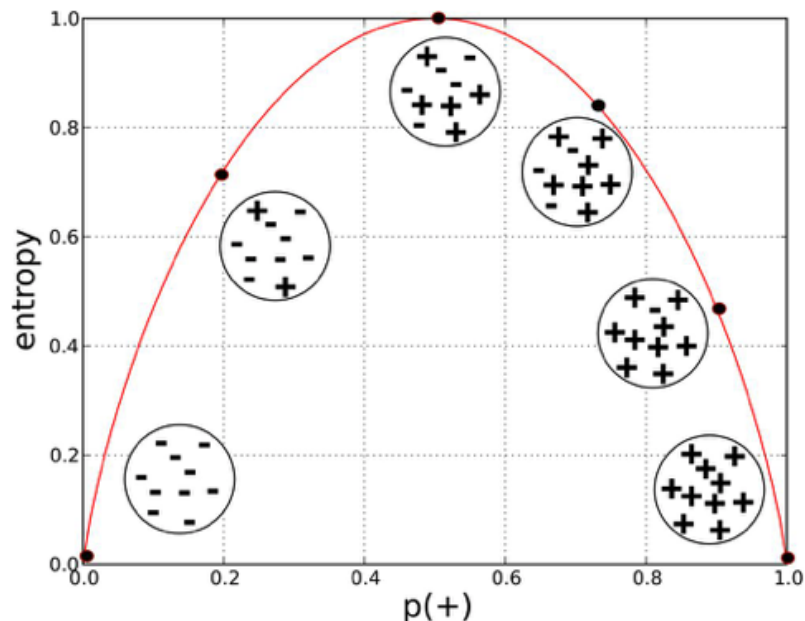
- A decision tree splits a dataset based on the values of certain features. To find the best features and values to split on, we need some way to measure the quality of a split. We will use information gain (which is based on the entropy in the nodes) for this purpose.



Entropy

- For decision trees we use the information theoretic entropy, also known as Shannon entropy. It tells us about the amount of information contained in a certain distribution of data.

- The best split is the split on a feature that separates most of the "1s" from the "0s" in the resulting two sets. Entropy can thus also be regarded as a measure of purity, and we aim to increase the purity of nodes.
- An entropy close to 1.0 indicates that a subset of the data contains an equal number of labels "1" and "0", and thus a split resulting in such subsets is not useful.
- In the graph below, the relation between entropy and the proportion of data points belonging to one class (in this case '+') in a data set is plotted.



- As can be seen in the image, the entropy is maximal when the set contains an equal number of "1" and "0" labels (At this point the uncertainty is the highest).
- The entropy decreases as the data set becomes 'purer'. Our goal is to decrease the entropy by making proper splits.
- The Shannon entropy for any number of classes is given as:

$$\circ \quad \phi(p) = - \sum_i p_i \log_2(p_i)$$

- As mentioned, we will only decide whether an entry belongs to class `0` or not: True or False. Thus, in our case, we can re-write Shannon entropy as follows:

$$\bullet \quad \phi(p) = -p \log_2(p) - (1 - p) \log_2(1 - p)$$

- where p , the probability that an item has label 0, is equivalent to the ratio between the number of items with label 0 (True) and the number of items with another label (False).
- First complete the `ratio()` function to compute `p`. The function, given a list of boolean values as class labels, should return the ratio of `True` labels in the list, e.g. `1.0` would indicate the list only contains `True` labels.

```
import math

def ratio(labels):
    if len(labels) == 0:
        return 0
    # START ANSWER
    return labels[np.where(labels == 1)].size/len(labels)
    # END ANSWER

#print('Ratio for train set:', ratio(y_train))
#print('Ratio for validation set:', ratio(y_validation))

# Verify the correctness of the ratio function
```

```
assert np.isclose(ratio(y_train), 0.53110)
assert np.isclose(ratio(y_validation), 0.54444)
```

- Next, we compute the entropy. Before we start writing the code, we deal with a possible source of error: the computation of $0 \log_2(0)$ (when p is 0) will correctly result in a math error. Then combine `ratio()` and `entropy_sub()` to compute the `entropy()` of a list of boolean class labels.

```
def entropy_sub(p):
    """
    Returns the value for p * log_2(p)
    """
    # START ANSWER
    if p == 0:
        return 0
    return p * math.log2(p)
    # END ANSWER

def entropy(labels):
    """
    Returns the entropy of an array of labels, computed using equation (3.1.b)
    """
    # START ANSWER
    p=ratio(labels)
    return -entropy_sub(p)-entropy_sub(1-p)
    # END ANSWER

print('Entropy for train set:', entropy(y_train))
print('Entropy for validation set:', entropy(y_validation))

# Verify the correctness of the entropy function
assert np.isclose(entropy(y_train), 0.9972)
assert np.isclose(entropy(y_validation), 0.9943)
```

- The entropies of s sets of labels can be combined using a weighted sum:

◦

$$I_m = \sum_{j=1}^s \frac{N_j}{N} \phi(p_j)$$

- This will give us the overall entropy T of a split on variable m , where N is the size of the set before the split, N_j is the size of the j^{th} set after the split, and $\phi(p_j)$ is the entropy of the j^{th} set.
- Complete the function `split_entropy()` to compute this value for a list of labels and `N`.
 - This is nothing more than a weighted average for the entropy for each partition (node) weighted against how large the partition is

```
def split_entropy(label_lists, N):
    information = 0 # I (m is assumed to be class 0 or 1??)
    # N is the size of the set before the split...

    for label_list in label_lists:
        # START ANSWER
        information += entropy(label_list) * label_list.size/N
        # END ANSWER
    return information

# Verify the correctness of the split_entropy function
labels = np.array([0, 0, 0, 0, 1, 1, 1, 1])
N = len(labels)
print('Entropy of the data before splitting:', entropy(labels))
```

```

# Worst case split
labels_list = np.array([[0, 0, 1, 1], [0, 0, 1, 1]])
print('Worst case:', split_entropy(labels_list, N))
assert np.isclose(split_entropy(labels_list, N), 1.0)

# Better split
labels_list = np.array([[0, 0, 0, 1], [1, 1, 1, 0]])
print('Better:', split_entropy(labels_list, N))
assert np.isclose(split_entropy(labels_list, N), 0.81128)

# Perfect split
labels_list = np.array([[0, 0, 0, 0], [1, 1, 1, 1]])
print('Optimal:', split_entropy(labels_list, N))
assert np.isclose(split_entropy(labels_list, N), 0.0)

```

Information gain

- Information Gain (IG) measures how much the entropy changes by making a specific split, i.e. the relative gain in predictability of the data by creating a specific distribution of labels
- IG is defined as the entropy of the original distribution $\phi(p)$ minus the entropy of the split distribution I_m resulting from the split on variable m .

$$IG_m = \phi(p) - I_m$$

- The IG thus depends on two things:
 - the current list of labels.
 - How these labels are divided into new distributions by the split.

```

def information_gain(labels, indices):
    labels_list = []
    for index_list in indices:
        labels_list.append(labels[index_list])
    # START ANSWER
    return entropy(labels) - split_entropy(labels_list, labels.size)
    # END ANSWER

labels = np.array([0, 0, 0, 0, 1, 1, 1, 1])
labels_list = np.array([[0, 0, 0, 1], [1, 1, 1, 0]])

# Now we create `indices` that correspond to the indices of the split (compare them with
# You will have to write code that creates such a list later.
indices = np.array([[0, 1, 2, 4], [5, 6, 7, 3]])

# Verify the correctness of the information_gain function
assert np.isclose(information_gain(labels, indices), 0.18872)

```

Creating decision trees

- With the data turned into a usable format and the functions to measure entropy and IG ready, we can start building the actual decision tree
- Classes are a great way to represent trees: each DecisionTree object class represents a node (or subtree) in the tree and we only have to store references to that node's children to build the structure of the tree. We will distinguish two types of 'nodes':
 - DiscreteTree nodes, split on the basis of the value a discrete variable
 - NumericTree nodes, split on the basis of the value of a numeric variable
- Because these two classes are very similar, we will use inheritance to avoid redundancy.
- We create a general DecisionTree node and then turn it into a DiscreteTree or NumericTree instance based on the best possible split


```

from collections import defaultdict

class DecisionTree(object):
    def __init__(self, data_discrete, data_numeric, labels, tree_type=0, thres=0.1):
        """ Creates a Decision Tree, based on the following arguments:
            data_discrete - A 2D array of ints, each row containing the discrete fea
            data_numeric - A 2D array of floats, each row containing the numeric fea
            labels - An array of boolean class labels, each corresponding to a
                    DataRow instance of a patient at the same index.
            tree_type - 0: create the Tree with the highest IG every node
                       1: create DiscreteTrees only
                       2: create NumericTrees only
            thres - The cutoff value for IG, to stop splitting the tree.
                   Below this value the node becomes a leaf node and no
                   further splits are made.

            N.B. This function has already been provided and does not need to be modifie
        # Store the basic attributes for any DecisionTree
        self.data_discrete = data_discrete
        self.data_numeric = data_numeric
        self.labels = labels
        self.tree_type = tree_type
        self.thres = thres

        # Compute the current ratio of labels and assign this node the most common label
        self.ratio = ratio(self.labels)
        # This will assign a boolean value to self.label, as `self.ratio >= 0.5` is a bo
        self.label = self.ratio >= 0.5

        if self.tree_type == 1:
            # Convert this DecisionTree to a DiscreteTree and perform the split
            discr_tree = DiscreteTree(self)
            self.convert_tree(discr_tree)
        elif self.tree_type == 2:
            # Convert this DecisionTree to a NumericTree and perform the split
            numer_tree = NumericTree(self)
            self.convert_tree(numer_tree)
        else:
            # If no specific type has been given (tree_type: 0), we determine which type
            # by computing both options and comparing the IG.
            # Create a DiscreteTree and NumericTree, passing all the stored attributes
            # as an argument, and compute the best possible split for each
            discr_tree = DiscreteTree(self)
            numer_tree = NumericTree(self)

            # Based on the results of the split computations, replace this generic
            # DecisionTree node with either a DiscreteTree or a NumericTree node
            if discr_tree.info_gain > numer_tree.info_gain:
                self.convert_tree(discr_tree)
            else:
                self.convert_tree(numer_tree)

        # Create an empty dictionary to contain the (possible) branches from this node,
        # where the values should be new DecisionTree nodes, or None if not present
        self.branches = defaultdict(lambda: None)

        # Check if this split produced a high enough IG to actually create
        # the resulting branches with new split nodes below it,
        # else no split is carried out and the original node is a leaf node
        self.leaf = self.info_gain < self.thres
        if not self.leaf:
            self.create_subtrees()

    def store_split_values(self, feat_index, feat_values, indices, info_gain):
        """ Stores the values of the passed parameters as object attributes. Is intended
            to store the results of a split computation for either a DiscreteTree or a
            NumericTree. The stored attributes are:

```

```

    feat_index - The index of the feature on which the split was
                 based.
    feat_values - A list of the possible values that this split feature can
                 take, each corresponding to a different branch in the DecisionTree
    indices - A list of index lists, with each list containing the indices
             defining a subset of the current data and label attributes, as
             computed by the split. The order of these subsets should match the
             order of the corresponding feat_values used to define the branches
             of the split.
    info_gain - IG computed for this split
    N.B. This function has already been provided and does not need to be modified
self.feat_index = feat_index
self.feat_values = feat_values
self.indices = indices
self.info_gain = info_gain

def convert_tree(self, new_tree):
    """ Converts this object to the tree passed as the new_tree parameter.
        All attributes from the new_tree are transferred.
        new_tree - Either a DiscreteTree or a NumericTree instance, to which
                  this object is converted
        N.B. This function has already been provided and does not need to be modified
self.__class__ = new_tree.__class__
self.__dict__ = new_tree.__dict__

def create_subtrees(self):
    """ Creates the different subsets of the current data and labels, and makes a
        a new DecisionTree node for each such subset, based on the indices attribute
        stored after the computed split. These new DecisionTrees are stored in the
        branches attribute, a dictionary mapping the value of a variable from the
        split to the new DecisionTree created by selecting that value for the split.
    for i, key in enumerate(self.feat_values):
        subset_discrete = self.data_discrete[self.indices[i]]
        subset_numeric = self.data_numeric[self.indices[i]]
        subset_labels = self.labels[self.indices[i]]
        subtree = DecisionTree(subset_discrete, subset_numeric, subset_labels, tree_
self.branches[key] = subtree

def classify(self, row_discrete, row_numeric):
    """ Traverses the DecisionTree based on the values stored in the given row and
        returns the most common label in the resulting leaf node.
        row - The index of the row being classified"""
    # Option 1: node is a leaf
    if self.leaf:
        return self.label

    subtree = self.get_subtree(row_discrete, row_numeric)

    # Option 2: no valid subtree
    if subtree is None:
        return self.label

    # Option 3: there is a valid subtree
    return subtree.classify(row_discrete, row_numeric)

def split(self):
    """ Must be implemented by the subclass based on the specific type of split performed
        The function here is only to ensure it is implemented, and should not be modified
    raise NotImplementedError

def get_subtree(self, instance):
    """ Must be implemented by the subclass based on the specific type of split performed
        The function here is only to ensure it is implemented, and should not be modified
    raise NotImplementedError

```

Discrete split

- This function should, for every discrete variable in the data, try to create a split based on that variable and compute the Information Gain of the resulting split.
- For discrete splits, we split the set into subsets: one for each discrete label. For example, if there are three possible labels for a certain feature, we should return three subsets.
- The question is: which feature should we pick to split on? You can find out by performing the following steps:
 - For each feature:
 - Split the set into subsets corresponding to each discrete label.
 - Compute the information gain for this split.
 - Split the dataset based on the feature with the highest information gain.
- Once the best feature for the split has been determined, the results of the split need to be stored in the instance, so they can be used to build the rest of the tree.
- Let's assume the algorithm decides to split on chest pain type (#9). The following attributes should then be stored when splitting
 - The index of the chest feature we split on
 - A list of discrete options for this feature (e.g. [0, 1, 2, 3], indicating the type of chest pain)
 - A list of indices per option, so the first sublist contains the indices of all rows with chest pain type = 0, etc. (e.g. [[0, 3, 4, 5, 7, 9, ...], [8, ...], [2, 6, ...], [1, ...]])
 - The information gain resulting from the split (e.g. 0.8)
- These attributes can then be used to build the rest of the tree.

```
def create_indices_list(column):
    """ Creates the indices list, containing for each possible value of the current feat
        the indices of corresponding rows (e.g. [[0, 2], [1, 3], ...] where the current
        feature is 0 in rows 0 and 2).
        Returns the list of indices for all feature values and as second output a list of
        column - The column of one feature from the data."""
    # START ANSWER
    options = np.unique(column)
    indices = [np.where(column == option)[0].tolist() for option in options]
    return indices, options
    # END ANSWER

vals = np.array([0, 0, 1, 0, 1, 1, 2, 2])
indices_list, feat_values = create_indices_list(vals)

# Verify the correctness of the create_indices_list function
assert ([[*i] for i in indices_list] == [[0, 1, 3], [2, 4, 5], [6, 7]])
assert np.array_equal(feat_values, [0, 1, 2])
```

```
class DiscreteTree(DecisionTree):
    def __init__(self, dtree):
        """ Takes a DecisionTree as initialization parameter and copies all its
            attributes. Then calls the split() function to determine the optimal
            discrete variable to split this subset of the data on.
            dtree - The DecisionTree instance whose attributes are copied to this
            DiscreteTree instance.
            N.B. This function has already been provided and does not need to be
            modified."""
        self.__dict__ = dtree.__dict__.copy()
        self.split()

    def split(self):
        """ Determines the best discrete variable to split the current dataset on,
            based on the IG resulting from the split. For this best split variable, the
            function stores several resulting attributes from the split, using the
            store_split_values function. See the documentation of store_split_values
            for an overview of what should be stored."""
        max_feat = None
```

```

max_feat_values = None
max_split = None
max_ig = 0

for feat in range(self.data_discrete.shape[1]):
    # 1. Call create_indices_list() for the feature column.
    # 2. Compute the IG of the split
    # 3. If IG > max IG, update max values

    # START ANSWER
    indices, feat_values = create_indices_list(self.data_discrete[:,feat])
    IG = information_gain(self.labels, indices)
    if IG > max_ig:
        max_feat = feat
        max_feat_values = feat_values
        max_split = indices
        max_ig = IG
    # END ANSWER

self.store_split_values(max_feat, max_feat_values, max_split, max_ig)

def get_subtree(self, row_discrete, row_numeric):
    """ Returns the subtree one branch down.
    Returns None if the value was not present at the split.
        row_discrete - array of the discrete values
        row_numeric - array of the numeric values"""
    value = row_discrete[self.feats_index]
    return self.branches.get(value, None)

```

Creating subtrees

- If we were to repeat this process of splitting each node, we end up with a tree structure
- But we need to stop at a certain moment to avoid building infinite trees and to avoid overfitting.
- There are quite a few strategies to decide when to stop. The simplest of these is just to stop splitting when the Information Gain of a split drops below a certain threshold.

Classifying patients

- With this structure built, classifying a new patient record is done by traversing the tree given a patient record.
- At each DecisionTree node, we have three options, based on the type of node:
 - The node is a leaf node, in which case the classification will be the most common label of that node
 - The node does not have a valid subtree for the splitted value, so the classification will also be the node label
 - The node has a subtree for the splitted value, in which case you can recursively continue classifying on the subtree

Validating patient records

- a validate() function which takes as input a trained decision tree, a validation set of patient records and corresponding labels, and returns the percentage that is classified correctly.

```

def validate(decision_tree, data_discrete, data_numeric, labels):
    """ Classifies all patient records and compares the outcome to
    the provided labels. Returns the percentage of elements that was classified
    correctly.
        data_discrete - A 2D array of ints, each row containing the discrete feature
        data_numeric - A 2D array of floats, each row containing the numeric feature
        labels - List of boolean labels each belonging to a patient record"""

```

```

# START ANSWER
result = 0
for i in range(len(data_discrete)):
    subtree = decision_tree.get_subtree(data_discrete[i], data_numeric[i])
    if subtree != None:
        if subtree.classify(data_discrete[i], data_numeric[i]) == labels[i]:
            result += 1
return result / len(data_discrete)
# END ANSWER

```

Optimizing splits

- The numeric split is based on a split boundary, where all values smaller than the boundary are placed in one branch, and those greater or equal in the other branch.
- Implementing the `find_best_split` function. This function tries every possible split boundary for every feature and uses the split with the best IG overall.

```

def find_best_split(data, labels):
    max_feat = None
    max_split = None
    max_ig = 0
    max_boundary = None

    for feat in range(data.shape[1]):
        col = data[:, feat]
        for curr_boundary in col:
            # START ANSWER
            indices = []
            indices.append(np.where(col < curr_boundary)[0].tolist())
            indices.append(np.where(col >= curr_boundary)[0].tolist())
            IG = information_gain(labels, indices)
            if IG > max_ig:
                max_feat = feat
                max_split = indices
                max_ig = IG
                max_boundary = curr_boundary
            # END ANSWER
    return max_feat, max_split, max_ig, max_boundary

max_feat, max_split, max_ig, max_boundary = find_best_split(x_train_num[:10,:], y_train)
print(max_split, max_feat, max_ig, max_boundary)

# Verify the correctness of the find_best_split function
assert ([[*i] for i in max_split] == [[1, 3, 5, 6, 7], [0, 2, 4, 8, 9]])
assert np.array_equal(feats_values, [0, 1, 2])
assert max_feat == 3
assert np.isclose(max_ig, 0.27807)
assert max_boundary == 159.0

```

```

class NumericTree(DecisionTree):
    def __init__(self, dtree):
        """ Takes a DecisionTree as initialization parameter and copies all its
            attributes. Then calls the split() function to determine the optimal
            numeric variable to split this subset of the data on.
            dtree - The DecisionTree instance whose attributes are copied to this
                    NumericTree instance.
            N.B. This function has already been provided and does not need to be modified
        """
        self.__dict__ = dtree.__dict__.copy()
        self.split()

    def split(self):
        """ Determines the best boundary for any numeric variable to split the

```

```

current dataset on, based on the IG resulting from the split. For this
best split boundary, the function stores several resulting attributes
from the split, using the store_split_values function. See the
documentation of store_split_values for an overview of what should
be stored. In addition, one more attribute is stored in the numeric
case, namely the boundary value used for the split."""
max_feat, max_split, max_ig, boundary = find_best_split(self.data_numeric, self.
self.boundary = boundary

max_feat_values = [False, True]
self.store_split_values(max_feat, max_feat_values, max_split, max_ig)

def get_subtree(self, row_discrete, row_numeric):
    """ Returns the subtree one branch down.
        row_discrete - array of the discrete values
        row_numeric - array of the numeric values"""
    value = row_numeric[self.feats_index] >= self.boundary
    return self.branches.get(value, None)

```

Threshold

- Different threshold (to decide when to stop splitting) when creating a DecisionTree yield different accuracies
- First let's define accuracy

```

def get_accuracy(threshold = 0.1, n_iterations = 50, standarization=False):
    hybrid_accuracy = 0
    discrete_accuracy = 0
    numeric_accuracy = 0

    for i in range(n_iterations):

        x_train_disc, x_train_num, x_validation_disc, x_validation_num, y_train, y_valid

        # START ANSWER
        hybrid_tree = DecisionTree(x_train_disc, x_train_num, y_train, tree_type = 0, th
        discrete_tree = DecisionTree(x_train_disc, x_train_num, y_train, tree_type = 1,
        numeric_tree = DecisionTree(x_train_disc, x_train_num, y_train, tree_type = 2, t
        hybrid_accuracy += validate(hybrid_tree, x_validation_disc, x_validation_num, y
        discrete_accuracy += validate(discrete_tree, x_validation_disc, x_validation_num
        numeric_accuracy += validate(numeric_tree, x_validation_disc, x_validation_num,
        # END ANSWER

        hybrid_accuracy /= n_iterations
        discrete_accuracy /= n_iterations
        numeric_accuracy /= n_iterations

    return hybrid_accuracy, discrete_accuracy, numeric_accuracy

hybrid, discrete, numeric = get_accuracy(0.1, standarization=False)

print('Hybrid tree accuracy:', hybrid)
print('Discrete tree accuracy:', discrete)
print('Numeric tree accuracy:', numeric)

```

- Then let's plot the different accuracies for different threshold values

```

threshold_values = np.linspace(0.00, 0.3, 10)

accuracies_hybrid = []
accuracies_discrete = []
accuracies_numeric = []

```

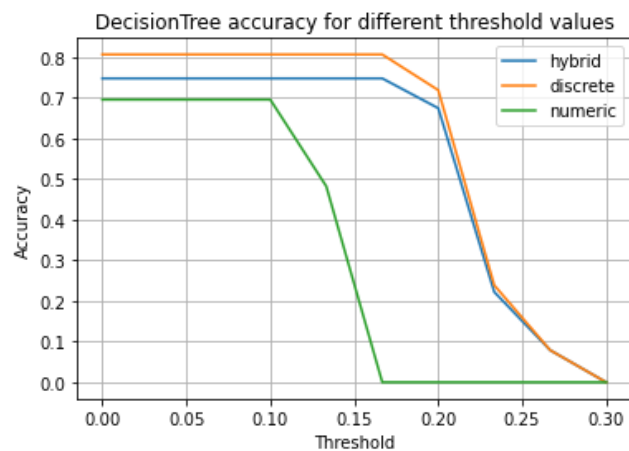
```

for threshold in threshold_values:
    hybrid, discrete, numeric = get_accuracy(threshold, n_iterations=10)
    accuracies_hybrid.append(hybrid)
    accuracies_discrete.append(discrete)
    accuracies_numeric.append(numeric)

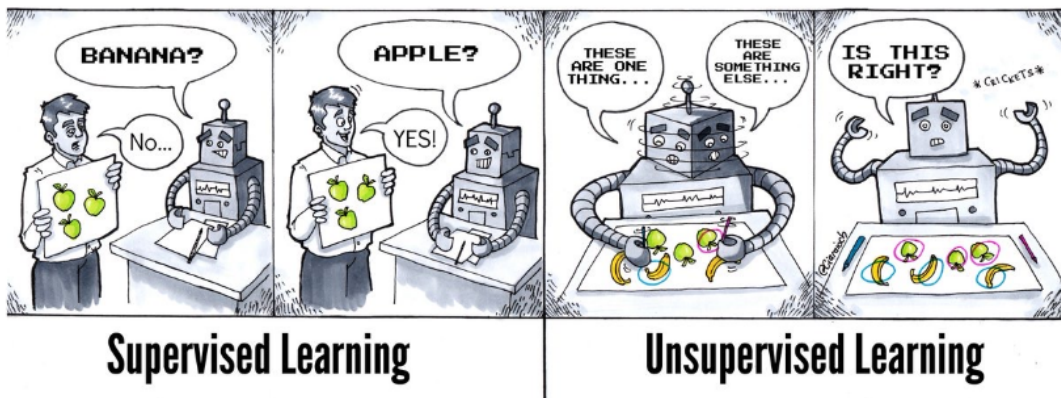
_, axis = plt.subplots()
axis.plot(threshold_values, accuracies_hybrid, label = 'hybrid')
axis.plot(threshold_values, accuracies_discrete, label = 'discrete')
axis.plot(threshold_values, accuracies_numeric, label = 'numeric')

axis.legend()
axis.set_xlabel('Threshold')
axis.set_ylabel('Accuracy')
plt.title('DecisionTree accuracy for different threshold values')
plt.grid()
plt.show()

```



Unsupervised learning



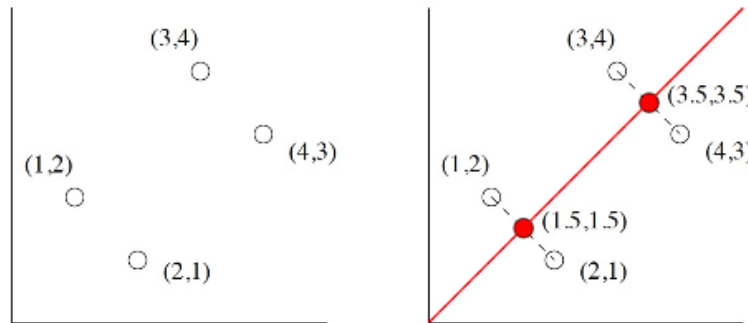
Dimintality reduction

- We may be able to improve the “distinctivness” of the classifier by using more features, however some features are more useful than others, some being completely useless.
- Reducing the dimension has multiple benefits:
 - Curse of dimensionality: more features require more data to classify the object
 - Speed
 - Size
 - 2D/3D visual discovery of data structure (64 d is impossible for a human to grasp)
 - Removes redundancy features (i.e. that have strong correlation)

- Removes noisy features
- Intrinsic dimensionality: A data set might be reduced to a lower dimension without loss of information
- Multiple factors come in to play when deciding which features to select. The statistical ones include
 - Variance:
 - The farther away the points are from the mean, the less it is like a constant (with the same value for both classes), which then it'd be rather useless. So the more variance the better
 - Confidence (that it is a distinctive feature)
- Other option: combine features together... leads to PCA

Principal component Analysys (PCA)

- You make up a linear combination of n features (into a new feature in a smaller dimension), thus giving a (different) weight to each feature that creates a new one, which reduces the dimension of the dataset.



- The choice for these weights is based on PCA:
 - Principal Components Analysis maps the data onto a linear subspace, such that the variance of the projected data is maximized.

Linear algebra re-cap

Variance vs covariance

- Variance = the average squared difference of the data points against the mean
- Covariance = the direction in which two random variables change when they are compared to each other

- **Variance:**
 - Variance is used in statistics to describe the spread between a data set from its mean value.

$$\sigma_x = \frac{1}{N} \sum_{n=1}^N (x_n - \frac{1}{N} \sum_{n=1}^N x_n)^2$$

$$\sigma_x = \frac{1}{N} \sum_{n=1}^N (x_n - \mu_x)^2$$

Where N is number of data points / examples

- **Covariance**
 - A covariance refers to the measure of how two random variables will change when they are compared to each other.

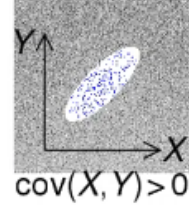
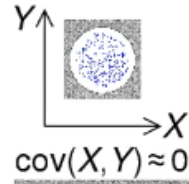
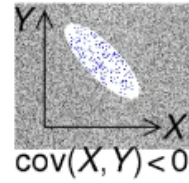
$$\sigma_{xy} = \frac{1}{N} \sum_{n=1}^N (x_n - \mu_x)(y_n - \mu_y) = E[(x_n - \mu_x)(y_n - \mu_y)]$$

- The covariance matrix contains all the covariances

Covariance matrix

- The *covariance matrix* is the matrix with all pairwise covariances:

$$\Sigma = \begin{bmatrix} \sigma_{x_1} & \sigma_{x_1x_2} & \dots & \sigma_{x_1x_D} \\ \sigma_{x_2x_1} & \sigma_{x_2} & \dots & \sigma_{x_2x_D} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{x_Dx_1} & \sigma_{x_Dx_2} & \dots & \sigma_{x_D} \end{bmatrix}$$



- We can shift the data points to have 0 means to make computation easier:

The *covariance matrix* is the matrix with all pairwise covariances:

$$\Sigma = \begin{bmatrix} E[(x_1 - \mu_1)(x_1 - \mu_1)] & E[(x_1 - \mu_1)(x_2 - \mu_2)] & \dots & E[(x_1 - \mu_1)(x_D - \mu_D)] \\ E[(x_2 - \mu_2)(x_1 - \mu_1)] & E[(x_2 - \mu_2)(x_2 - \mu_2)] & \dots & E[(x_2 - \mu_2)(x_D - \mu_D)] \\ \vdots & \vdots & \ddots & \vdots \\ E[(x_D - \mu_D)(x_2 - \mu_1)] & E[(x_D - \mu_D)(x_2 - \mu_2)] & \dots & E[(x_D - \mu_D)(x_D - \mu_D)] \end{bmatrix}$$

What if the data is zero-mean?

$$\mu_n = 0 \text{ for all } n$$

$$\Sigma = \begin{bmatrix} E[(x_1)(x_1)] & E[(x_1)(x_2)] & \dots & E[(x_1)(x_D)] \\ E[(x_2)(x_1)] & E[(x_2)(x_2)] & \dots & E[(x_2)(x_D)] \\ \vdots & \vdots & \ddots & \vdots \\ E[(x_D)(x_2)] & E[(x_D)(x_2)] & \dots & E[(x_D)(x_D)] \end{bmatrix} = \frac{1}{N} X^T X$$

X is the matrix of data points
Rows are single data points
columns are features

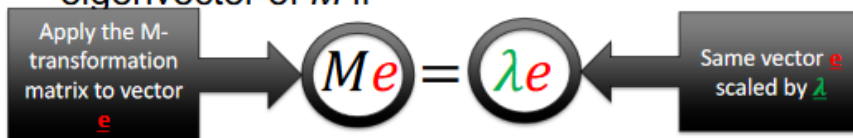
66

Eigen vectors and eigen values

- Eigen vector is a type of vector such that when transformed by multiplying it with a given matrix, the vector outcome is in the same same direction as the inputted (eigen) vector.
- That is, the outcome is a scaled version of the input. The scale is called eigen value

Eigenvalues & eigenvectors: Definition

- M square matrix, λ constant, e a non-zero column vector
- λ is an eigenvalue of M and e is the corresponding eigenvector of M if



- Avoiding ambiguity regarding length: eigenvector to be *unit vector*
- λ and e form eigenpairs

- Eigenpairs can be found with the characteristic equation:
 - $\det(A - \lambda I) = 0$
- Alternatively, you can use "power iteration", which bruteforces the calculation below until it converges:

- $$x_{k+1} := \frac{Mx_k}{\|Mx_k\|}$$

- Then calculate λ_1 as usual, by solving $Mx = \lambda x$ or by doing $\lambda = \mathbf{v}^T M \mathbf{v}$
- And get the second eigen pair by using power iteration on the new Matrix M^* :
 - $M^* = M - \lambda_1 x x^T$
 - $x x^T$ is also known as the **outer product** of \mathbf{v} with itself
- Eigen-decomposition is the factorization of a matrix into a principal form such that the matrix is represented in terms of it's eigen values and eigen vectors
- Only diagonalizable matrices can be factorized in this way
 - Since we are using the covariance matrix, which is by definition diagonalizable, we will always be able to decompose it.

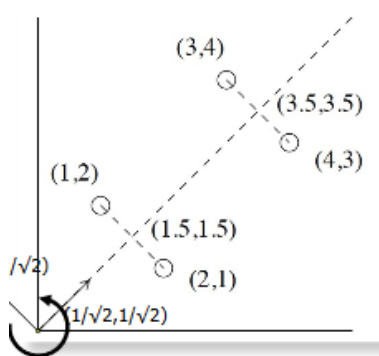
- **Matrix of eigenvectors for XX^T becomes**

$$E = \begin{bmatrix} 1/\sqrt{2} & -1/\sqrt{2} \\ 1/\sqrt{2} & 1/\sqrt{2} \end{bmatrix}$$

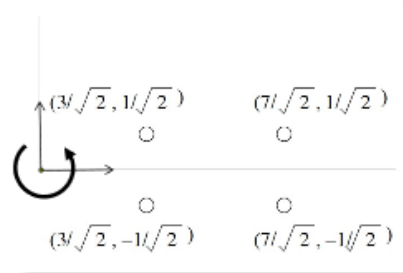
$$XE = \begin{bmatrix} 1 & 2 \\ 2 & 1 \\ 3 & 4 \\ 4 & 3 \end{bmatrix} \begin{bmatrix} 1/\sqrt{2} & -1/\sqrt{2} \\ 1/\sqrt{2} & 1/\sqrt{2} \end{bmatrix} = \begin{bmatrix} 3/\sqrt{2} & 1/\sqrt{2} \\ 3/\sqrt{2} & -1/\sqrt{2} \\ 7/\sqrt{2} & 1/\sqrt{2} \\ 7/\sqrt{2} & -1/\sqrt{2} \end{bmatrix}$$

- Any matrix of orthonormal vectors represents a rotation of the axes in a Euclidean space. Matrix E can be viewed as a rotation (in this case 45 degrees counterclockwise)

- First point $[1,2]$ transformed into $[3/\sqrt{2}, 1/\sqrt{2}]$



Original points, eigenvectors, projections



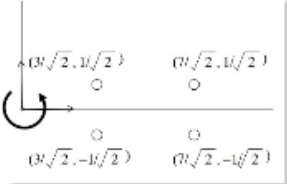
New coordinate system

- From 2d to 1d


$$E = \begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & 1/\sqrt{2} \end{bmatrix}$$

$$XE = \begin{bmatrix} 1 & 2 \\ 2 & 1 \\ 3 & 4 \\ 4 & 3 \end{bmatrix} \begin{bmatrix} 1/\sqrt{2} & -1/\sqrt{2} \\ 1/\sqrt{2} & 1/\sqrt{2} \end{bmatrix} = \begin{bmatrix} 3/\sqrt{2} & 1/\sqrt{2} \\ 3/\sqrt{2} & -1/\sqrt{2} \\ 7/\sqrt{2} & 1/\sqrt{2} \\ 7/\sqrt{2} & -1/\sqrt{2} \end{bmatrix}$$

New coordinate system



XE_1
New coordinate system
1 dimensional !



Maximizing the variance

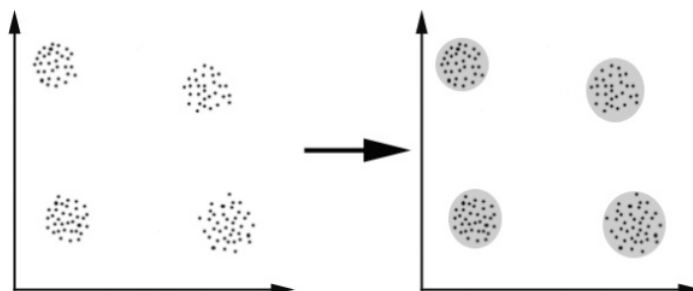
- Principal components (of the factorized version of the covariance matrix) are given by the eigen vectors of the covariance matrix
- To maximize variance:
 - The first principal component is given by the eigen vector with the highest eigen value

PCA issues

- Covariance is sensitive to large scaled values
 - Can be fixed by normalizing values (with 0 mean and 1 standard deviation): $\frac{x-\mu}{\sigma}$
- PCA assumes that underlying subspace is linear (1d = line, 2d = plane, 3d = cube? etc). If it isn't then PCA isn't the best choice
- PCA is unsupervised, hence:
 - maximizes overall variance of the data along a small set of directions
 - does not know anything about class labels
 - can pick direction that makes it hard to separate classes
- too expensive for many applications

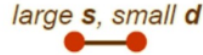
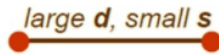
Clustering

- Finding natural groups in data where
 - Items within the group are close together
 - Items between groups are far apart



1. Proximity measure, either

- Similarity measure $s(x_i, x_k)$: large if x_i and x_k are similar, or
- Dissimilarity (distance) measure $d(x_i, x_k)$: small if x_i and x_k are similar



2. Criterion function to evaluate a clustering



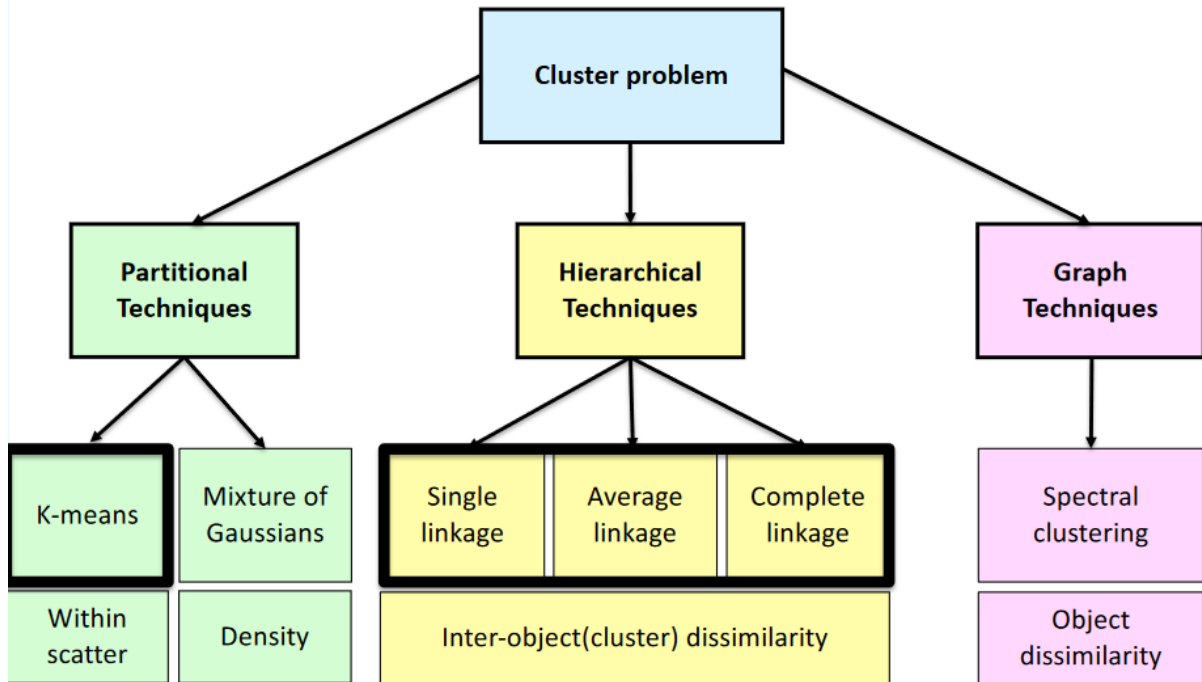
3. Algorithm to compute clustering

- Eg. By optimizing the criterion function

Cluster evaluation (a hard problem)

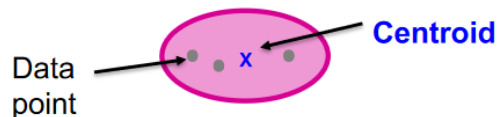
- Intra-cluster cohesion (compactness):
 - Cohesion measures how near the data points in a cluster are to the cluster's mean.
 - Sum of squared errors (SSE) is a commonly used measure.
- Inter-cluster separation (isolation):
 - Separation means that different cluster means should be far away from one another.
- In most applications, expert judgments are still the key

Clustering techniques

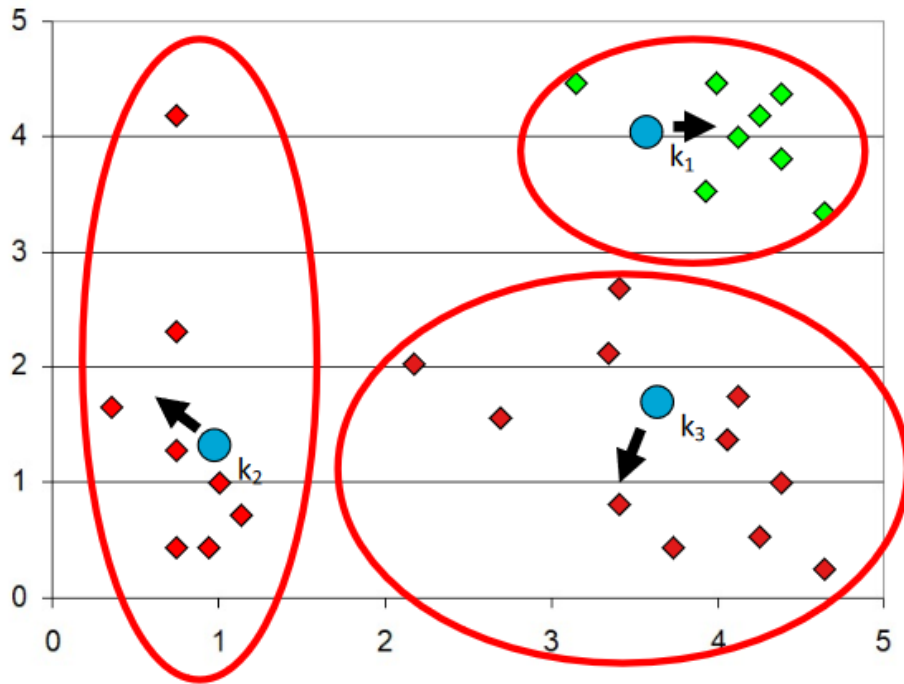
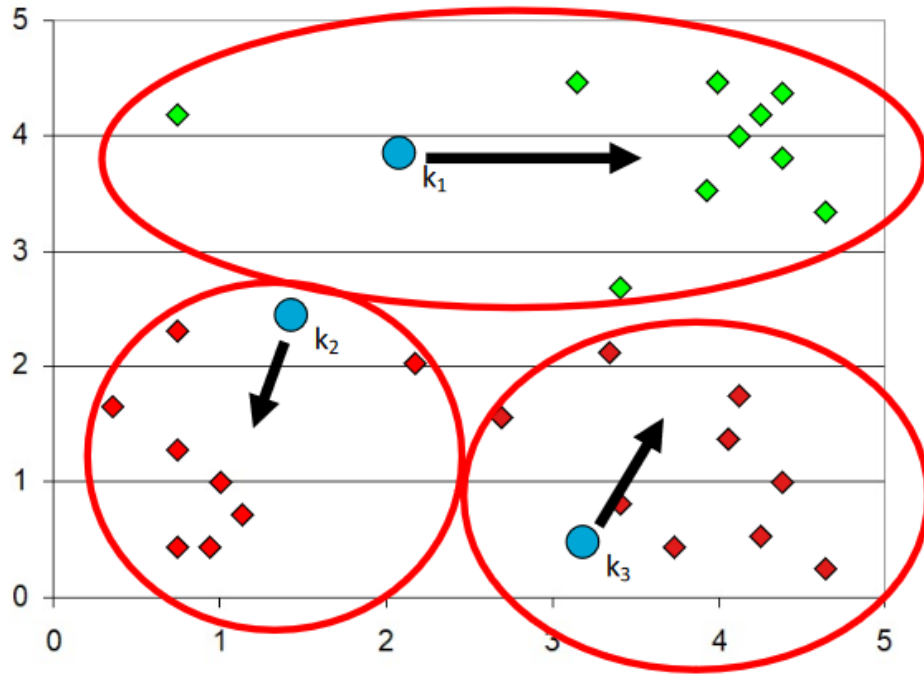


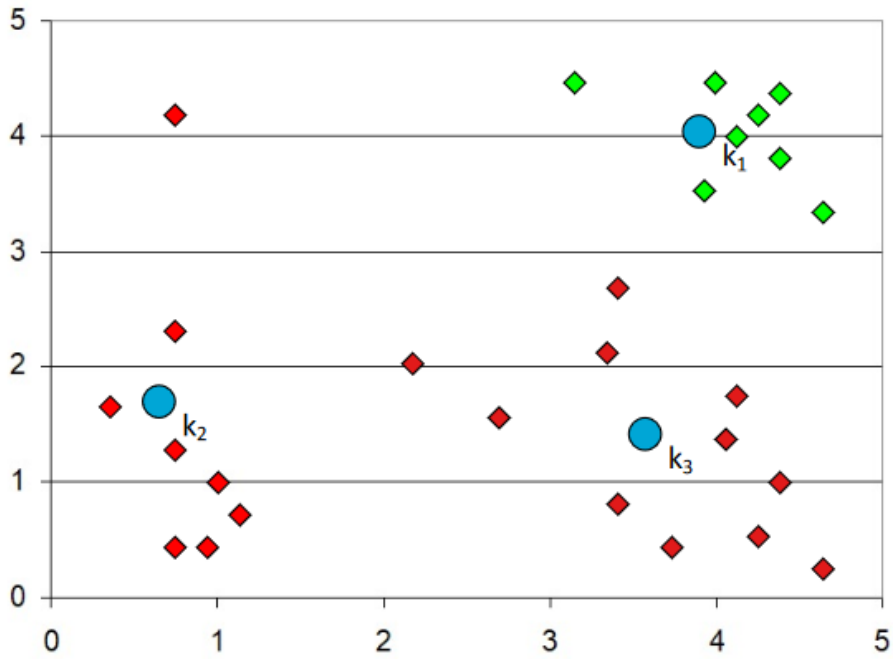
K-means clustering

- K-means (MacQueen, 1967) is a partitional clustering algorithm
- Let the set of n data points be $\{x_1, x_2, \dots, x_n\}$ where $x_i = (x_{i1}, x_{i2}, \dots, x_{ip})$ is a feature vector and p the number of dimensions.
- The k-means algorithm partitions the given data into k clusters:
 - Each cluster has a cluster centre (cluster mean), called centroid.
 - k is specified by the user



- Given k , the k-means algorithm works as follows:
 - Choose k (random) data points (seeds) to be the initial centroids, cluster centers
 - Assign each data point to the closest centroid
 - Re-compute the centroids using the current cluster memberships
 - If a convergence criterion is not met, repeat steps 2 and 3

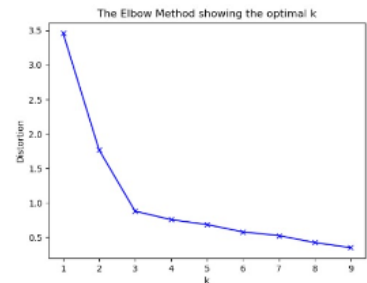
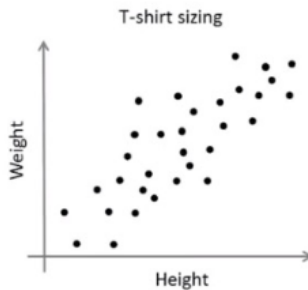
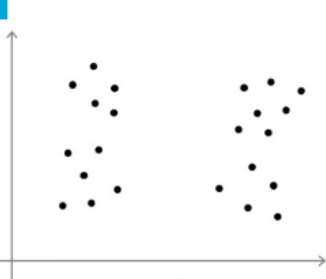




- Convergence criterion include:
 - no (or minimum) re-assignments of data points to different clusters
 - no (or minimum) change of centroids
 - minimum decrease in the sum of squared errors (SSE)

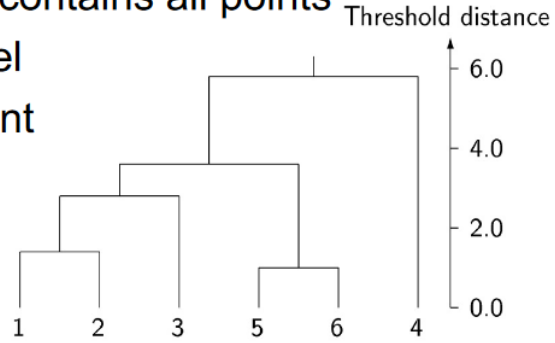
Choosing the number of clusters

- Inspect visually
- Known purpose
- Elbow method



Hierarchical clustering

- Instead of picking k find a hierarchy of structure
 - Course effects: top level contains all points
 - Fine-grained: bottom level one cluster per data point



Hierarchical clustering approaches

- Agglomerative (bottom-up):
 - each point starts as cluster
 - group two closest clusters
 - stop at some point

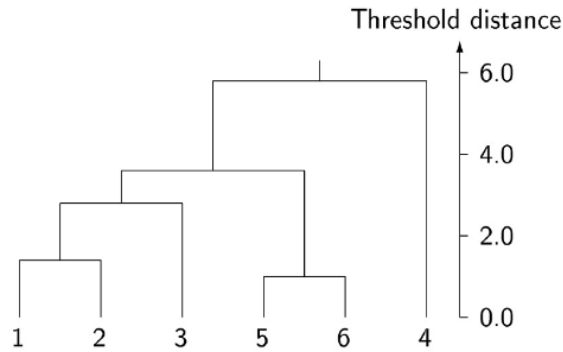


Figure 11.1 Dendrogram.

- Divisive (top-down):
 - all points start in one cluster
 - split cluster in some sensible way
 - stop at some point

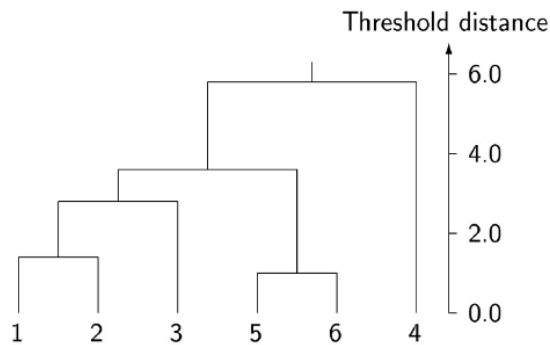
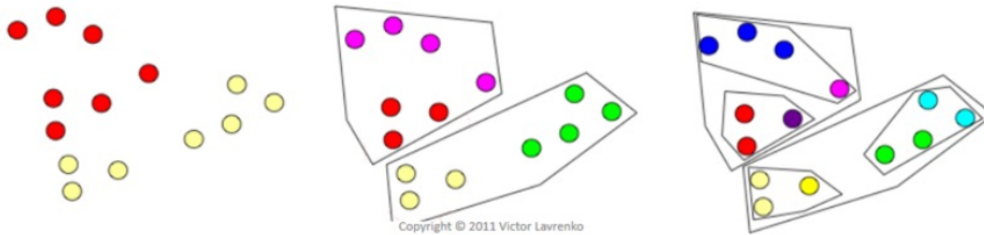


Figure 11.1 Dendrogram.

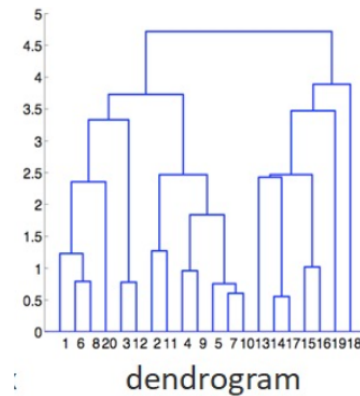
Divisive: hierarchical k-means

- Apply k-means recursively:
 - Run k-mean on the original data for $k=2$
 - For each of the resulting clusters run k-means with $k=2$



Agglomerative clustering

- Starting from individual observations, produce sequence of clusterings of increasing size
- At each level, two clusters chosen by criterion are merged



70

Agglomerative clustering

1. Determine distances between all clusters
2. Merge clusters that are closest
3. IF #clusters>1 THEN GOTO 1

Different merging rules

- **Single linkage:** two nearest objects in the clusters :

$$g(R, S) = \min_{ij} \{d(x_i, x_j) : x_i \in R, x_j \in S\}$$

- **Complete linkage:** two most remote objects in the clusters :

$$g(R, S) = \max_{ij} \{d(x_i, x_j) : x_i \in R, x_j \in S\}$$

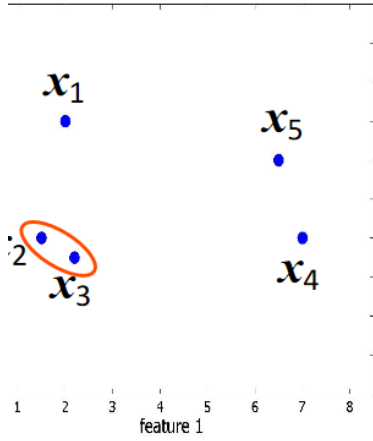
- **Average linkage:** cluster centres :

$$g(R, S) = \frac{1}{|R||S|} \sum_{ij} \{d(x_i, x_j) : x_i \in R, x_j \in S\}$$

Hierarchical clustering

- Step 1:**

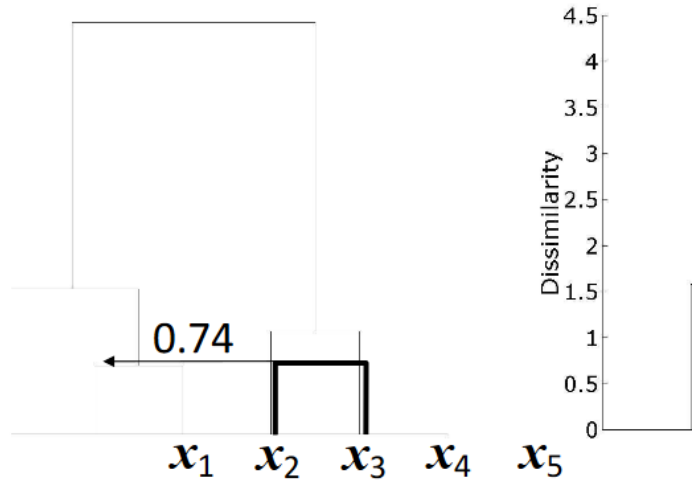
Find the most similar pair: $\min_{(i,j)} \{d(i,j)\} = d(2,3)$



	x_1	x_2	x_3	x_4	x_5
x_1	0.00	1.58	1.76	5.22	4.53
x_2		0.00	0.74	5.50	5.10
x_3			0.00	4.81	4.48
x_4				0.00	1.12
x_5					0.00

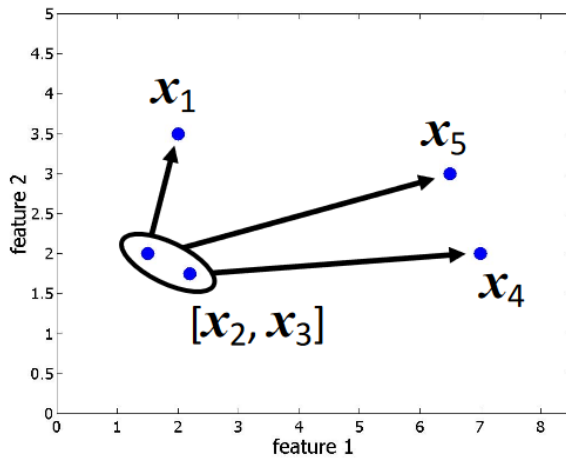
- Step 2:**

Merge x_2 and x_3 into a single object, $[x_2, x_3]$;



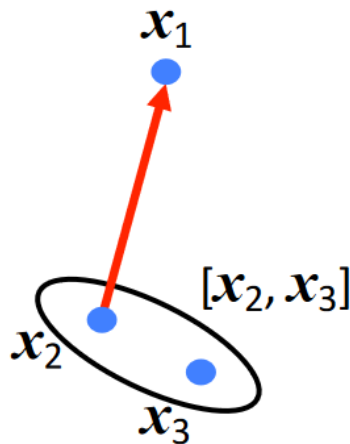
- **Step 3:**

Recompute D – what is the distance between $[x_2, x_3]$ and the rest?



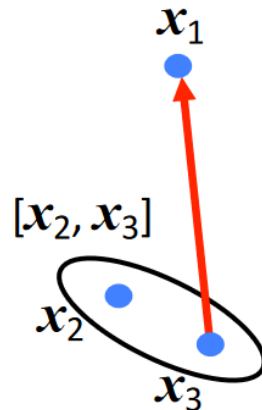
- **Step 3:**

Recompute D – **single linkage**: $d([x_2, x_3], x_1) = \min(d(x_1, x_2), d(x_1, x_3))$



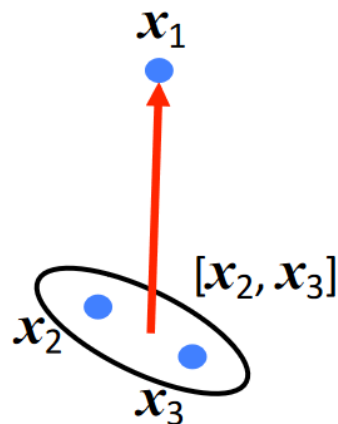
- **Step 3:**

Recompute D – **complete linkage**: $d([x_2, x_3], x_1) = \max(d(x_1, x_2), d(x_1, x_3))$

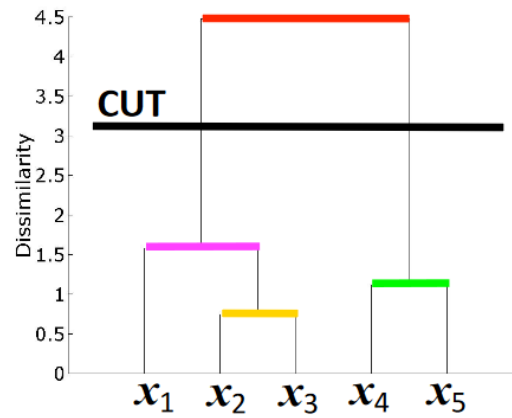
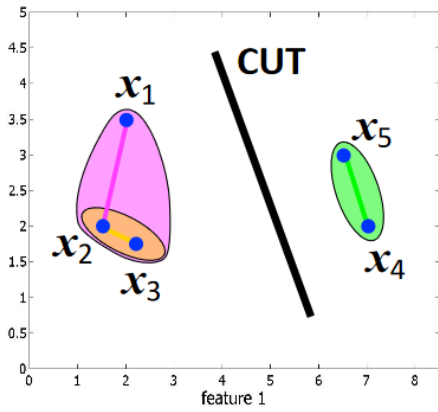
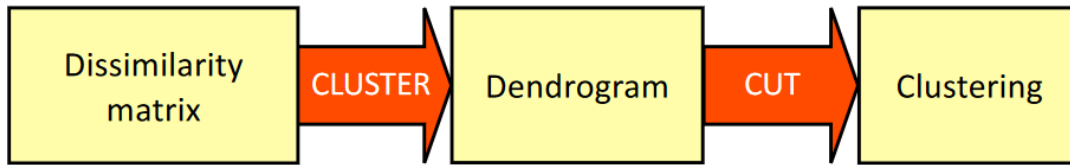
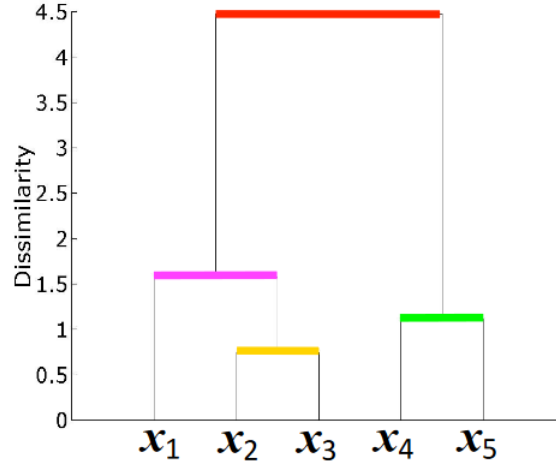
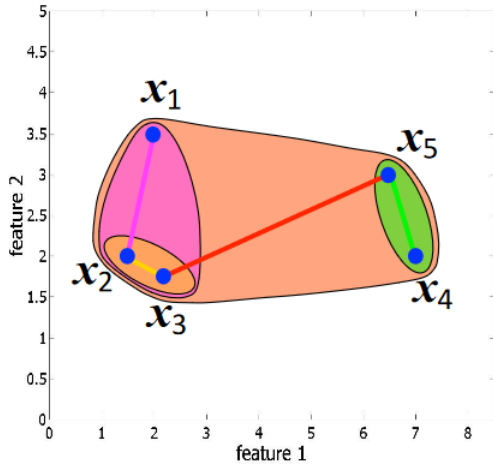


- **Step 3:**

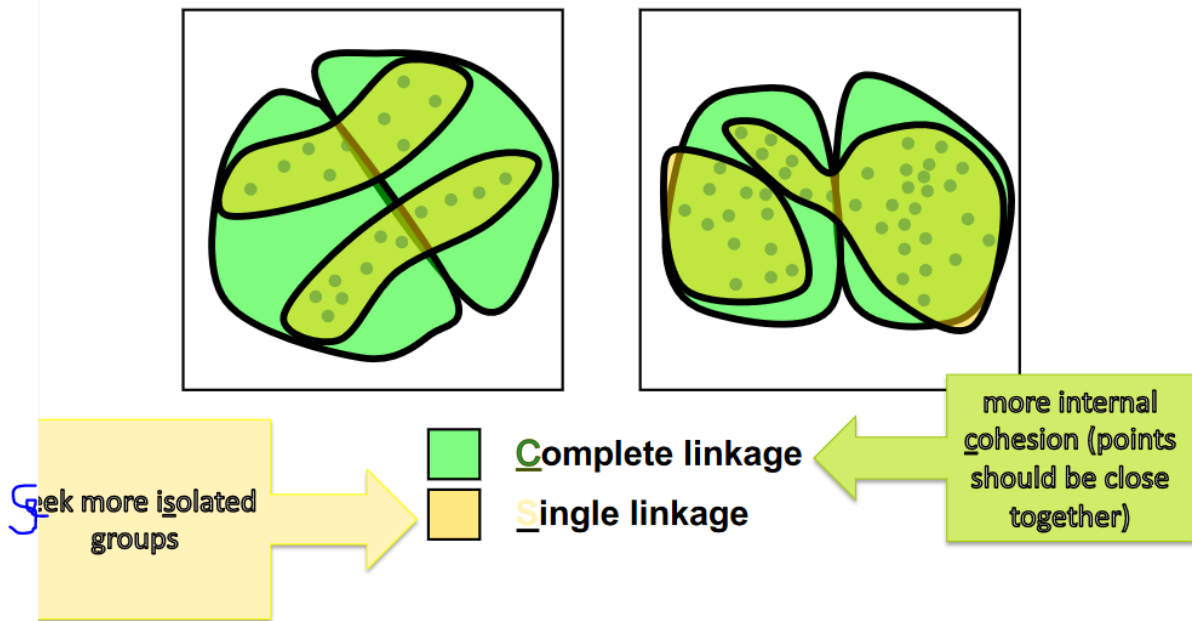
Recompute D – **average linkage**: $d([x_2, x_3], x_1) = \text{mean}(d(x_1, x_2), d(x_1, x_3))$



- Repeat steps 1-3 until a single cluster remains



Linkage and cluster shape



- Pros
 - Dendrogram gives overview of all possible clusterings
 - Linkage type allows to find clusters of varying shapes
 - Different dissimilarity measures can be used
- Cons
 - Computationally intensive
 - Clustering limited to "hierarchical nestings"

Lab: Unsupervised learning

Learning without examples

- When we want to learn from data without knowing the labels, we apply unsupervised learning: different techniques to make sense of data from the data itself.
- An example of a task where unsupervised learning can be successfully applied is dimensionality reduction: trying to find the 'essential' features or combinations of features to describe objects.

Finding the eigen pairs

- The Power Iteration method is a relatively simple method for calculating eigenvectors for a square ($D \times D$) matrix M . The process works as follows:
 1. Construct a vector \mathbf{v}_0 of ones of length $D \times 1$.
 2. Until convergence, compute:

$$\mathbf{v}_{k+1} = \frac{M\mathbf{v}_k}{\|M\mathbf{v}_k\|}$$
 where $\|M\mathbf{v}_k\|$ is the (L2) norm (length) of $M\mathbf{v}_k$.
 3. Output vector \mathbf{v} as the principal eigenvector of M .
 4. Compute M^* as:

$$\lambda = \mathbf{v}^T M \mathbf{v} \quad M^* = M - \lambda \times \mathbf{v} \mathbf{v}^T$$
 5. If more eigenvectors are required, go to step 1 with M^* as input.

```
import numpy as np
import numpy.linalg as la
```

```

def power_iteration(matrix, n_vectors, e):
    """
    This function returns a list with `n_vectors` amount of eigenvectors (numpy vectors)
    `matrix` and the convergence parameter `e`.
    :param matrix: the square matrix
    :param n_vectors: the number of eigenvectors
    :param e: the convergence parameter
    :return: the list of eigenvectors found
    """
    assert (matrix.shape[0] == matrix.shape[1] & matrix.shape[1] >= n_vectors)

    eigen_vectors = list()

    # START ANSWER
    v_0 = np.ones(matrix.shape[0])
    v_k1 = v_0
    M_v0 = matrix @ v_0
    i = e+1
    j = 0

    while e <= i:
        M_v0 = matrix @ v_0
        v_k1 = M_v0/la.norm(M_v0)

        i = abs(la.norm(v_0) - la.norm(v_k1))
        v_0 = v_k1

    principal = v_k1
    principal_v = principal.T @ matrix @ principal
    eigen_vectors.append(principal)

    matrix_star = matrix -(principal_v * np.outer(principal,principal))

    while n_vectors > 1:
        eigen_vectors += power_iteration(matrix_star, n_vectors-1, e)
        n_vectors -= 1

    # END ANSWER

    return eigen_vectors

```

- you will need to read data from a file. Use the read_data function below to do this.

```

def read_data(file_name):
    """
    This function loads a given matrix data file into a numpy matrix.
    :param file_name: name of the file to be read
    :return: the data as a numpy array
    """

    lines = [line.rstrip('\n') for line in open(file_name)]

    result = np.zeros((len(lines), len(lines[0].split(" "))))

    for (i, line) in enumerate(lines):
        line = line.split(" ")
        for (j, number) in enumerate(line):
            result[i][j] = float(number)

    return result

```

```
# Read the data and call the power_iteration function for this exercise.
```



```
# START ANSWER
power_iteration(read_data("data/matrix.txt"),2,1.0e-100)
# END ANSWER
```

- You will see that the subsequent calculated eigen pairs have smaller eigen values.

Principal component analysis

- We will use Principal Component Analysis to find the principal components in a dataset. Principal components can be seen as vectors along which most variance is found in the data
- We will now create a matrix that reads data from `data/gaussian.txt`. The size of this matrix is $N \times 2$, where N is the number of points in the dataset. Then, we will use `plot_data()` to plot the data.

```
import matplotlib.pyplot as plt
%matplotlib inline

def plot_data(data, eigen_vectors = None):
    """
    This function plots the data of the given `eigen_vectors` with a scatterplot of the data.
    If no eigenvectors are available, it just plots the data
    :param data: the data
    :param eigen_vectors: the eigenvectors
    """
    # Plot the features as a scatterplot
    x = [[el[0]] for el in data]
    y = [[el[1]] for el in data]
    plt.scatter(x, y)

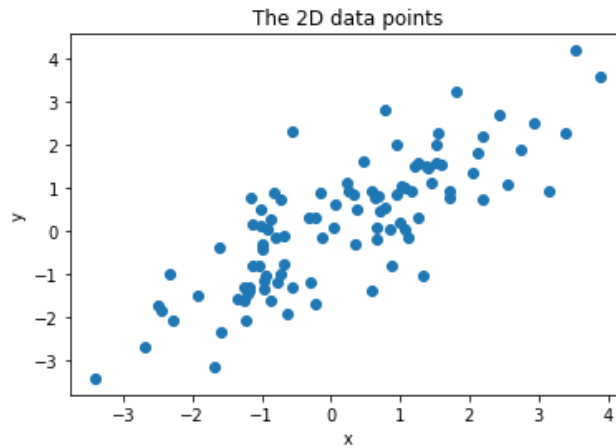
    if eigen_vectors:
        # Plot the two PCA Lines
        for vector in eigen_vectors:
            line = _set_line(vector)
            plt.plot(line[0], line[1], 'red')

    plt.show()

def _set_line(vector):
    # Fixed number for the line size of this plot
    line_size = 6

    # Set the coordinates for the PCA Lines
    axis = np.zeros((2, 2))
    axis[0][0] = vector[0] * line_size
    axis[1][0] = vector[1] * line_size
    axis[0][1] = vector[0] * -line_size
    axis[1][1] = vector[1] * -line_size
    return axis

data = read_data("data/gaussian.txt")
plt.title('The 2D data points')
plt.xlabel('x')
plt.ylabel('y')
plot_data(data)
```



- The principal components of a dataset can also be seen as the eigenvectors of the covariance matrix. Compute the covariance matrix of the Gaussian dataset as follows:

$$\text{cov}(X) = \sum \frac{1}{N} (X - \bar{x})^T (X - \bar{x})$$

- where \bar{x} is the mean row of $N \times D$ dimensional data matrix X . Next, compute the eigenvectors of this covariance matrix and plot the vectors accordingly.

```
def covariance(data):
    """
    This function computes the covariance matrix of a given `data`.
    :param data: the starting data
    :return: the covariance matrix
    """
    # START ANSWER
    mean = np.mean(data, axis=0)
    matrix = np.array([0,0])

    for i in range(data.shape[0]):
        matrix = matrix + 1/data.shape[0] * np.outer(data[i]-mean,data[i]-mean)

    # END ANSWER
    return matrix
```

```
# Now let's check your implementation with the numpy built-in cov function.
data = read_data("data/gaussian.txt")
your_cov_matrix = covariance(data)
np_cov_matrix = np.cov(np.transpose(data))

# Your value might differ slightly as the numpy built-in cov function is a bit more prec
err_msg = "Your covariance matrix is allowed to differ from the numpy matrix, but no mor
np.testing.assert_allclose(np_cov_matrix, your_cov_matrix, atol=0.025, err_msg=err_msg)
```

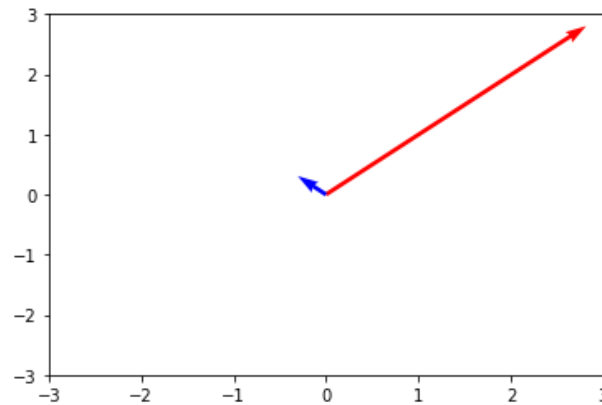
```
# Use power iteration method to compute the eigenvectors using your covariance matrix and
# The plot should contain both the dataset (like in the above plot) and the eigenvectors
# START ANSWER
V = power_iteration(your_cov_matrix, your_cov_matrix.shape[0], 1.0e-100)
v_1 = V[0].T @ your_cov_matrix @ V[0]
v_2 = V[1].T @ your_cov_matrix @ V[1]

V = np.array([v_1 * V[0], v_2 * V[1]])
origin = np.array([[0, 0],[0, 0]]) # origin point

plt.quiver(*origin, V[:,0], V[:,1], angles='xy', scale_units='xy', color=['r','b','g'],
plt.xlim(-3, 3)
```

```
plt.ylim(-3, 3)
plt.show()
```

```
# END ANSWER
```



PCA of faces

- you will need to use the `create_image` function presented below.

Note: If you have trouble installing PIL, try `pip install Pillow`

```
import math
from PIL import Image

def create_image(fv):
    """
    This function creates a grey image based on the given feature vector `fv`.
    :param fv: the feature vector
    :param title: the title of the image
    """
    width = height = int(math.sqrt(len(fv)))

    # Filter label and threshold from data
    img = Image.new('L', (width, height), "black")
    pixels = img.load()
    min_v = min(fv)
    max_v = max(fv)

    # Iterate over each pixel and set p value
    j = 0
    for (idx, p) in enumerate(fv):
        i = idx % width
        pixel = int(((p - min_v) / (max_v - min_v) * 255))
        pixels[i, j] = pixel
        if i == (width - 1):
            j += 1

    # Resize image to make it better visible
    img = img.resize((256, 256), Image.ANTIALIAS)
    return img
```

- Create a matrix object and let it read in `data/faces.txt`. This matrix is of size $N \times D$, where N is the number of images in the dataset and D is the number of pixels per image (in this case 32×32 , so $D = 1024$). Similar as before, compute the covariance matrix. Use this covariance matrix to compute the first 10 principal components and visualize them using the given `create_image()` function.

Note: `create_image()` only accepts rows of the dataset (`lists`), use the `transpose` function of `numpy` to convert columns to rows.

```

from IPython.display import display # To display images, usage: display(image)

data = read_data("data/faces.txt")

# Display image of the mean
mean = np.mean(data, axis=0)
print("Mean image")
mean_image = create_image(mean)
display(mean_image)

# Now plot the image of each of the eigenvectors of the given dataset
# START ANSWER
cov_matrix = np.cov(np.transpose(data))
V = power_iteration(cov_matrix, 10, 1.0e-100)

for i in range(10):
    mean_image = create_image(V[i])
    display(mean_image)

# END ANSWER

```

- mean face:



- (principal) eigen vector face (other 9 vector faces are skipped here):



You can see how the eigen vector highlights the mouth, the eyes, the nostrils and the top of the forehead as the regions of the face with most variance

K-means for the most varied features (audio of digits)

- In this assignment, you will work with the *TIDIGITS* dataset. This dataset was created by Texas Instruments (hence, TI) and is a set of voice recordings of digits. You can go ahead and give the audio files a listen in the data folder: `data/tidigits/...`
- Before we can use any clustering algorithms on this data, we need a way to describe our audio files in a way that a computer can understand. An audio file is read as an array, where each value in the array is the amplitude of the audio at a the corresponding time. We refer to this data as *raw* waveforms. It's infeasible to use the raw data for clustering. We need to

extract a limited number of features to describe each audiofile: we perform **feature extraction**.

- We will extract *MFCC* features, which are often used for speech processing. To extract these, we split the audio file in frames of 25 ms each. Next, we apply a number of complicated operations to retrieve 13 features per frame. If a file is split into, for example, 100 frames, this means we have $13 * 100 = 1300$ features! To bring this number down, we sample 5 frames from regular intervals (the size of the interval is dependent on the length of the audio file) and flatten this to an array of $5 * 13 = 65$ features (this number is thus independent of the length of the audio file).
- Run the code below to extract MFCC features for the 150 files provided. This will give you a dataset with 50 'one' audiofiles, 50 'two' audiofiles and 50 'three' audiofiles.

Before doing this, you need to have `python_speech_features` and `soundfile` installed. These can be installed with the command:

```
pip install python_speech_features SoundFile
```

If this fails, you can use the backup dataset that is provided down below.

```
import numpy as np
import python_speech_features as features
import soundfile as sf
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline

def extract_mfcc(sound, sample = 5):
    # Read in the flac file
    data, samplerate = sf.read(sound)

    # Extract MFCC features.
    # For each frame (25ms segment of audio) in the file, we get 13 MFCC features,
    # giving us a [n x 5] matrix of features
    mfcc_feat = np.asarray(features.mfcc(data,samplerate), dtype='float32')

    # We sample 5 frames and flatten the feature vectors into one large 'supervector'.
    idx = np.floor(np.linspace(0, mfcc_feat.shape[0] - 1, sample)).astype(int)
    mfcc_sampled = mfcc_feat[idx]
    mfcc_feat_vector = mfcc_sampled.flatten()

    return mfcc_feat_vector

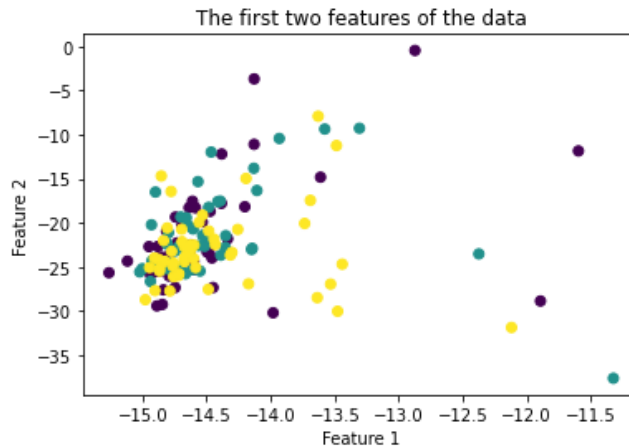
# Read audiofiles and extract MFCC feature vectors
one = []
for i in range(50):
    feat = extract_mfcc("data/tidigits/1/{:d}.flac".format(i))
    one.append(feat)
two = []
for i in range(50):
    feat = extract_mfcc("data/tidigits/2/{:d}.flac".format(i))
    two.append(feat)
three = []
for i in range(50):
    feat = extract_mfcc("data/tidigits/3/{:d}.flac".format(i))
    three.append(feat)

# Concatenate into one large dataset
X_train = np.concatenate((one, two, three))
ones = np.ones(50)
y_train = np.concatenate((ones, ones * 2, ones * 3)).astype(int)
np.savetxt("data/tidigits_features.txt", X_train)
np.savetxt("data/tidigits_targets.txt", y_train)
```

Feature reduction

- Now we have 150 data points (i.e. 150 audio files) and each data point consists of 65 features describing the audio file. Let's plot the the first two features.

```
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train)
plt.title('The first two features of the data')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```



- Obviously these (arbitrarily selected) features are not useful for making up distinct groups. But we can use PCA to get the eigen vectors of the features with most variance (and hence easier to group into distinct groups)
- We'll go for 4 features, this involves:
 - compute the covariance matrix of the dataset
 - compute the first four eigenvectors (i.e. principal components) of the covariance matrix
 - reorient the data points from the original axes to the ones represented by these principal components (this is done using the dot product).

```
import numpy.linalg as la
# First compute the covariance matrix of the dataset

# START ANSWER
cov_X = np.cov(np.transpose(X_train))
# END ANSWER

# Next, retrieve four eigenvectors of the covariance matrix
# CHALLENGE: you can use your own implementation of power iteration from part I of the L

# START ANSWER
def normalize(v):
    return v / la.norm(v)

def eigen_value(matrix, v):
    return np.matmul(np.matmul(v.T, matrix), v)[0][0]

def power_iteration(matrix, n_vectors, e):
    """
    This function returns a list with `n_vectors` amount of eigenvectors (numpy vectors)
    `matrix` and the convergence parameter `e`.
    :param matrix: the square matrix
    :param n_vectors: the number of eigenvectors
    :param e: the convergence parameter
    :return: the list of eigenvectors found
    """
```

```

assert (matrix.shape[0] == matrix.shape[1] & matrix.shape[1] >= n_vectors)

eigen_vectors = []
for i in range(n_vectors):
    # 1. Create vector consisting of ones
    v = np.ones((matrix.shape[0], 1))
    v_ = normalize(np.matmul(matrix, v))

    # 2. Compute until convergence -> L2 norm of the difference between v_ and v < e
    while la.norm(v - v_) >= e:
        v = v_
        v_ = normalize(np.matmul(matrix, v))

    # 3. Output vector v as the principal eigenvector of M
    eigen_vectors.append(v_)

    # 4. Compute M*
    matrix = matrix - eigen_value(matrix, v_) * np.matmul(v_, v_.T)

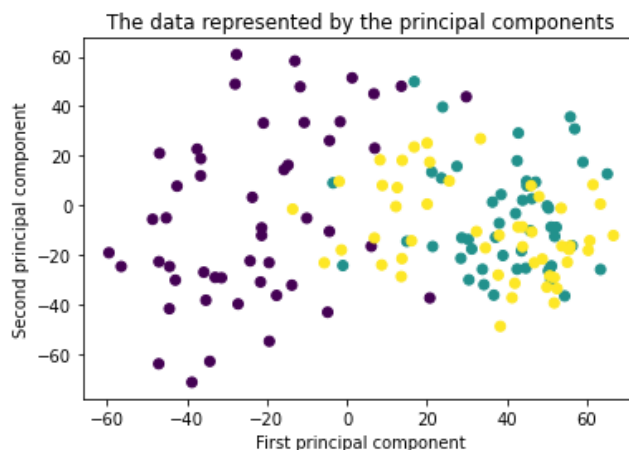
return eigen_vectors

eigen_vectors = power_iteration(cov_X, 4, 10 ** -5)
# END ANSWER

# Finally, we project our points onto the eigenvectors (principal components) using matrix multiplication
X_train_reduced = np.zeros((X_train.shape[0], 4))
for i in range(len(X_train)):
    X_train_reduced[i, :] = np.matmul(X_train[i, :].T, eigen_vectors).T

# And plot the points with the new data
plt.scatter(X_train_reduced[:, 0], X_train_reduced[:, 1], c=y_train)
plt.title('The data represented by the principal components')
plt.xlabel('First principal component')
plt.ylabel('Second principal component')
plt.show()

```



Clustering algorithm

- We will now implement the (original) k-means clustering algorithm. This algorithm works as follows: we start with k clusters and pick some random centre (mean) for each cluster. Next, we assign points to each cluster based on the distance to the centres and we update the centre to be the mean of all points in that cluster. This is repeated until the centres have converged.
- Pointwise, the steps of the k-means clustering algorithm approach are as follows:
 - initialize k cluster centers at random locations
 - assign each point to a cluster
 - update the cluster centers
 - go to step 2 unless converged or a certain number of iterations has been reached.

- First, we will create a `Cluster` class.

```
import numpy as np

# This object is used to store the clusters. A Cluster object consists of a numpy matrix
# containing all feature vectors in the cluster and the centroid of all the vectors.
# The object also contains a boolean for speedup purposes.
class Cluster(object):

    def __init__(self, array=np.array([])):
        self.changed = True
        self.data = np.array(array)
        self.cd = self.data

    def reset_cluster(self):
        self.data = np.array([])

    def is_changed(self):
        return self.changed

    def set_changed(self, changed):
        self.changed = changed

    def set_centroid(self, vector):
        self.cd = vector

    def append(self, other):
        # Set changed flag to true (the cluster has changed)
        self.set_changed(True)

        self.data = np.vstack((self.data, other))

    def centroid(self):
        # If the matrix consists of 1 vector, no need to compute centroid.
        if len(np.shape(self.data)) == 1:
            return self.data
        # Check whether the cluster has changed since last computation (for speedup)
        # and update the changed flag.
        # START ANSWER
        if(self.changed):
            self.changed = False
            self.cd = np.mean(self.data, axis = 0)
        # END ANSWER
        return self.cd

# Test case for the Cluster class
c = Cluster(np.array([[0, 1], [2, 0]]))

# Verifies that the centroid is calculated correctly
np.testing.assert_array_equal(c.centroid(), np.array([1.0, 0.5]))

# Verifies that the centroid is calculated correctly after a new data point has been add
c.append(np.array([1, 2]))
np.testing.assert_array_equal(c.centroid(), np.array([1.0, 1.0]))
```

- Step 1: initialize k cluster centers at random locations

```
from random import sample

# This function selects random k points from the dataset. For each random point it initi
# and adds the cluster to the list of clusters.
def add_init_points(points, clusters, k):
    # START ANSWER
    centroids = sample(range(0, len(points) - 1), k)
```



```

#centroids = sample(List(points), k)
for c in centroids:
    cluster = Cluster(points[c])
    #cluster.set_centroid(c)
    clusters.append(cluster)
# END ANSWER
return clusters

# Verifies that a cluster has been added as a init point
init_points = add_init_points(np.array([[0, 0], [0, 0]]), [], 1)
assert len(init_points) == 1
np.testing.assert_array_equal(init_points[0].centroid(), np.array([0.0, 0.0]))

```

- Step 2: Assign each point to a cluster based on their distance to each centroid

```

def distance(p1, p2):
    # Euclidian distance between 2 points (in any space)
    # START ANSWER
    return np.linalg.norm(p1-p2)
    # END ANSWER

# Verifies that the distance metric is correct
np.testing.assert_array_equal(distance(np.zeros([1, 2]), np.ones([1, 2])), np.sqrt(2))

```

- Step 3: we perform an iteration of the k-means clustering algorithm. This consists of the following steps:
 - calculate the centroids of each cluster and save them
 - remove all points from all clusters
 - add each point to the closest cluster centroid (the centroids that you saved earlier).

```

import sys

# This function updates the list of k clusters
def update_k_means(points, clusters, k):
    # Reset clusters
    centroids = []

    # Add initial points
    # If add_clusters is true, initialise clusters with add_init_points
    # Then add the cluster centroids to the centroids list
    add_clusters = len(clusters) == 0
    # START ANSWER
    if (add_clusters):
        clusters = add_init_points(points, clusters, k)
    #print(clusters)
    for c in clusters:
        centroids.append(c.centroid())
    # END ANSWER

    # Reset clusters from last iteration,
    # so the clustering can be performed with new centroids
    for cluster in clusters:
        cluster.reset_cluster()

    clusters = [None for e1 in range(k)]
    for p in points:
        # Calculate the min distance to one of the centroids
        # START ANSWER
        min_dist = sys.float_info.max
        label = 1
        for c in range(len(centroids)):
            temp_min = distance(p, centroids[c])
            if (temp_min < min_dist):

```

```

        min_dist = temp_min
        label = c
    # END ANSWER

    # Add the data point to the cluster with min_distance to centroid
    if label >= 0:
        if clusters[label] is None:
            clusters[label] = Cluster(p)
        else:
            clusters[label].append(p)

    return clusters

```

- Step 4: plot the clustered data and have it analyse the TIDIGITS dataset with k set to 3.

```

import matplotlib.cm as cm

def plot_k_means_data( clusters, k, itr):
    colors = cm.brg(np.linspace(0,1,k))
    for (i,cl) in enumerate(colors):
        x = [[el[0]] for el in clusters[i].data]
        y = [[el[1]] for el in clusters[i].data]
        plt.scatter(x, y, c=[cl])
        plt.scatter(clusters[i].centroid()[0], clusters[i].centroid()[1], c='black')
        plt.title("Clusters at update " + str(itr))
    plt.show()

def plot_k_means_data_update(clusters_prev, clusters, k):
    if(clusters_prev == []):
        return
    colors = cm.brg(np.linspace(0,1,k))
    for (i,cl) in enumerate(colors):

        x = [[el[0]] for el in clusters[i].data]
        y = [[el[1]] for el in clusters[i].data]
        plt.scatter(x, y, c=[cl], marker = '*')

        plt.scatter(clusters[i].centroid()[0], clusters[i].centroid()[1], c='black')

        x_prev = [[el[0]] for el in clusters_prev[i].data]
        y_prev = [[el[1]] for el in clusters_prev[i].data]

        plt.scatter(x_prev, y_prev, c=[cl], alpha=0.2, s=100)

        plt.scatter(clusters_prev[i].centroid()[0], clusters_prev[i].centroid()[1], c='g')
        plt.legend(['New clusters', 'New centroids', 'Previous clusters', 'Previous cent
                    loc='center left', bbox_to_anchor=(1, 0.5))
        plt.title('Difference between updates')
    plt.show()

```

```

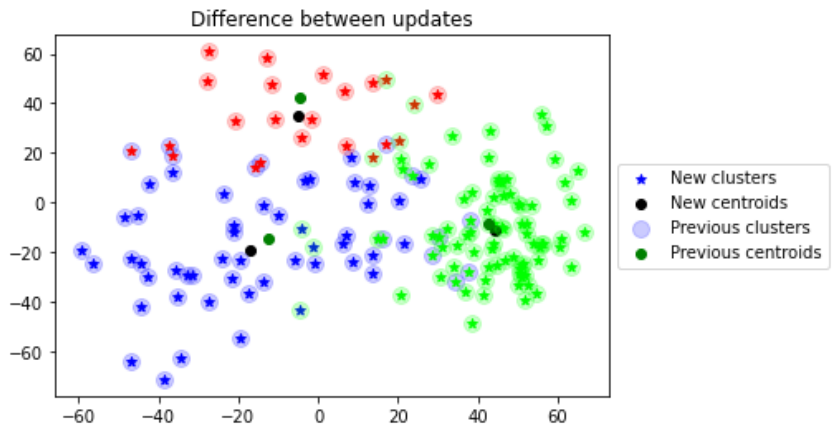
import copy
import random

random.seed(42)
points = X_train_reduced # Ex 1.5a
# points = X_train # Ex 1.5b (optional)
clusters = []
clusters_prev = []
centroids = []
k = 3
centroids_prev = [np.zeros((points.shape[1])) for x in range(0,k)]

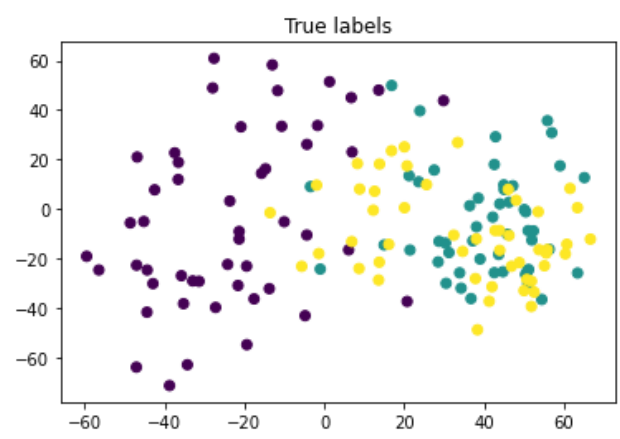
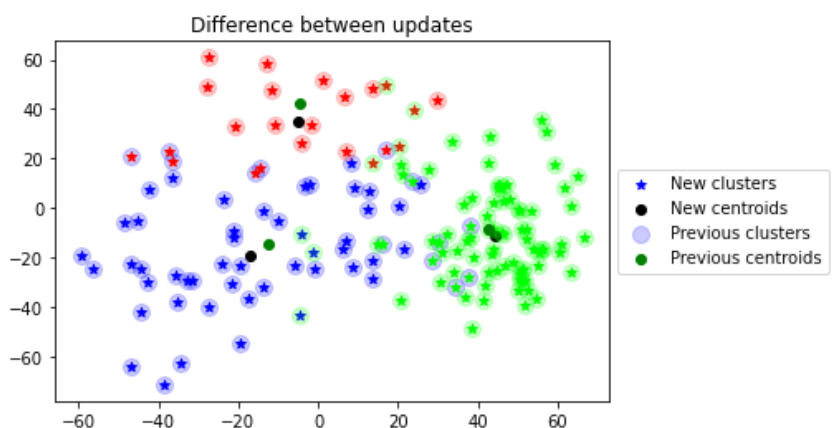
# Run here the update_k_means function with k=3 using the clusters list.
# Keep updating until there is no change in centroids in consecutive iterations

```

```
# or a maximum number of iterations (e.g. 10) is reached.  
  
# START ANSWER  
i = 0  
clusters_prev = update_k_means(points, clusters_prev, k)  
cluster = copy.deepcopy(clusters_prev)  
while i <= 9 and any(c is not None and c.is_changed() for c in cluster):  
    temp = copy.deepcopy(clusters_prev)  
    clusters_prev = copy.deepcopy(cluster)  
    clusters = update_k_means(points, temp, k)  
    plot_k_means_data_update(clusters_prev, clusters, k)  
    i+= 1  
# END ANSWER  
  
# Ground truth  
plt.scatter(points[:, 0], points[:, 1], c=y_train)  
plt.title('True labels')  
plt.show()
```



...



- Step 5: Now that we have a cluster, we would like to find out how good our clustering is. We will also do this using an unsupervised approach: without knowing the correct labels for each cluster, we *can* say something about the spread of each cluster.
- For this purpose you will compute the *sum of residual squares* (SRS) of the cluster. This is computed as follows: sum over the squared euclidean distances between each point and their corresponding centroid and divide by the total number of points. In math notation:

$$srs = \frac{1}{N} \sum_{i \in C} (p_i - c)^2$$

- With N the number of points, C the cluster containing point i and centroid c, and p_i the feature vector for point i.

```
# This function calculates the average sum of residual squares of the given cluster
def calculate_average_sum_rs(cluster):
    if len(cluster.data) == 0:
        return None
    # START ANSWER
    sum = 0
    for c in cluster.data:
        sum += distance(c, cluster.centroid())
    return sum / len(cluster.data)
    # END ANSWER
```

- Step 6: we will use this metric to try and automatically determine how many clusters we should use
 - Implement the `tune_k` method. This method should test out several values for k in range from 1 to 15. Then, for each k, run the k-means algorithm on `data/cluster.txt` with 10 update iterations. After the algorithm is done, calculate the average SRS of all clusters and print them.

```
random.seed(42)

# This function tries a number of k's for the update_k_means function (which is our kMean
# and calculates the SRS for each k.
# For each k, n_updates iterations of the update_k_means function are performed.
def tune_k(min_k, max_k, n_updates):
    assert 0 < min_k < max_k
    assert n_updates > 1
    srss = []

    # START ANSWER
    for i in range(max_k):
        clusters = []
        clusters_prev = []
        i = 0
        clusters_prev = update_k_means(points, clusters_prev, k)
        cluster = copy.deepcopy(clusters_prev)
        while i <= n_updates and any(c is not None and c.is_changed() for c in cluster):
            temp = copy.deepcopy(clusters_prev)
            clusters_prev = copy.deepcopy(cluster)
            clusters = update_k_means(points, temp, k)
            i += 1
        avg_sum = 0
        for c in clusters:
            avg_sum += calculate_average_sum_rs(c)
        avg_sum /= 3
        srss.append(avg_sum)
    # END ANSWER

plt.plot(list(range(min_k, (max_k+1))), srss, marker='o')
plt.xlabel('k')
```

```
plt.ylabel('Sum of Residual Squares')  
plt.grid(linestyle='-', linewidth=1)  
plt.show()
```

```
min_k=1  
max_k=15  
n_updates=10  
tune_k(min_k, max_k, n_updates)
```

