MACHINE LEARNING
AND
NEURAL NETWORKS

About The Instructor

- from Budapest, Hungary
- **BSc** in physics
- **MSc** in applied mathematics
- working as a software engineer
- special addiction to algorithms, artificial intelligence and quantitative finance

About The Course

- **Machine Learning**
  - linear regression, logistic regression, SVM, clustering algorithms ...
- **Neural Networks**
  - Feedforward-neural networks, backpropagation, gradient descent ...
- **Deep Learning**
  - deep networks, convolutional neural networks, recurrent neural networks ...
- **Reinforcement Learning**
  - Markov Decision Processes, value- and policy iteration, Q-learning

Types of Learning

SUPERVISED LEARNING

We have a dataset: with samples and labels as well

~ most of the machine learning techniques rely heavily on datasets

WE GIVE THE ALGORITHM THE RIGHT ANSWERS !!!

x	y	x **XOR** y
0	**0**	**0**
0	**1**	**1**
1	**0**	**1**
1	**1**	**0**

During the training procedure:

→ the input is the features **(x,y)**

→ the output is the **x XOR y** label

The aim is to make sure the prediction of the neural network is approximately the same as the label in the dataset

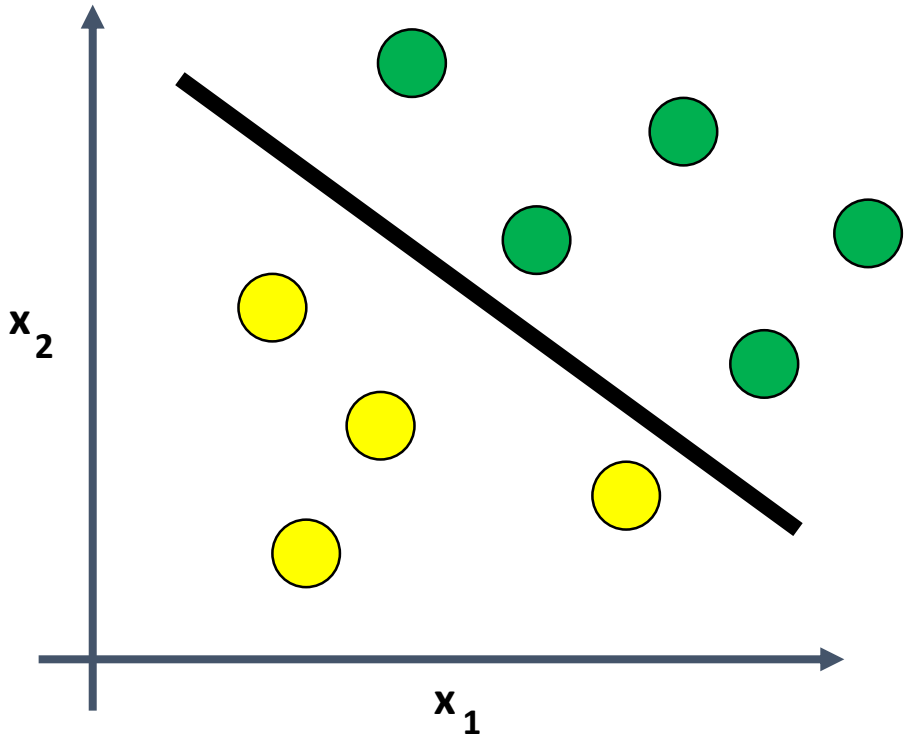
Types of Learning

SUPERVISED LEARNING

We have a dataset: with samples and labels as well

~ most of the machine learning techniques rely heavily on datasets

WE GIVE THE ALGORITHM THE RIGHT ANSWERS !!!



During the <u>training procedure</u>:

→ the algorithm will find out what is the difference between the two classes

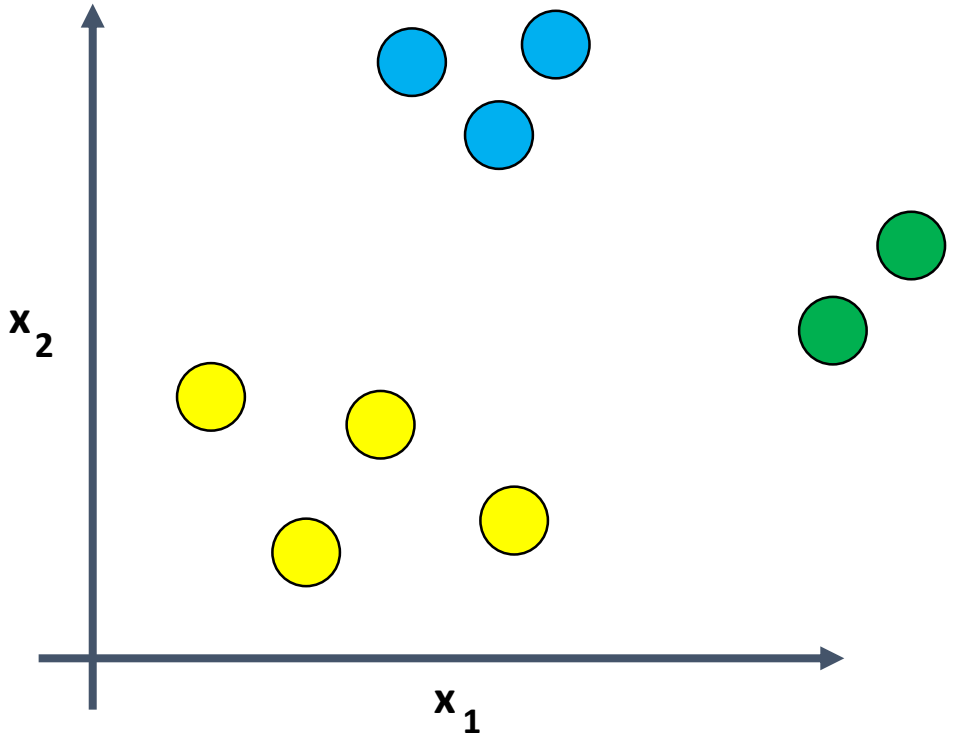
Types of Learning

UNSUPERVISED LEARNING

We have a dataset: samples without labels

~ the algorithm will find some patterns in this unlabeled dataset

For example: clustering algorithms



During the <u>training procedure</u>:

→ the algorithm will find some relevant features to make the classification

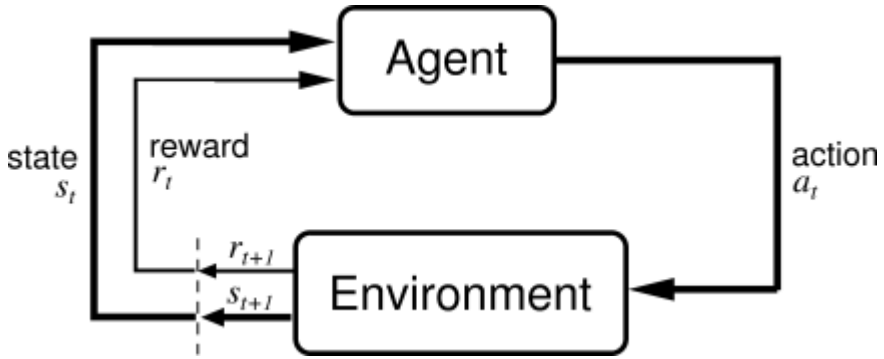
Types of Learning

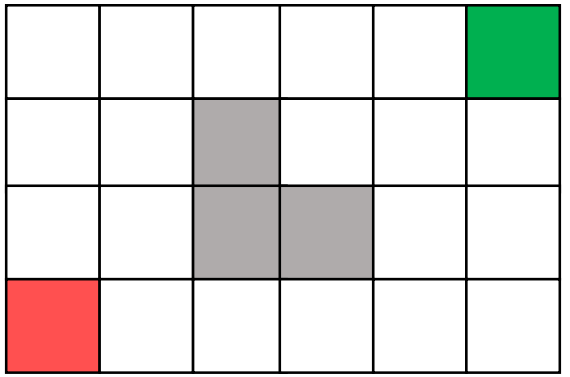
REINFORCEMENT LEARNING

We do not have a dataset at all !!!

~ the artificial intelligence agent will interacts with the environment (states) and figurest out what to do (actions)

UPON DOING ACTIONS THE AGENT RECEIVES A REWARD/PENALTY !!!





finding the shortest path: there are states (cells) and actions (we can visit the neighbor cells)

Types of Learning

REINFORCEMENT LEARNING

0		**X**
	0	
		X

→ reinforced learning can be used to learn playing tic-tac-toe game

→ environment: cells on the board

→ the agent (computer player) makes a move according to the board states

→ move: where to put the **X**

→ eventually the game will end: agent will receive reward (win) or receive penalty (lose)

The computer (agent) initially plays poorly: but after the training procedure (so playing a lot) it will be able to choose the right actions/moves !!!

Linear Regression

It is an approach for modelling the relationship between scalar dependent variable **y** and one or more explanatory variables **x**

SIMPLE LINEAR REGRESSION

→ single explanatory variable **x**

→ we want to approximate the price of houses if we know the sizes

MULTIPLE LINEAR REGRESSION

→ several explanatory variables **x**

→ we want to approximate the price of houses if we know the sizes, number of rooms...

We use linear predictor functions: this is why it is called **LINEAR** regression

Linear Regression

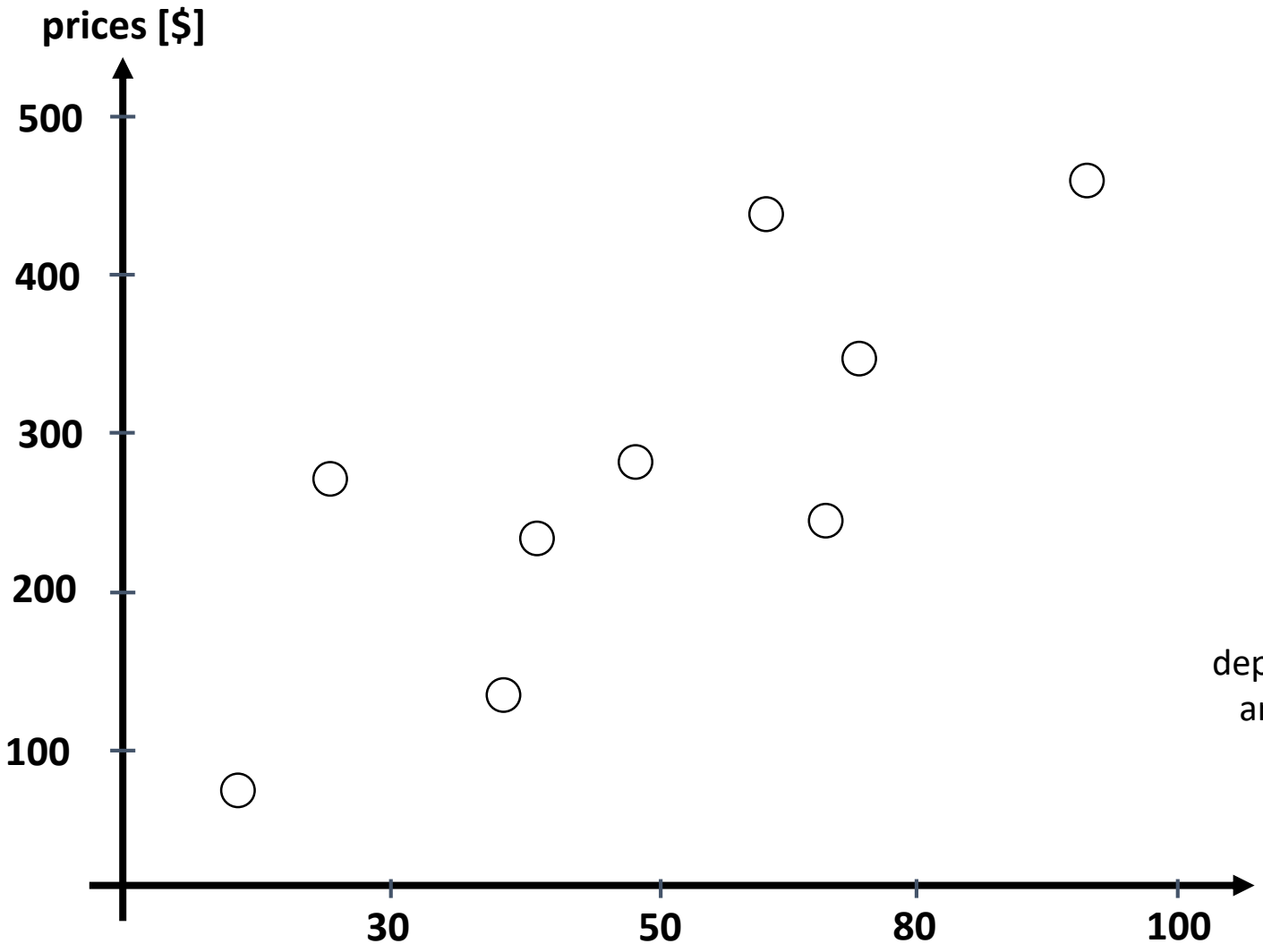
Linear regression is the first machine learning algorithms: as usual it needs a dataset

Dataset: **house_sales.csv**

Contains information about a home sold between May 2014 and May 2015 along with the price ~ multiple explanatory variables are present

```
id,date,price,bedrooms,bathrooms,sqft_living,sqft_lot,floors,waterfront,view,condition,grade,sqft_above,sqft_basement,yr_built,yr_renovated,zipcode,lat,long,sqft_living15,sqft_lot15
"7129300520","20141013T000000",221900,3,1,1180,5650,"1",0,0,3,7,1180,0,1955,0,"98178",47.5112,-122.257,1340,5650
"6414100192","20141209T000000",538000,3,2.25,2570,7242,"2",0,0,3,7,2170,400,1951,1991,"98125",47.721,-122.319,1690,7639
"5631500400","20150225T000000",180000,2,1,770,10000,"1",0,0,3,6,770,0,1933,0,"98028",47.7379,-122.233,2720,8062
"2487200875","20141209T000000",604000,4,3,1960,5000,"1",0,0,5,7,1050,910,1965,0,"98136",47.5208,-122.393,1360,5000
"1954400510","20150218T000000",510000,3,2,1680,8080,"1",0,0,3,8,1680,0,1987,0,"98074",47.6168,-122.045,1800,7503
"7237550310","20140512T000000",1.225e+006,4,4.5,5420,101930,"1",0,0,3,11,3890,1530,2001,0,"98053",47.6561,-122.005,4760,101930
"1321400060","20140627T000000",257500,3,2.25,1715,6819,"2",0,0,3,7,1715,0,1995,0,"98003",47.3097,-122.327,2238,6819
"2008000270","20150115T000000",291850,3,1.5,1060,9711,"1",0,0,3,7,1060,0,1963,0,"98198",47.4095,-122.315,1650,9711
"2414600126","20150415T000000",229500,3,1,1780,7470,"1",0,0,3,7,1050,730,1960,0,"98146",47.5123,-122.337,1780,8113
"3793500160","20150312T000000",323000,3,2.5,1890,6560,"2",0,0,3,7,1890,0,2003,0,"98038",47.3684,-122.031,2390,7570
"1736800520","20150403T000000",662500,3,2.5,3560,9796,"1",0,0,3,8,1860,1700,1965,0,"98007",47.6007,-122.145,2210,8925
"9212900260","20140527T000000",468000,2,1,1160,6000,"1",0,0,4,7,860,300,1942,0,"98115",47.69,-122.292,1330,6000
```

Linear Regression



What is the aim of Linear Regression?

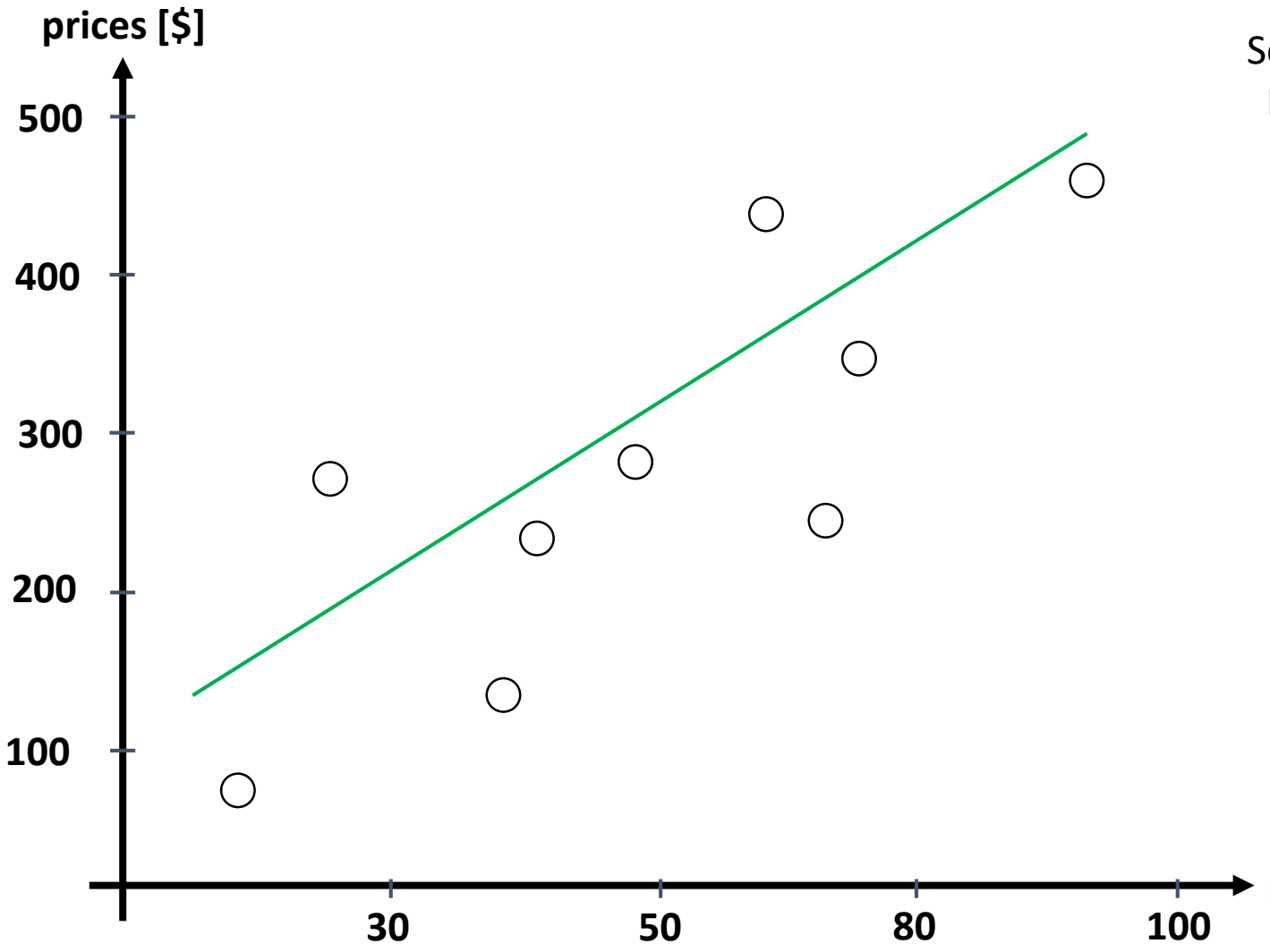
→ want to find some linear relationship between the features

independent variable we use to make predictions (size of house)

$$H(x) = b_0 + b_1 x$$

dependent variable, this is what we are trying to predict or estimate (price of the house)

Linear Regression



So we build a model based on the dataset
It is a linear model: so the result is a linear line

→ the model defines the relationship between the variables

→ we can make predictions with the trained model

What does it mean? If we have a new **x** feature (size of the house) we can get the **H(x)** price of the house accordingly

Linear Regression

we have a **dataset** (all machine learning algorithm needs one)

↓

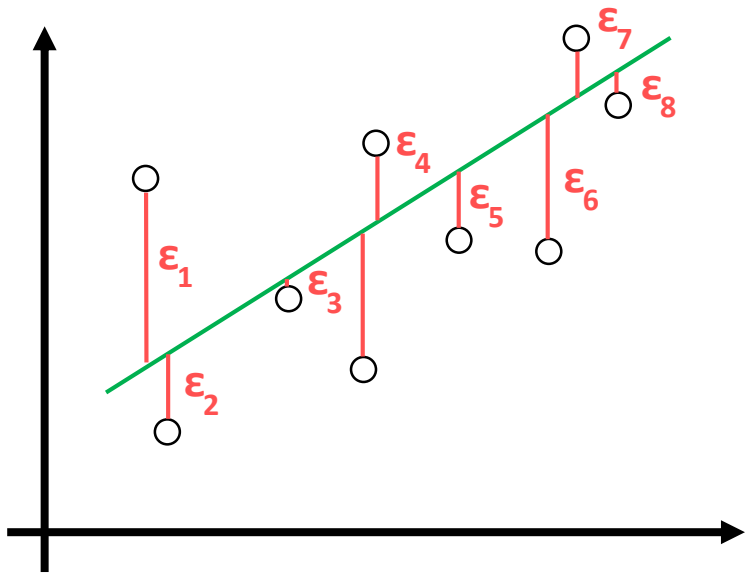
we **train** the algorithm: which means finding the linear relationship between the **H(x)** and **x** variables

$$H(x) = b_0 + b_1 x$$ **OPTIMIZATION !!!**

↓

after the training we have the **b** values which means we can make **predictions** with the model (for new datapoints)

Linear Regression



$$MSE = \sum_{i} \varepsilon_i^2$$

Mean Squared Error (MSE)

It is the difference between the **y** actual values present in the dataset (supervised learning) and the **H(x)** values predicted by the model

$$[\ H(x) - y\ ]^2$$ **„cost-function"**

→ if this term is small (small error): it means the model predictions are very close to the actual values **GOOD**

→ if this term is big (huge error): it means the model predictions differ from the actual values **BAD**

Linear Regression

OPTIMIZATION PROBLEM

This is why optimization algorithms are so important

~ no matter what problem we are dealing with, finally we have to use some optimization methods to solve it

this is the prediction from
our linear model (linear regression)

$$\min_{b} [H(x) - y]^2$$

we want to find the minimum by
tuning these parameters

this is the value we know from
the training data
// supervised learning !!!

Linear Regression

OPTIMIZATION PROBLEM

This is why optimization algorithms are so important

~ no matter what problem we are dealing with, finally we have to use some optimization methods to solve it

$$\min_{\underline{b}} [\boldsymbol{H}(\boldsymbol{x}) - \boldsymbol{y}]^2$$

Design Matrix Approach (linear algebra)

→ we can transform the problem into linear equations and use the standard method (using matrix operations)

$$\underline{b} = (X'X)^{-1}X'\,\underline{y}$$

https://onlinecourses.science.psu.edu/stat501/node/382

Gradient Descent

→ it is a first-order iterative optimization algorithm for finding the minimum of a function

Linear Regression

OPTIMIZATION PROBLEM

This is why optimization algorithms are so important

~ no matter what problem we are dealing with, finally we have to use some optimization methods to solve it

Design Matrix Approach (linear algebra)

→ for low-dimensional problems
it is the best approach

→ low-dimension means few features

→ if the matrix is huge: matrix operations are expensive in higher dimensions
Matrix inversion: $\mathbf{O(N^3)}$

USUALLY THIS IS THE CASE !!!

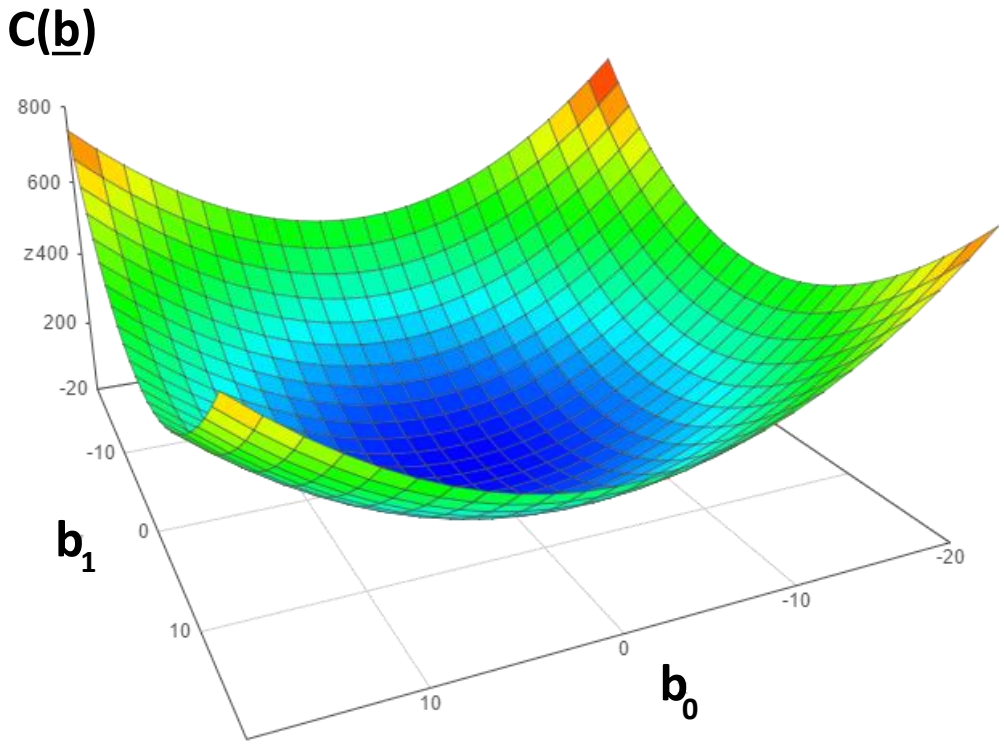
Gradient Descent

→ iterative approach

→ works fine in higher dimensions and usually this is the case

Gradient Descent

We have to know the partial derivative of the **C(b)** cost function
and go to the direction of the gradient
// gradient ~ partial derivative



$$\frac{\partial C(\mathbf{b})}{\partial b_0} \qquad \frac{\partial C(\mathbf{b})}{\partial b_1}$$

→ the **∇f(x)** gradient of a given **f(x)** function
is pointing in the direction of maximum

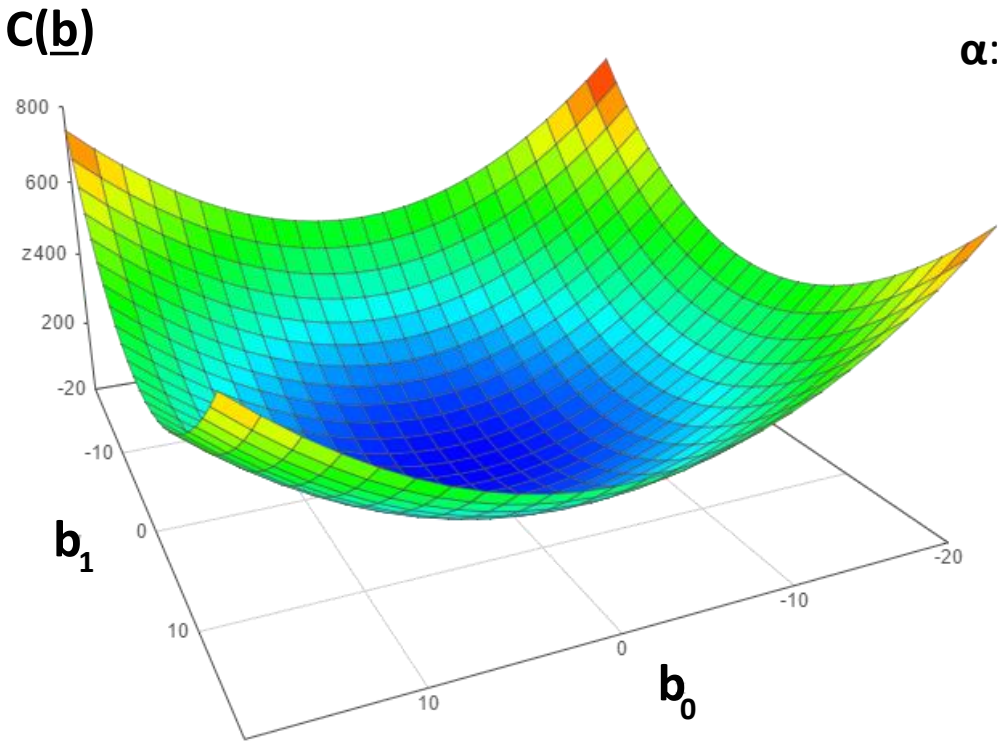
→ we are after the minimum: so we have to use
- ∇f(x) instead

make iterations {

$$b_0 \leftarrow b_0 - \alpha \frac{\partial C(\mathbf{b})}{\partial b_0} \qquad b_1 \leftarrow b_1 - \alpha \frac{\partial C(\mathbf{b})}{\partial b_1}$$

}

Gradient Descent



α: learning-rate

→ small learning-rate: the algorithm takes small steps towards the minimum
~ takes more time to converge

→ huge learning-rate: the algorithm takes big steps towards the minimum
~ algorithm is faster but not as accurate

Linear Regression - Parameters

The R^2 statistic is defined as follows:

$$R^2 = 1 - \frac{RSS}{TSS}$$ // the higher the better the model fits the data

→ measures the accuracy of the regression models
It is the square of the correlation coefficient **r**
~ so it measures how strong of a linear relationship is between two variables !!!

→ **RSS** - „residual sum of squares"
Measures the variability left unexplained after performing the regression $\sum_{i=1}^{n} [H(x) - y]^2$

→ **TSS** - „total sum of squares"
It measure the total variance in **y**

$$\frac{1}{n} \sum_{i=1}^{n} (y - \mu)^2$$

Logistic Regression

Linear Regression solves regression problem: the prediction is a value

For example: we have the features (size of flat, number of rooms...) and this model is able to predict the price

→ **Logistic Regression** solves classification problems

Usually we use this method for binary classification

For example: spam detection for emails, predicting if a customer will default on a loan ...

OUTCOME OF DEPENDENT VARIABLE IS DISCRETE !!!

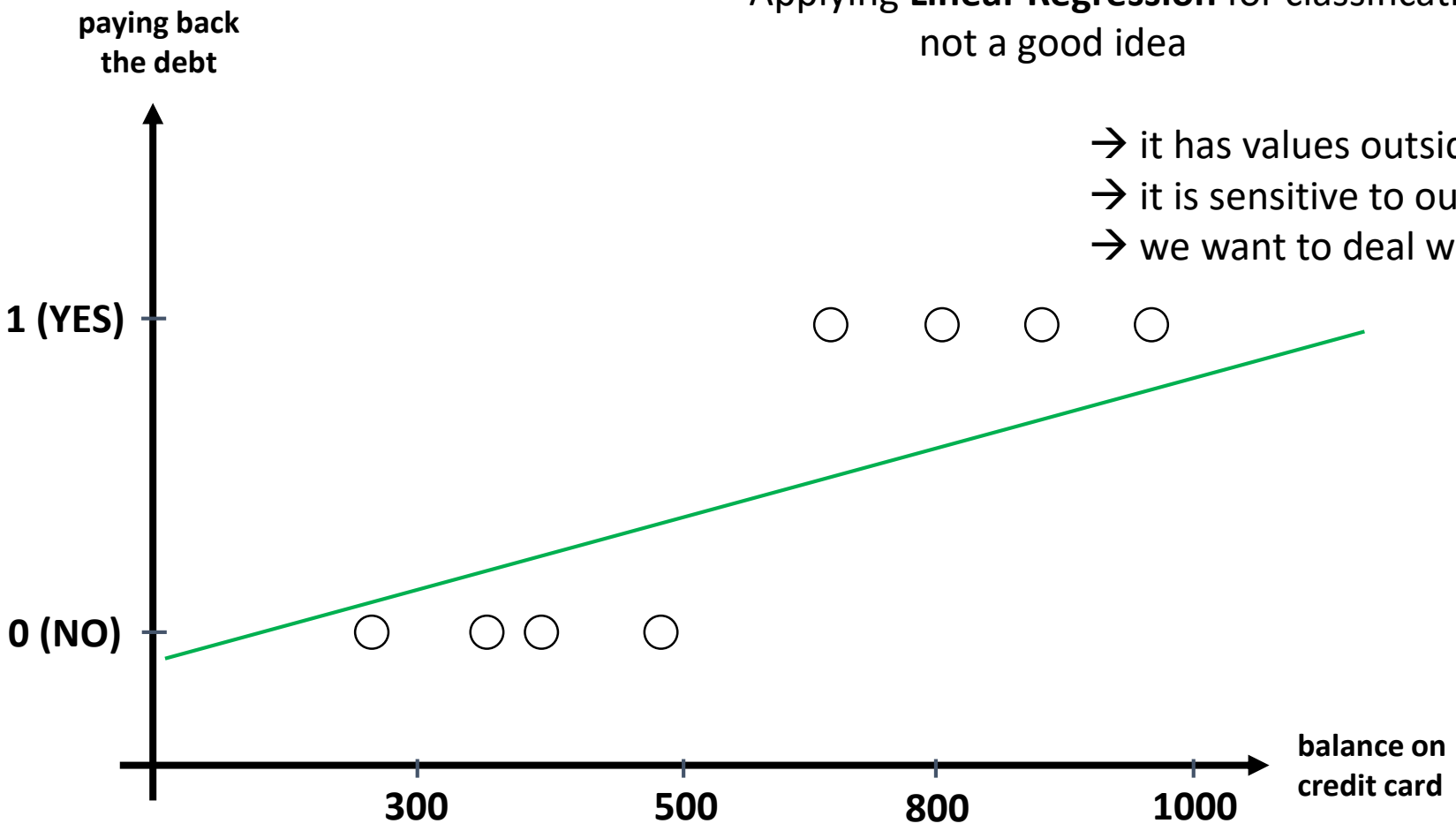
→ it assigns probabilities to given outcomes

So the output is a probability that the given input belongs to a certain class

Logistic Regression

Applying **Linear Regression** for classification problem is not a good idea

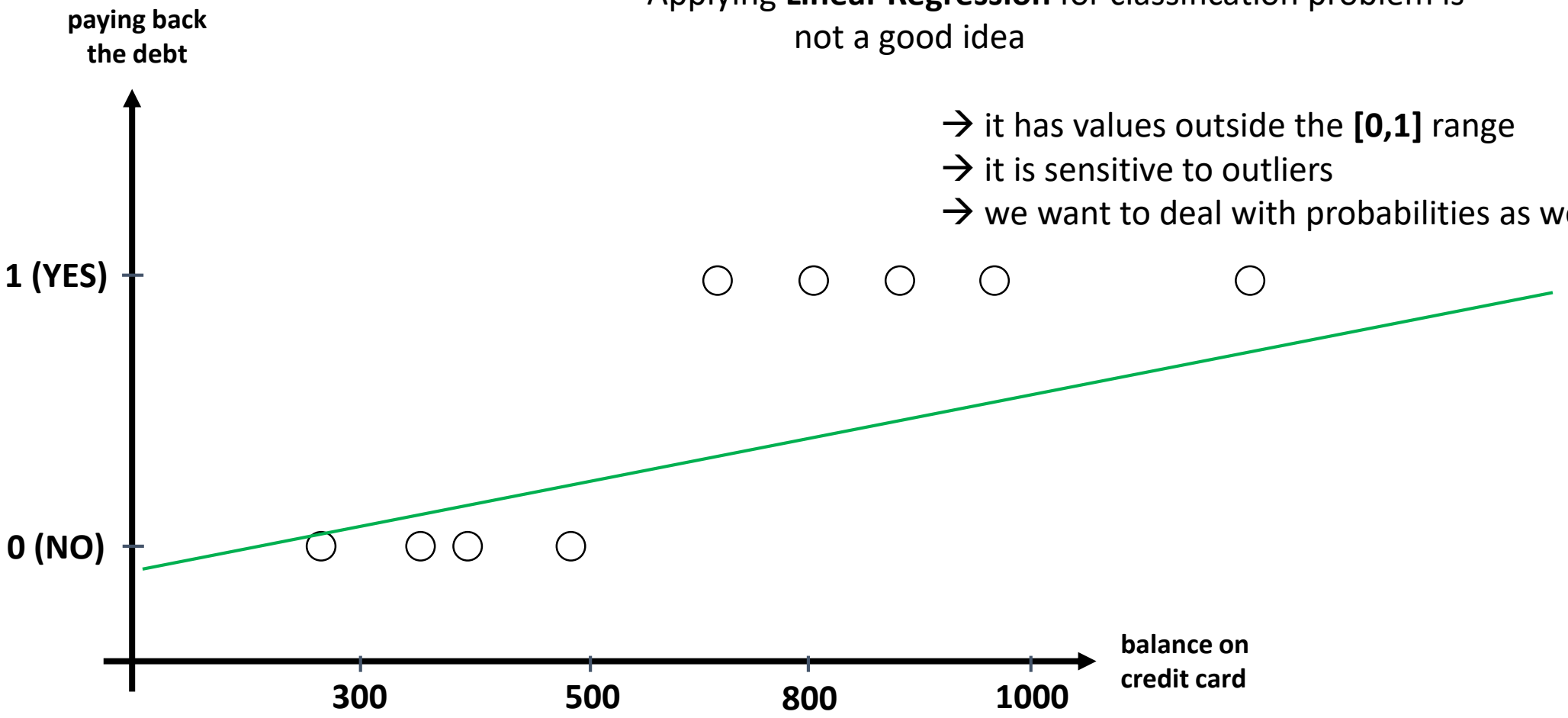
→ it has values outside the **[0,1]** range
→ it is sensitive to outliers
→ we want to deal with probabilities as well



Logistic Regression

Applying **Linear Regression** for classification problem is not a good idea

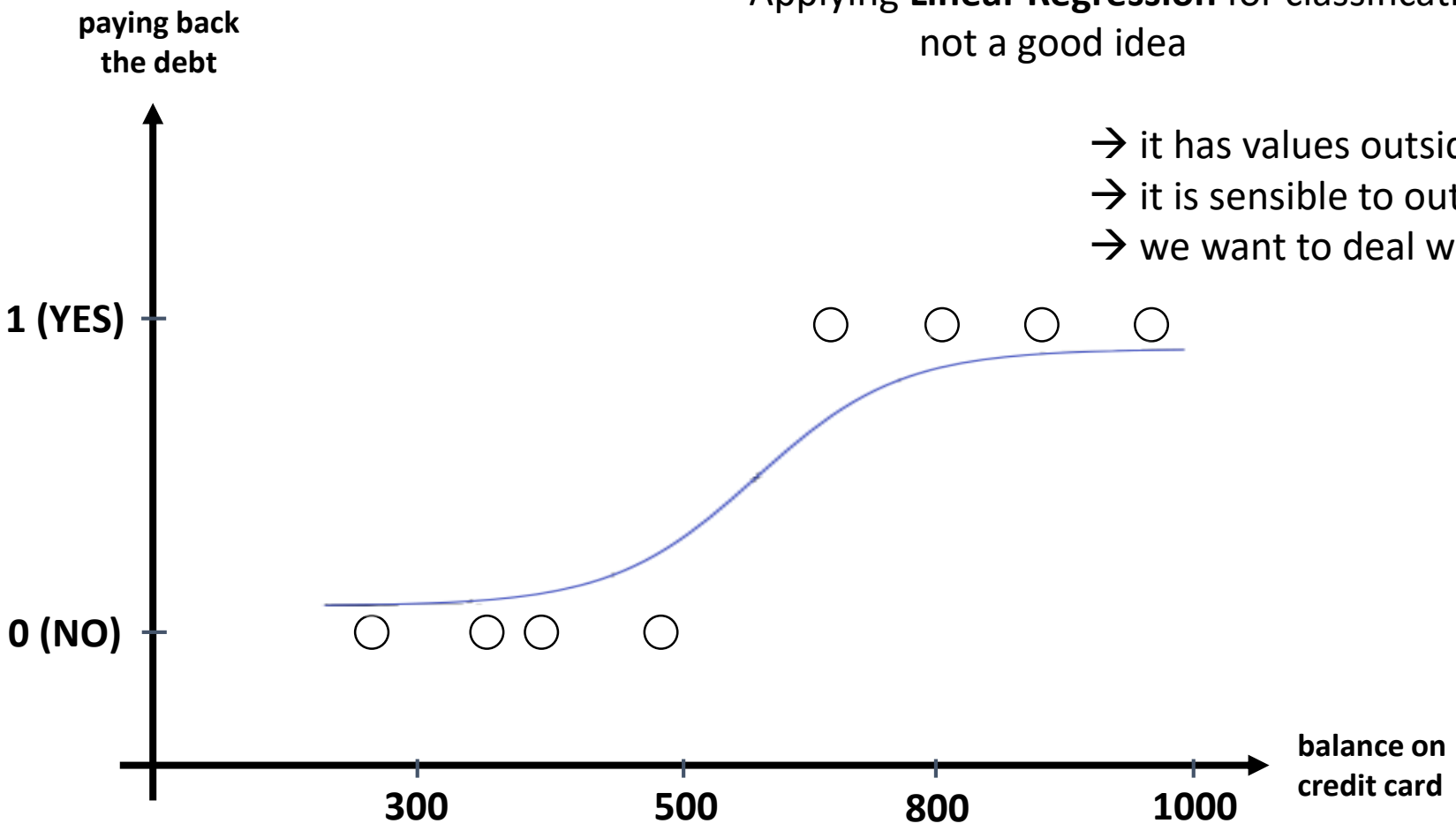
→ it has values outside the **[0,1]** range
→ it is sensitive to outliers
→ we want to deal with probabilities as well



Logistic Regression

Applying **Linear Regression** for classification problem is not a good idea

→ it has values outside the **[0,1]** range
→ it is sensible to outliers
→ we want to deal with probabilities as well



Logistic Regression

We have to deal with the **sigmoid-function**

$$f(x) = \frac{1}{1 + e^{-x}}$$

→ we are able to solve the problems we have discussed

→ it has a value between **[0,1]**

→ it can be interpreted as probability

Logistic Regression

We have to deal with the **sigmoid-function**

the probability of default given **x** balance

p(x) = P(y=1|x=balance)

$$p(x) = \frac{1}{1 + e^{-(b_0 + b_1 * x)}}$$

these are the model parameters $\mathbf{b_0}$ and $\mathbf{b_1}$

There are several ways to fit the model

→ gradient descent

→ maximum-likelihood method

Logistic Regression

We have to deal with the **sigmoid-function**

$$\ln\left(\frac{p(x)}{1 - p(x)}\right) = b_0 + b_1 x$$

„logit transformation"

→ the point of the logit transformation is to make it linear: so logistic regression is a linear regression on the logit transformation

→ how to fit the **b** parameters: with gradient descent and maximum-likelihood method

It is a ***generalized linear model***: not because the estimated probability of the response is linear but because the **logit** of the estimated probability response is a linear function of the parameters

Logistic Regression

SIMPLE LOGISTIC REGRESSION

→ we have just a single **x** parameter

→ For example: balance on credit card

$$p(x) = \frac{1}{1 + e^{-(b_0 + b_1 * x)}}$$

MULTINOMIAL LOGISTIC REGRESSION

→ we have multiple **x** parameters

→ For example: balance on credit card, age, gender, demographics, loan to income ratio ...

$$p(x) = \frac{1}{1 + e^{-(b_0 + b_1 * x_1 + \ldots + b_n x_n)}}$$

Usually we use logistic regression for binary classification: so with **2** output classes
For example: email is spam or not, client is sick or healthy ...

Logistic Regression – Maximum Likelihood

Maximum Likelihood Estimation is a method of estimating the parameters of a statistical model given observations, by finding the parameter values that maximize the likelihood of making the observations given the parameters

→ the method is based on a **likelihood-function**

log likelihood-function

$$l(\beta,x) = \ln L(\beta,x)$$

likelihood-function

The aim is the same as we have seen for Linear Regression: we are after the optimal **β** values that maximize the likelihood-function

$$\beta \text{ such that } \{ \arg\max_{\beta} l(\beta,x) \}$$

Logistic Regression – Maximum Likelihood

Maximum Likelihood Estimation is a method of estimating the parameters of a statistical model given observations, by finding the parameter values that maximize the likelihood of making the observations given the parameters

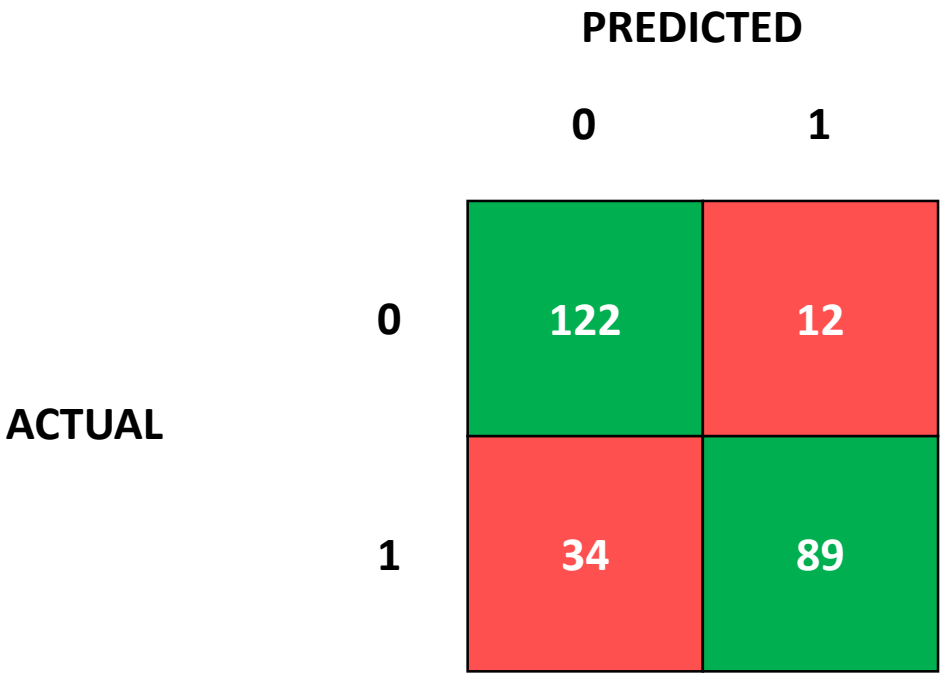
$$L(\underline{\beta}) = \prod_{i=1}^{n} p(y_i, x_i)^{y_i} (1 - p(y_i, x_i))^{1-y_i}$$

likelihood-function for logistic regression

$$p(y_i, \underline{x}_i) = \frac{1}{1 + e^{-(\underline{x}_i' * \beta)}}$$

→ this estimate is usually obtained by using the iterative algorithm **Newton-Raphson** method

Confusion Matrix



Describes the performance of a classification model

→ diagonal elements: the correct classifications

→ off-diagonals: incorrect predictions

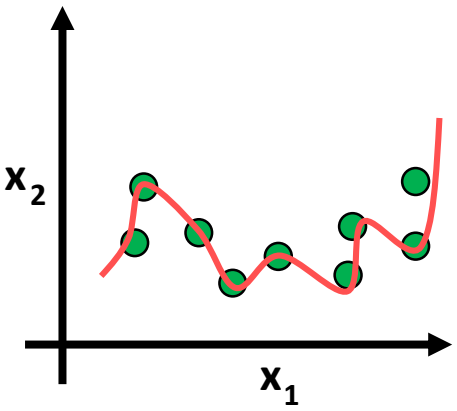
Cross Validation

TRAINING DATASET	TEST DATASET
~ **70%** of the original dataset	~ **30%** of the original dataset

We fit the model to the **training dataset**: then we test the model on the **test dataset**
~ we only have information how the model performs to our in-sample data
but we would like to see the accuracy when dealing with new data

OVERFITTING: model has trained „too well" on the training dataset
→ it means it is very accurate on the training dataset but yields
poor results on the test set (due to too complex models)

→ the model learns the „noise" instead of the actual relationships
between the variables in the data
(of course this noise is not present in the test set...)



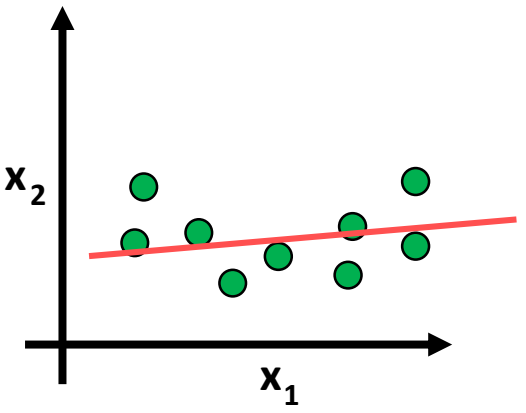
Cross Validation

TRAINING DATASET	TEST DATASET
~ **70%** of the original dataset	~ **30%** of the original dataset

We fit the model to the **training dataset**: then we test the model on the **test dataset**
~ we only have information how the model preforms to our in-sample data
but we would like to see the accuracy when dealing with new data

UNDERFITTING: model has not been fitted well to the training dataset
→ misses the trends in the training dataset

→ this is usually the case when we use too simple models for the problem



Cross Validation

K-Folds Cross Validation

→ helps to avoid underfitting as well as overfitting
(help to avoid overfitting more than underfitting)

→ the aim is to be able to generalize the model to new datasets with same accuracy
→ we use all the data for training !!!

TRAINING DATASET

Let's split the data into **k** folds (for example **k=5**)

Cross Validation

K-Folds Cross Validation

→ helps to avoid underfitting as well as overfitting
(help to avoid overfitting more than underfitting)

→ the aim is to be able to generalize the model to new datasets with same accuracy
→ we use all the data for training !!!

Let's split the data into **k** folds (for example **k=5**)

→ we run **k** separate learning experiments
k-1 folds for training and **1** fold for the test set

→ we average the results from this **k** experiments

Cross Validation

K-Folds Cross Validation

→ helps to avoid underfitting as well as overfitting
(help to avoid overfitting more than underfitting)

→ the aim is to be able to generalize the model to new datasets with same accuracy
→ we use all the data for training !!!

Let's split the data into **k** folds (for example **k=5**)

→ we run **k** separate learning experiments
k-1 folds for training and **1** fold for the test set

→ we average the results from this **k** experiments

ADVANTAGE: all observations are used for both training and validation
+ each observations are used for validation exactly once

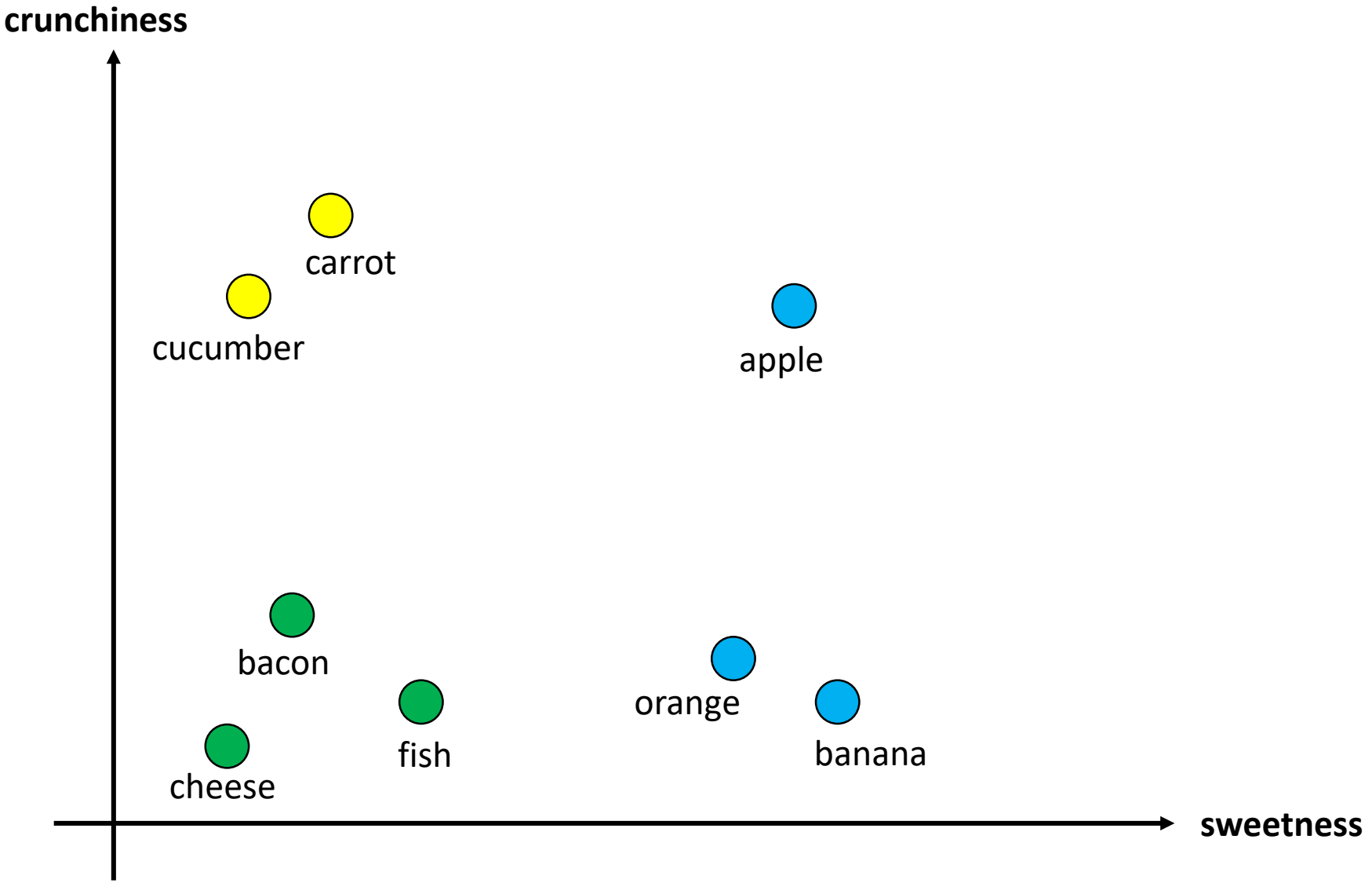
K-Nearest Neighbor Classifier

- K-Nearest Neighbors (**kNN**) classifiers can classify examples by assigning them the class of the most similar labeled examples
- very simple **BUT** extremely powerful algorithm !!!
- **kNN** is well suited for classification tasks where the relationship between the features are very complex and hard to understand
- we have a training dataset → examples that are classified into several categories
- we have a new example (with the same number of features as the training data) → **kNN** algorithm identifies **k** elements in the training dataset that are the „nearest” in similarity
- the unlabeled test example is assigned to the class of the majority of the **k** nearest neighbors

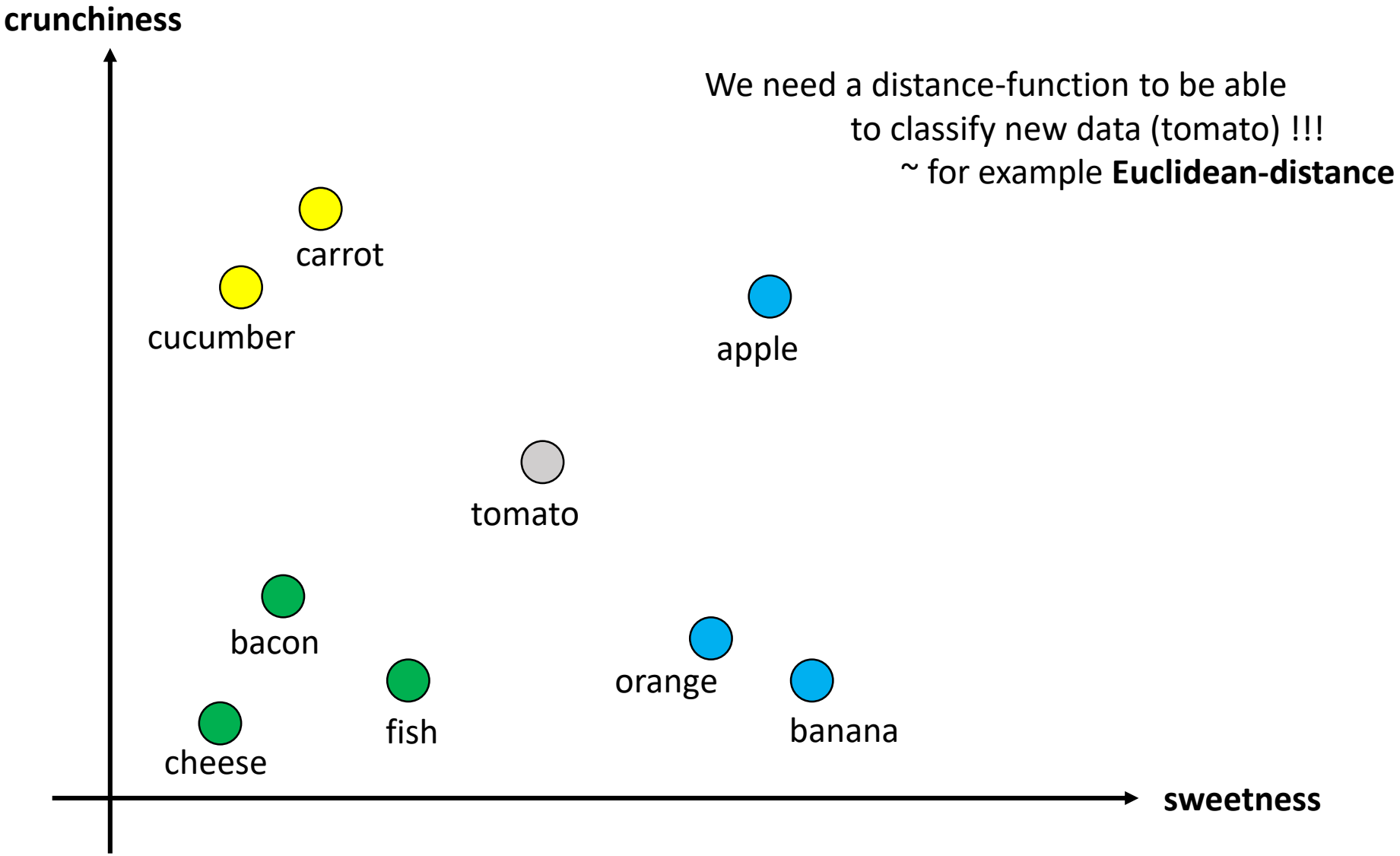
K-Nearest Neighbor Classifier

ingredients	sweetness	crunchiness	type
apple	10	9	fruit
bacon	1	4	protein
banana	10	1	fruit
carrot	7	10	vegetable
cheese	1	1	protein
tomato	6	4	???

K-Nearest Neighbor Classifier



K-Nearest Neighbor Classifier



We need a distance-function to be able
to classify new data (tomato) !!!
~ for example **Euclidean-distance**

K-Nearest Neighbor Classifier

- lazy learners does not learn anything !!!
- we just store the training data: training is very fast (because there is no training at all) BUT making the prediction is rather slow (calculating the distances)
- **WE DO NOT BUILD A MODEL !!!**
- this is a non-parametric learning: no parameters are to be learned about the data

  Linear Regression / Logistic Regression: have to learn the **β** parameters either

  with gradient descent or with maximum likelihood method

K-Nearest Neighbor Classifier

$$\text{dist}(x,y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \cdots + (x_n - y_n)^2}$$

K-Nearest Neighbor Classifier

ingredients	sweetness	crunchiness	type
apple	10	9	fruit
bacon	1	4	protein
banana	10	1	fruit
carrot	7	10	vegetable
cheese	1	1	protein
tomato	6	4	???

dist(tomato,carrot) = $\sqrt{(6-7)^2+(4-10)^2}$ = 6.083

K-Nearest Neighbor Classifier

dist(tomato,carrot) = $\sqrt{(6-7)^2+(4-10)^2}$ = 6.083

dist(tomato,apple) = $\sqrt{(6-10)^2+(4-9)^2}$ = 6.403

dist(tomato,bacon) = $\sqrt{(6-1)^2+(4-4)^2}$ = 5

dist(tomato,banana) = $\sqrt{(6-10)^2+(4-1)^2}$ = 5

dist(tomato,cheese) = $\sqrt{(6-1)^2+(4-1)^2}$ = 5.83

k=1 we consider the smallest distance: bacon and banana

k=2 we consider the **2** smallest distances: bacon and banana
50%-50% that tomato is a fruit or a protein

k=3 we consider the **3** smallest distances: bacon, banana and cheese
So tomato appears to be a protein !!!

K-Nearest Neighbor Classifier

HOW TO CHOOSE THE OPTIMAL K VALUE

- deciding how many neighbors to use for **kNN** → determines how well the model will generalize and work on other datasets

- **k** is small → noisy data or outliers have a huge impact on our classifier ... this is called „underfitting”

- **k** is large → the classifier has the tendency to predict the majority class regardless of which neighbors are nearest ... this is called „overfitting”

Normalization

- features are usually transformed into a range before the **kNN** algorithm is applied
- **WHY?**
- the distance formula depends on how features are measured
- if certain features have much larger values than others → the distance measurements will be strongly dominated by the larger values
- we have to rescale the various features such that each one contributes relatively equally to the distance formula

1.) **min-max normalization**

2.) **z-transformation**

Normalization

MIN-MAX NORMALIZATION

→ this process transforms a feature such that all of its values fall in a range between **0** and **1**

→ normalized feature values can be interpreted as indicating how far, from **0%** to **100%**, the original value fall along the range between the original minima and maxima

$$X_{new} = \frac{X - \min(X)}{\max(X) - \min(X)}$$

Normalization

Z-SCORE NORMALIZATION (STANDARDIZATION)

→ it is a way of normalizing the dataset as well: the algorithm uses mean and standard deviation to do so

$$X_{new} = \frac{X - \text{mean}(X)}{StandardDeviation(X)}$$

For **Principle Component Analysis** we prefer using z-score normalization

~ but for image processing: **pixel intensities** have to be normalized to fit within a certain range + neural networks requires data that on a **0-1** scale

Naive Bayes Classifier

- very efficient supervised learning algorithm
- it scales well even in high dimensions !!!
- it is able to compete with **SVM** or **random forest** classifiers
- it is able to make good predictions even when the training data is relatively small

Why is it naive?

The naive assumption is that every pair

of features are independent

Naive Bayes Classifier

Naive means there is a strong **independence assumptions** between the given features

For example: a fruit can be considered to be an apple if it is red, rounded and about 8cm in diameter

~ **Naive Bayes Classifier** considers each of these features contribute independently to the probability that this fruit is an apple
(do not care about the correlation between color, roundness and diameter)

Naive Bayes Classifier

ABSTRACT MATHEMATICAL APPROACH

x are the features

$P(C_k | x_1 x_2 \dots x_n)$ this model relies heavily on conditional probability

C is the possible outcome of **k** classes

$$P(C_k | x_1 x_2 \dots x_n) = \frac{p(C_k)\, p(\underline{x} | C_k)}{p(\underline{x})}$$

this is the **Bayes theorem**, we can decompose the conditional probability

the same for all c_k values

$$P(x_1 x_2 \dots x_n | C_k) = P(x_1 | C_k)\, P(x_2 | C_k) \dots P(x_n | C_k)$$

INDEPENDENT FEATURES !!!

$$P(C_k | x_1 x_2 \dots x_n) \sim p(C_k) \prod_{i=1}^{n} p(x_i | C_k)$$

Naive Bayes Classifier

ABSTRACT MATHEMATICAL APPROACH

If we assume the features are independent we can come up with a powerful probability-based classification algorithm

→ we have to choose the C_k class with the highest probability

$$C = \arg\max_{c_k} \; p(C_k) \prod_{i=1}^{n} p(x_i | \; C_k)$$

„Naive Bayes Classifier"

Naive Bayes Classifier

ADVANTAGES

→ relatively simple to understand
→ it can be trained on small datasets as well
→ it is a fast approach
→ it is not sensitive to irrelevant features

DISADVANTAGES

→ it assumes every feature is independent: of course it is not always true !!!

Naive Bayes Classifier

Why is it so powerful for **text classification**?

The two assumptions of Naive Bayes Classifier:

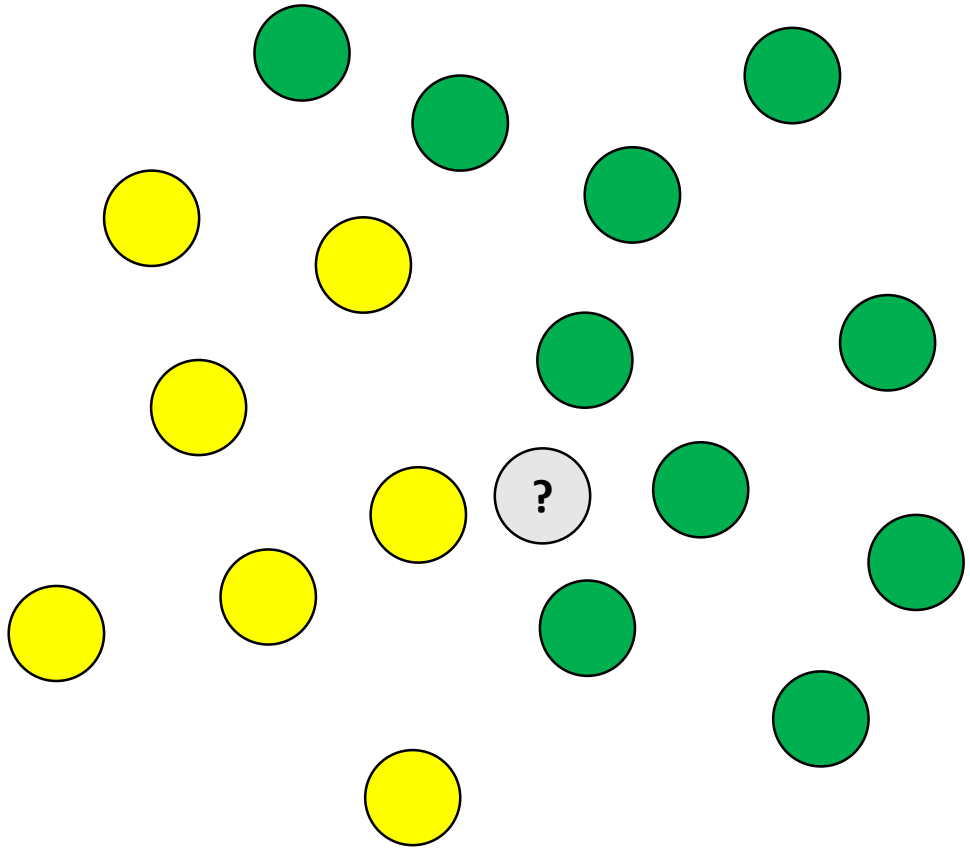
→ the probability of occurrence of any word given the class label is independent of the probability of occurrence of any other word given that label

→ the probability of occurrence of a word in a document is independent of the location of that word within the document

It is the same assumption of bag-of-words model: documents are just a bunch of words thrown together

THESE ASSUMPTIONS ARE TRUE FOR TEXT CLASSIFICATION !!!

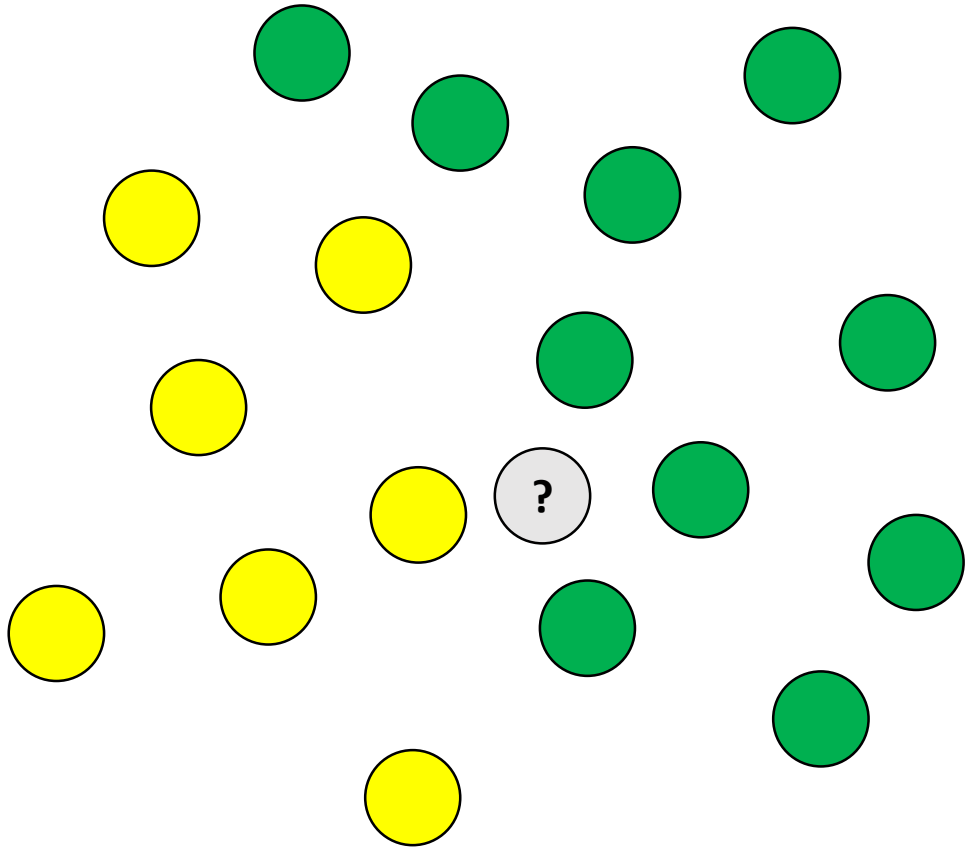
Naive Bayes Classifier



Naive Bayes Classifier

P(yellow) = $\frac{7}{17}$

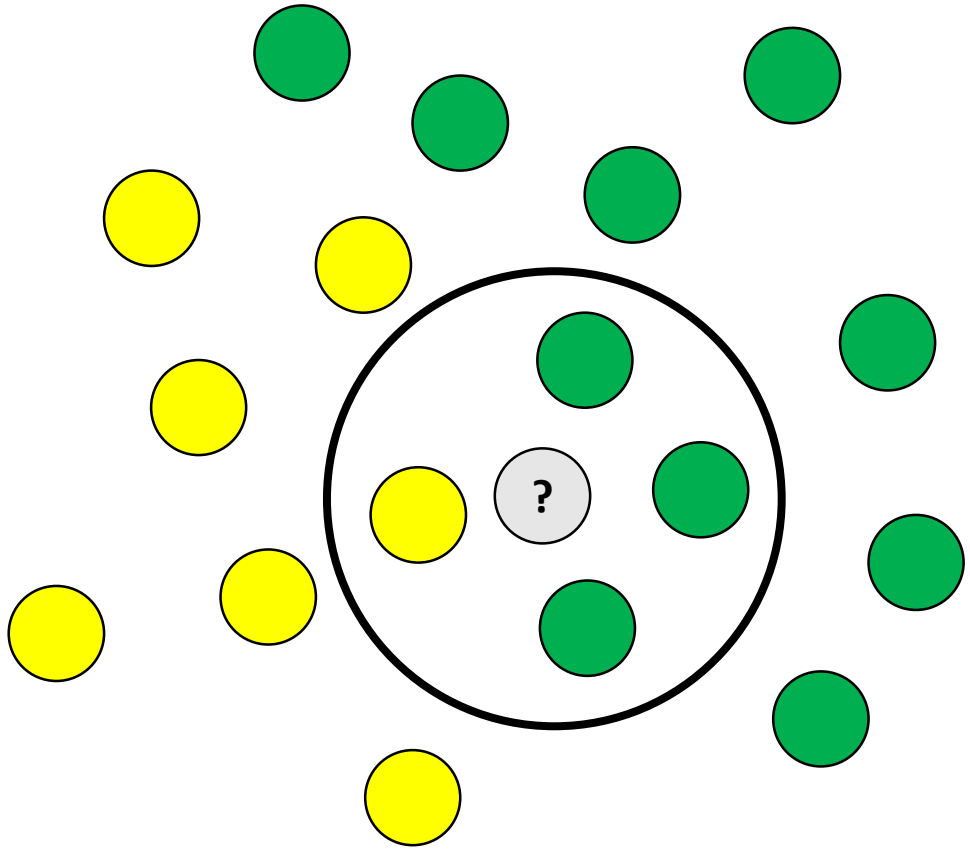
P(green) = $\frac{10}{17}$



Naive Bayes Classifier

P(yellow) = $\frac{7}{17}$

P(green) = $\frac{10}{17}$



P’(? | green) = $\frac{3}{10}$

P’(? | yellow) = $\frac{1}{7}$

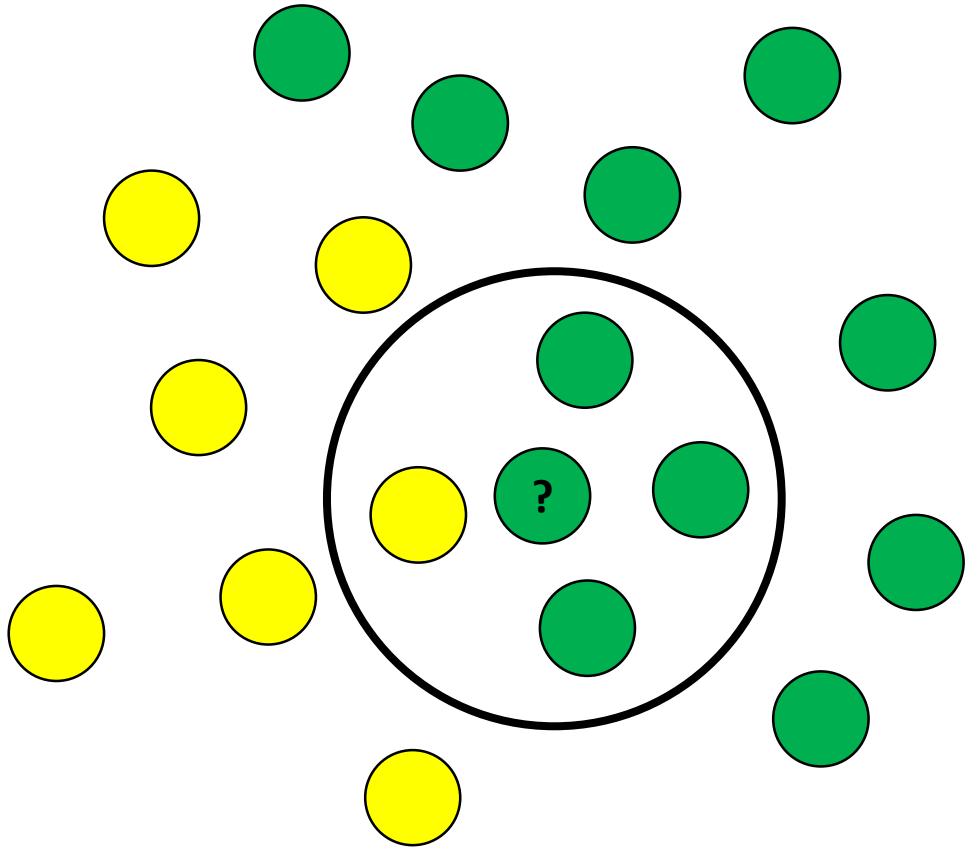
posterior probability P’’(? is green) = P(green) * P’(? | green) = $\frac{10}{17} * \frac{3}{10} = \frac{30}{170}$

posterior probability P’’(? is yellow) = P(yellow) * P’(? | yellow) = $\frac{7}{17} * \frac{1}{7} = \frac{7}{119}$

Naive Bayes Classifier

P(yellow) = $\frac{7}{17}$

P(green) = $\frac{10}{17}$



P’(? | green) = $\frac{3}{10}$

P’(? | yellow) = $\frac{1}{7}$

posterior probability P’’(? is green) = P(green) * P’(? | green) = $\frac{10}{17} * \frac{3}{10} = \frac{30}{170}$

posterior probability P’’(? is yellow) = P(yellow) * P’(? | yellow) = $\frac{7}{17} * \frac{1}{7} = \frac{7}{119}$

Decision Trees

Decision Tree is a type of supervised learning approaches: mostly used in classification problems but it can be used for regression as well

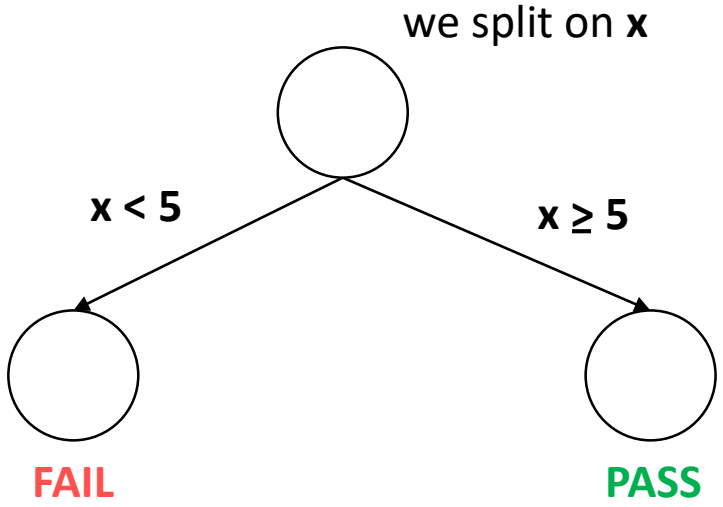
~ works fine for both categorical variables and continuous input as well

→ very similar to search trees

→ split the data/population into tow or more homogeneous sets based on significant splitter in input variables

For example: we have a single feature **x**, the number of hours a student spent with studying

Want to predict the **y** probability of passing the exam



Categorical Variable Decision Tree:

→ categorical target variable such as yes/no or fail/pass

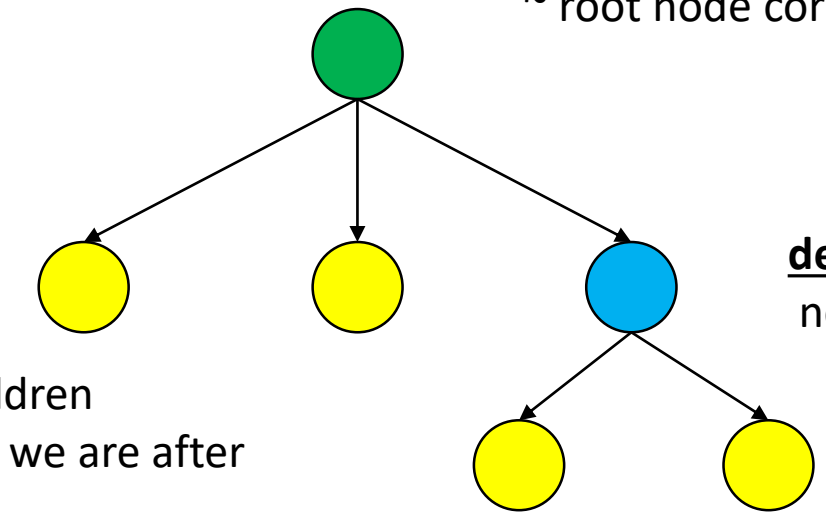
Continuous Variable Decision Tree:

→ continuous target variable: such as for regression

Decision Trees

root node: represent the entire dataset/population and this further gets divided into several subsets

~ root node corresponds to the best predictor



decision node: the algorithm splits the node into sub-nodes based on given feature in the dataset

leaf nodes: with no children

~ the values are what we are after

HOW TO DECIDE WHAT NODES (FEATURES) ARE IMPORTANT?
WHAT SHOULD BE THE ROOT NODE?

Decision Trees

Decision Tree classifier accuracy depends heavily on splits
Hot to construct the tree? What are the nodes?

There are several algorithms for this problem:

1.) Gini Index Approach

2.) Calculating the **information entropy (ID3** algorithm or **C4.5** approach**)**

3.) algorithm based on variance reduction

Decision Trees

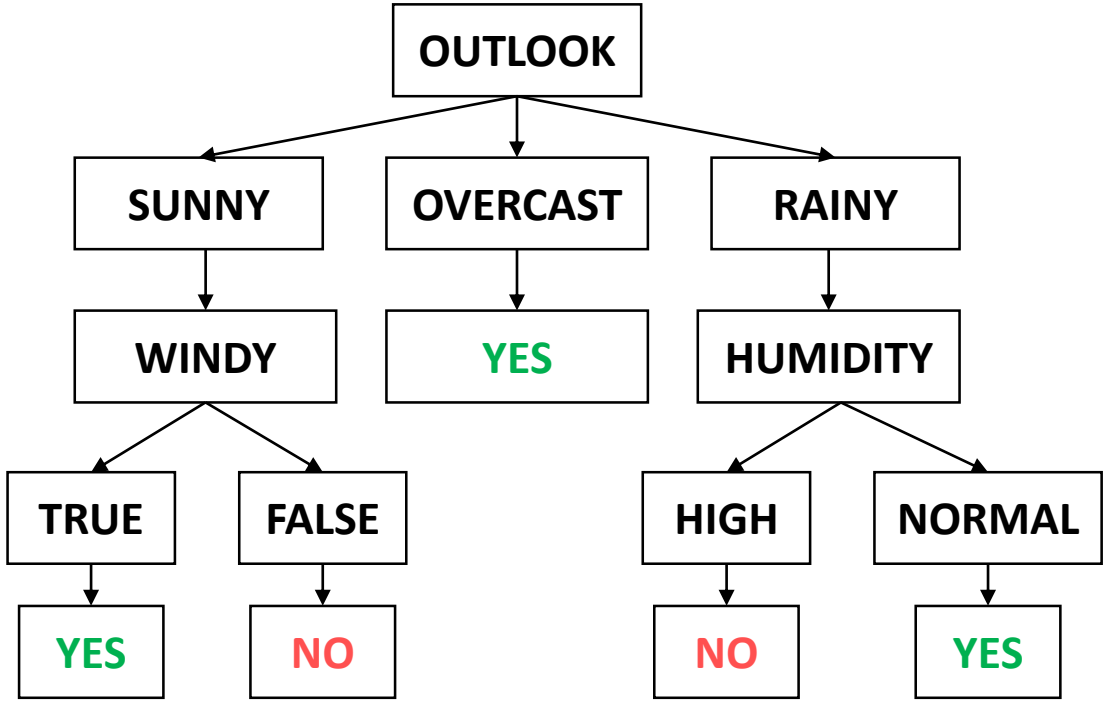
outlook	temperature	humidity	wind	play
sunny	hot	high	false	no
sunny	hot	high	true	no
overcast	hot	high	false	yes
rainy	mild	high	false	yes
rainy	cold	normal	false	yes
rainy	cold	normal	true	no
overcast	cold	normal	true	yes
sunny	mild	high	false	no
sunny	cold	normal	false	yes
rainy	mild	normal	false	yes
sunny	mild	normal	true	yes
overcast	mild	high	true	yes
overcast	hot	normal	false	yes
rainy	mild	high	true	no

We are dealing with categorical variables (predictors) or features: **outlook, temperature, humidity, wind**

~ we want to predict whether to play golf or not (play is he **target variable**)

Decision Trees

outlook	temperature	humidity	wind	play
sunny	hot	high	false	no
sunny	hot	high	true	no
overcast	hot	high	false	yes
rainy	mild	high	false	yes
rainy	cold	normal	false	yes
rainy	cold	normal	true	no
overcast	cold	normal	true	yes
sunny	mild	high	false	no
sunny	cold	normal	false	yes
rainy	mild	normal	false	yes
sunny	mild	normal	true	yes
overcast	mild	high	true	yes
overcast	hot	normal	false	yes
rainy	mild	high	true	no



Decision Trees

Usually **ID3** algorithm is used to build the decision tree:
~ it is a top-down greedy search of possible branches

→ it uses **entropy** and **information gain** to build the tree

The **H(X)** Shannon-entropy of a dicrete random variable **X** with possible values $x_1 x_2 \dots x_n$ and probability mass function **P(X)** is defined as:

$$H(X) = -\sum_{i=1}^{n} P(x_i) \log_2 P(x_i)$$

Example: https://en.wikipedia.org/wiki/Entropy_(information_theory)

For completely homogoeneous dataset (all TRUE or all FALSE values): entropy is **0**
If the dataset is equally divided (same amount of TRUEs and FALSEs): entropy is **1**

A BRANCH WITH ENTROPY MORE THAN 1 NEEDS SPLITING !!!
+ root node has the maximum information gain (entropy reduction)
+ leaf nodes have entropy 0

Decision Trees

outlook	temperature	humidity	wind	play
sunny	hot	high	false	no
sunny	hot	high	true	no
overcast	hot	high	false	yes
rainy	mild	high	false	yes
rainy	cold	normal	false	yes
rainy	cold	normal	true	no
overcast	cold	normal	true	yes
sunny	mild	high	false	no
sunny	cold	normal	false	yes
rainy	mild	normal	false	yes
sunny	mild	normal	true	yes
overcast	mild	high	true	yes
overcast	hot	normal	false	yes
rainy	mild	high	true	no

PLAYING GOLF

→ **9** times **YES**

→ **5** times **NO**

We just have to use the Shannon-entropy formula to calculate the **H(x)** values

H(PlayingGolf) = H(9,5) =

$$= -(0.64 \log_2 0.64) - (0.36 \log_2 0.36) = 0.94$$

Decision Trees

outlook	temperature	humidity	wind	play
sunny	hot	high	false	no
sunny	hot	high	true	no
overcast	hot	high	false	yes
rainy	mild	high	false	yes
rainy	cold	normal	false	yes
rainy	cold	normal	true	no
overcast	cold	normal	true	yes
sunny	mild	high	false	no
sunny	cold	normal	false	yes
rainy	mild	normal	false	yes
sunny	mild	normal	true	yes
overcast	mild	high	true	yes
overcast	hot	normal	false	yes
rainy	mild	high	true	no

$$E(T,X) = \sum_{x} P(x)\, E(x)$$

We have to calculate the entropy with respect to a given predictor/feature in order to be able to calculate information gain

		PLAY GOLF YES	PLAY GOLF NO
OUTLOOK	sunny	3	2
	overcast	4	0
	rainy	2	3

E(PlayGolf,Outlook) = P(sunny)E(3,2) + P(overcast)E(4,0) + P(rainy)E(2,3)

$$\frac{5}{14}\,0.971 + \frac{4}{14}\,0 + \frac{5}{14}\,0.971 = 0.693$$

Decision Trees

outlook	temperature	humidity	wind	play
sunny	hot	high	false	no
sunny	hot	high	true	no
overcast	hot	high	false	yes
rainy	mild	high	false	yes
rainy	cold	normal	false	yes
rainy	cold	normal	true	no
overcast	cold	normal	true	yes
sunny	mild	high	false	no
sunny	cold	normal	false	yes
rainy	mild	normal	false	yes
sunny	mild	normal	true	yes
overcast	mild	high	true	yes
overcast	hot	normal	false	yes
rainy	mild	high	true	no

Information gain: the decrease in entropy after a dataset is split on an attribute/feature

→ feature/attribute with the highest information gain will be the root node in the tree

Information Gain = H(PlayGolf) - E(PlayGolf,Outlook) =
= 0.64 – 0.693 = 0.247

		PLAY GOLF	
		YES	**NO**
OUTLOOK	sunny	**3**	**2**
	overcast	**4**	**0**
	rainy	**2**	**3**

Decision Trees

outlook	temperature	humidity	wind	play
sunny	hot	high	false	no
sunny	hot	high	true	no
overcast	hot	high	false	yes
rainy	mild	high	false	yes
rainy	cold	normal	false	yes
rainy	cold	normal	true	no
overcast	cold	normal	true	yes
sunny	mild	high	false	no
sunny	cold	normal	false	yes
rainy	mild	normal	false	yes
sunny	mild	normal	true	yes
overcast	mild	high	true	yes
overcast	hot	normal	false	yes
rainy	mild	high	true	no

Information gain: the decrease in entropy after a dataset is split on an attribute/feature

→ feature/attribute with the highest information gain will be the root node in the tree

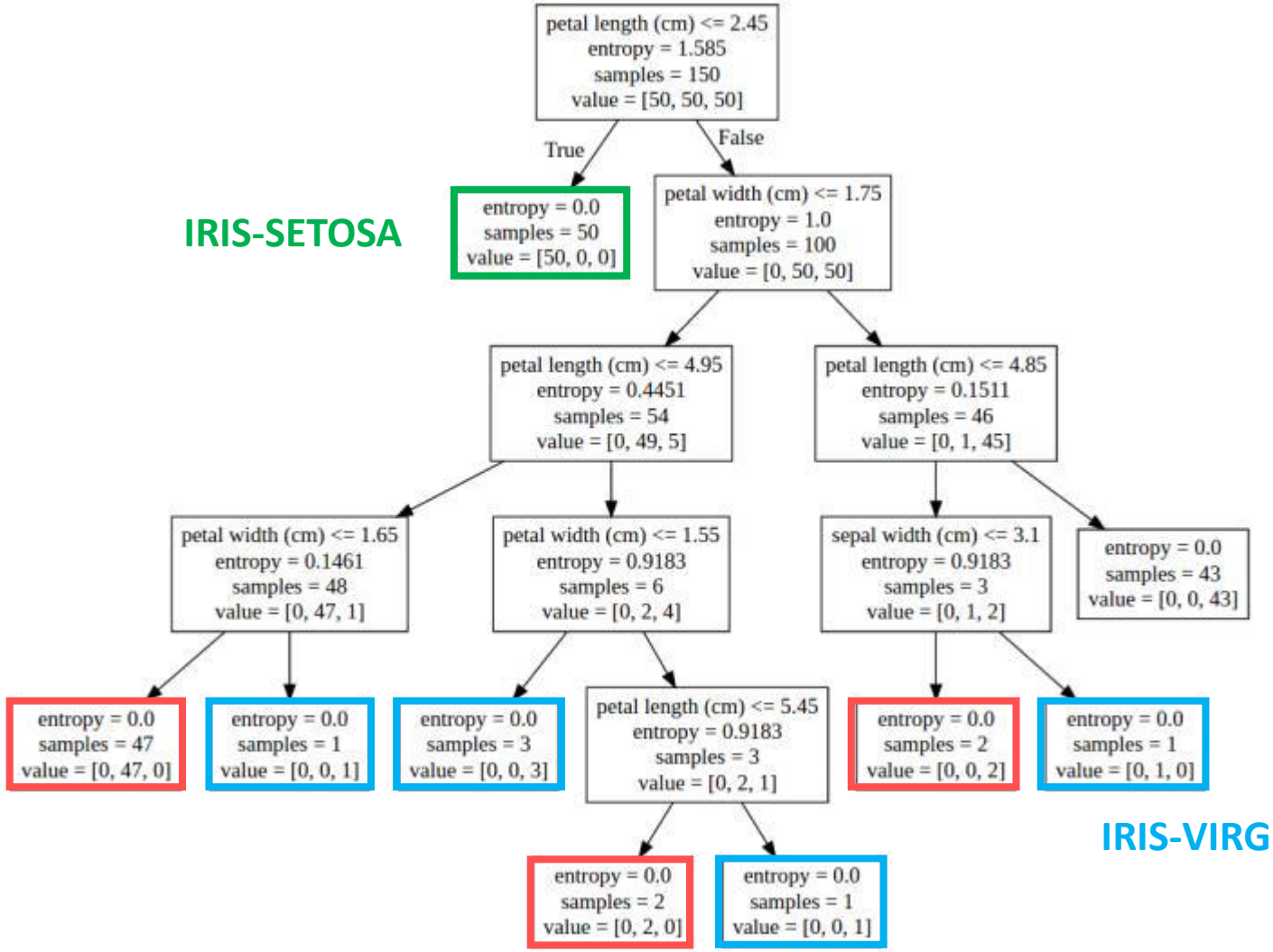
Information Gain (outlook) = 0.247

Information Gain (temperature) = 0.029

Information Gain (humidity) = 0.152

Information Gain (wind) = 0.048

Decision Trees



IRIS-SETOSA

IRIS-VERSICOLOR

IRIS-VIRGINICA

Decision Trees

Most significant problem: every split it makes at each node is optimized for the dataset it is fit to

~ this splitting process will rarely generalize well to other data !!!

Decision Trees

ADVANTAGES

→ easy to understand and interpret

→ it is one of the best approaches to identify most significant variables and the relationships between the variables

→ no need for data preprocessing

Decision trees are not influenced by outliers for example

→ can handle numerical variables as well as categorical variables

Categorical variables: **YES/NO** or **SUNNY/RAINY**

DISADVANTAGES

→ have the tendency to overfit

~ we can solve it by pruning for example

→ decision trees can be unstable because small variations in the data might result in a completely different tree being generated

~ generating the optimal tree is **NP-complete**: heuristic solutions are used (greedy approach)

Gini Index Approach

CART (Classification and Regression Tree) algorithm uses Gini indexes to decide how to make the splits

→ using Gini Index Approach is a bit better than calculating the information gain and entropy

THE ALGORITHMS ARE APPROXIMATELY THE SAME !!!

→ when dealing with entropy: we have to calculate logarithmic functions, which are computationally expensive
~ thats why we prefer Gini Index Approach

Gini Index Approach

$$G(X) = 1 - \sum_{i=1}^{n} P(x_i)^2$$

This is the formula we have to use in order to calculate the Gini-index of a **X** random variable

$$E(s,t) = G(t) - P(left)\, G(left\_t) - P(right)\, G(right\_t)$$

As we have seen for Information Gain, here as well we have to calculate the Gini-index concerning a given split

WE SPLIT THE FEATURE WITH THE LOWEST GINI-INDEX

Gini Index Approach

outlook	temperature	humidity	wind	play
sunny	hot	high	false	no
sunny	hot	high	true	no
overcast	hot	high	false	yes
rainy	mild	high	false	yes
rainy	cold	normal	false	yes
rainy	cold	normal	true	no
overcast	cold	normal	true	yes
sunny	mild	high	false	no
sunny	cold	normal	false	yes
rainy	mild	normal	false	yes
sunny	mild	normal	true	yes
overcast	mild	high	true	yes
overcast	hot	normal	false	yes
rainy	mild	high	true	no

PLAYING GOLF

→ **9** times **YES**

→ **5** times **NO**

We just have to use the Gini-index formula to calculate the **G(x)** values

G(PlayingGolf) = G(9,5) =

$$= 1 - 0.64^2 - 0.36^2 = 0.46$$

Gini Index Approach

$$E(s,t) = G(t) - P(\text{left})\, G(\text{left\_t}) - P(\text{right})\, G(\text{right\_t})$$

Gini-index for input node **t**

proportion of observation in the left node after splitting + Gini-index of the left node after splitting

proportion of observation in the right node after splitting + Gini-index of the right node after splitting

		DEFAULT YES	DEFAULT NO
GENDER	male	**3**	**2**
	female	**4**	**0**

G(t) = 1 - $(\frac{7}{9})^2$ **-** $(\frac{2}{9})^2$ **= 0.3432**

G(male) = 1 - $(\frac{3}{5})^2$ **-** $(\frac{2}{5})^2$ **= 0.48**

G(female) = 1 - $(\frac{4}{4})^2$ **-** $(\frac{0}{4})^2$ **= 0**

E(s,t) = 0.34 – $\frac{5}{9}$ **0.48 -** $\frac{4}{9}$ **0 = 0.07**

Pruning and Bagging

What is the aim when dealing with machine learning algorithms? We want to choose a model that capture the relatinships within the training dataset
+ **generalizes well to unseen data** (test set)

GENERALLY IT IS IMPOSSIBLE TO ACHIEVE BOTH OF THEM AT THE SAME TIME

~ this is called the **bias-variance trade-off**

Pruning and Bagging

bias: error from misclassifications in the learning algorithm

High bias → the algorithm misses the relevant relationships between features and target outputs (*underfitting*)

ERROR DUE TO MODEL MISMATCH

variance: error from sensitivity to small changes in the training set

High variance → can cause overfitting (the algorithm models the noise)

VARIATION DUE TO TRAINING SAMPLE AND RANDOMIZATION

bias / variance trade-off

~ we are not able to optimize both bias and variance at the same time

low bias → high variance

low variance → high bias

Pruning and Bagging

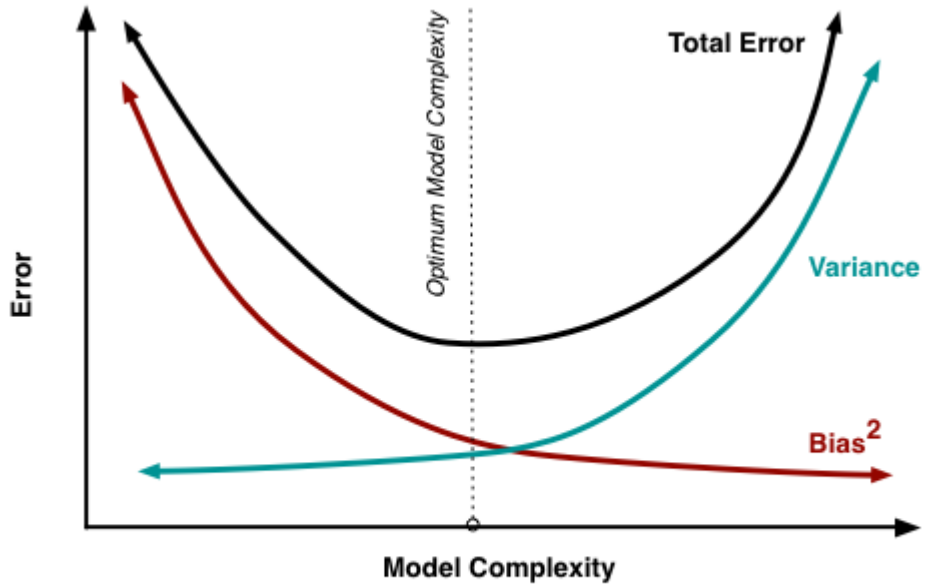


Fig. 6 Bias and variance contributing to total error.

http://scott.fortmann-roe.com/docs/BiasVariance.html

Pruning

- usually decision trees are likely to overfit the data leading to poor test performance

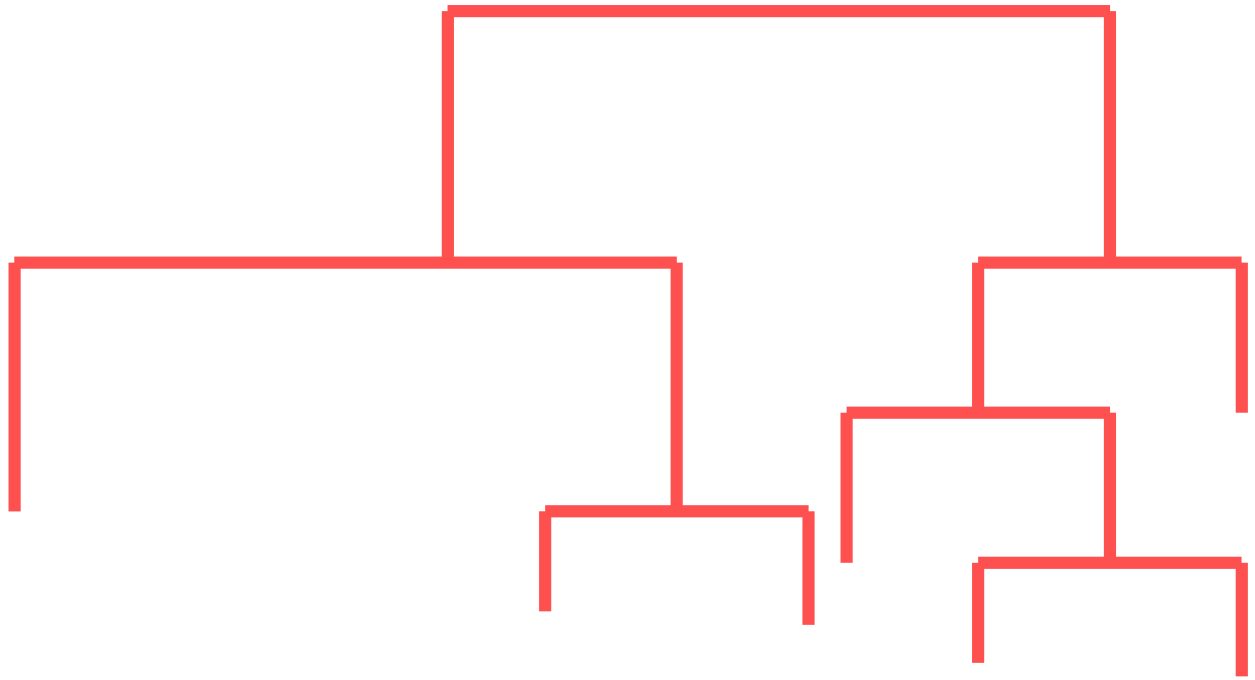
  + trees are unstable classifiers: if you perturb the data a little the tree might significantly change

    (low bias but high variance model !!!)

- smaller tree + fewer splits → better predictor at the cost of a little extra bias
- **better solution**: grow a large tree and then prune it back to a smaller subtree

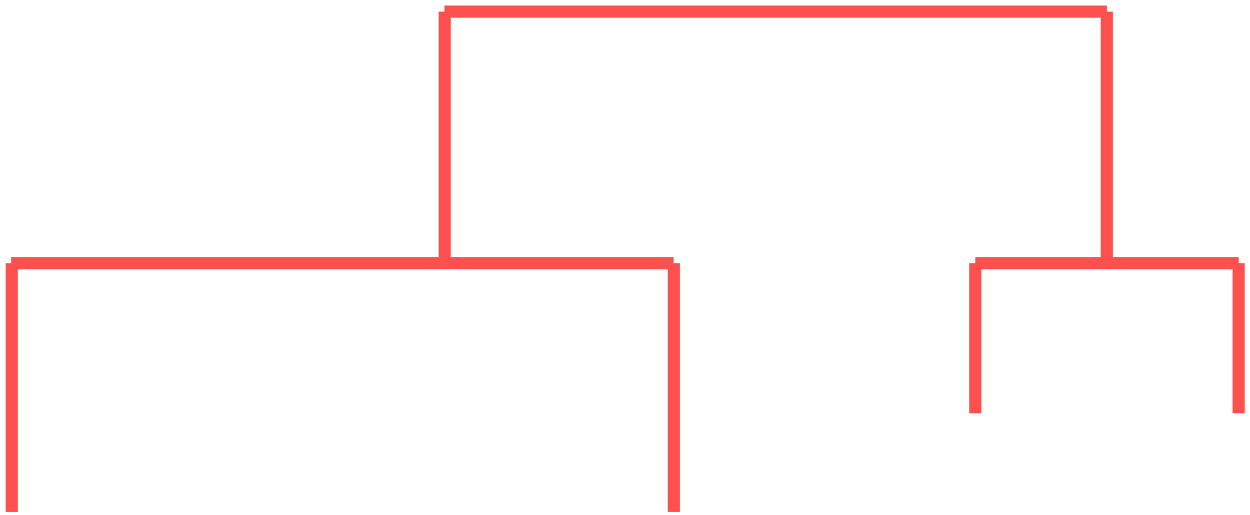
  „weakest link pruning”

Pruning



BEFORE PRUNING

Pruning



AFTER PRUNING

Bagging

BOOTSTRAP AGGREGATION

A rather counter-intuitive theory: a weak learner is not able to make good predictions

→ weak learner is just a bit better than random guess or coin flip
For example: decision trees with depth **1**

→ combining weak learners can prove to be an extremely powerful classifier !!!

„wisdom of the crowd"

(Black-Scholes model is approximately the same: two risky positions taken together can effectively eliminate risk itself)

Bagging

BOOTSTRAP AGGREGATION

→ reduces the variance of a learning algorithm

→ if we have a **X** set of **n** independent varibles $\mathbf{x_1}, \mathbf{x_2}, \dots, \mathbf{x_n}$ each with variance **V** then the variance of the mean **X** (the mean of the $\mathbf{x_1}, \mathbf{x_2} \dots \mathbf{x_n}$ variables) is $\frac{\mathbf{V}}{\mathbf{n}}$

WE CAN REDUCE THE VARIANCE BY AVERAGING A SET OF OBSERVATIONS !!!

→ good idea: have multiple training sets and construct a decision tree (without pruning) on every single training set !!!

→ **PROBLEM**: we do not have several training sets

Bagging

BOOTSTRAP AGGREGATION

→ we should take repeated samples from the single data set + construct trees + average all the predictions in the end

~ all the trees are fully grown unpruned decision trees

THIS IS CALLED BAGGING !!!

→ pruning: variance decreases but we have some bias ... here we can reduce the variance without extra bias

Regression problem: we take the average
Classification problem: we take the majority vote

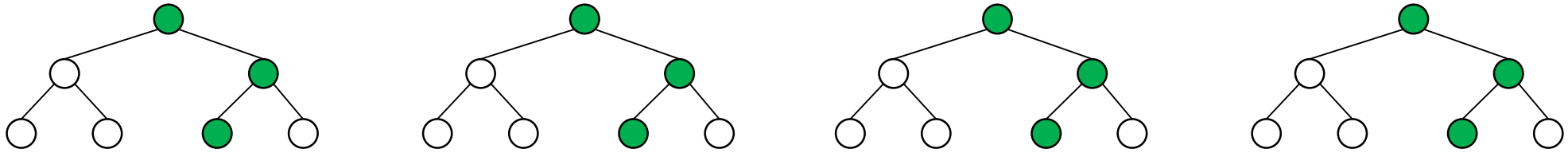
Bagging

BOOTSTRAP AGGREGATION

One problem with bagging: the constructed trees are highly **correlated**

Why do correlation occur?

Because every dataset has a strong predictor/feature. All the bagged trees tend to make the same splits because they all share the same features !!!
~ because of this all of these trees look very similar



TREE #1 **TREE #2** **TREE #3** **TREE #4**

CORRELATED TREES BECAUSE WE USE ALL THE FEATURES

Random Forest Classifier

→ better than bagging: this algorithm **decorrelates** the single decision trees that has been constructed

→ this reduces the variance even more when averaging the trees

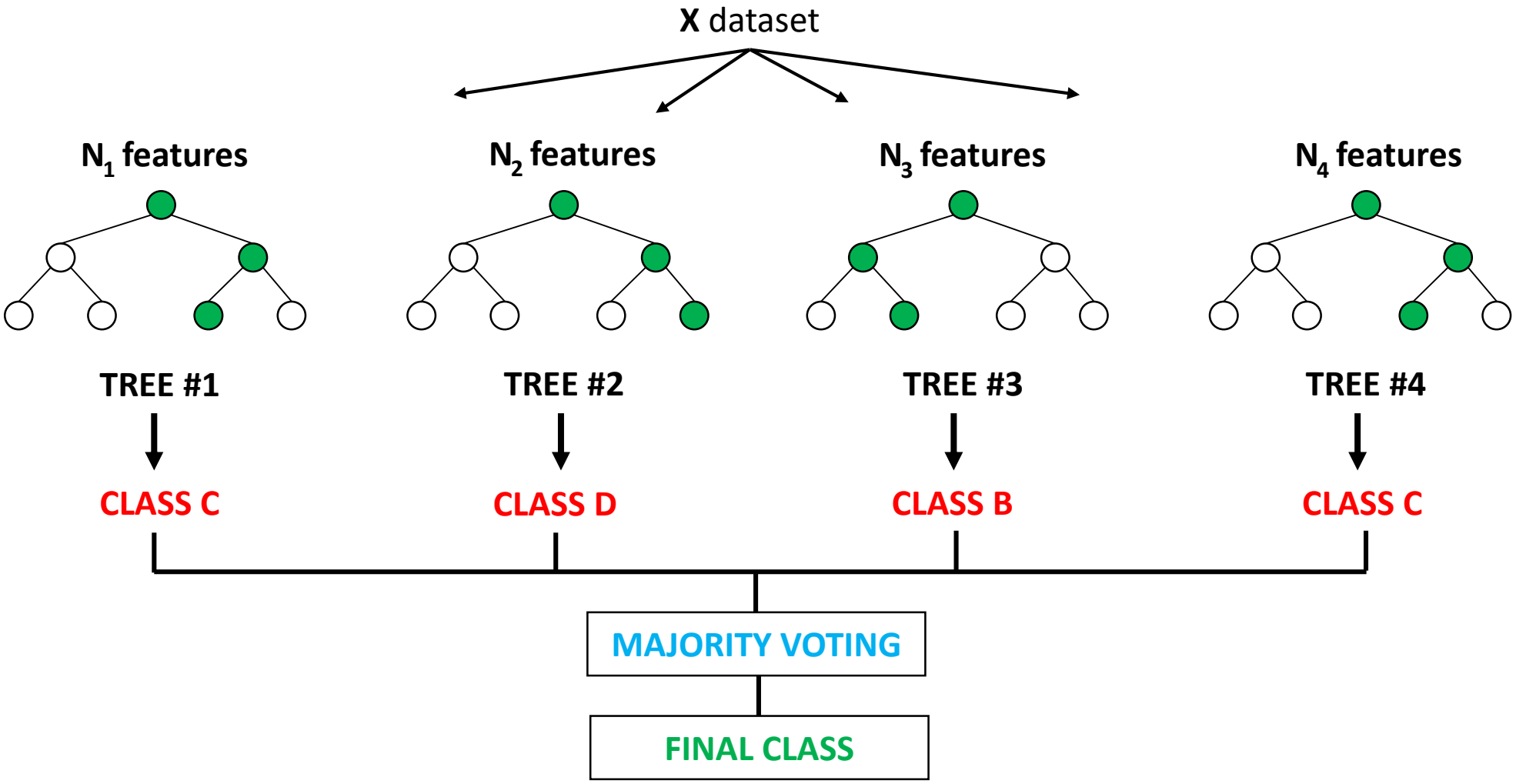
→ similar to bagging: we keep constructing decision trees on the training data **BUT** on every split in the tree, a random selection of features / predictors is chosen from the full feature set

The number of features considered at a given split is approximately equal to the square root of the total number of features (for classification)

Bagging: algorithm searches over all the **N** features to find the best feature that best splits the data at that node

Random Forest Classifier: algorithm searches over a random $\sqrt{\mathbf{N}}$ features to find the best one

Random Forest Classifier



Random Forest Classifier

Why is it good?

- if one or a few features are very strong predictors for the response variable (target output), these features will be selected in many of the decision trees: so they will become correlated
- huge advantage: at some point the variance stops decreasing no matter how many more trees we add to our random forest + it is not going to produce overfitting !!!

Boosting

- it can be used for classification and regression too
- helps to reduce variance and bias !!!
- bagging: creates multiple copies of the original data: constructs several decision trees on the copies and combining all the trees to make predictions

  WE CONSTRUCT THESE TREES INDEPENDENTLY !!!

- **boosting**: here the decision trees are grown sequentially so each tree is grown using information from previously grown trees

  THESE TREES ARE NOT INDEPENDENT OF EACH OTHER

  ~ boosting is a sequential learning algorithm

Boosting

A rather counter-intuitive theory: a weak learner is not able to make good predictions

→ weak learner is just a bit better than random guess or coin flip
For example: decision trees with depth **1**

→ combining weak learners can prove to be an extremely powerful classifier !!!

→ by fitting small trees (**decision stumps**) we slowly improve the final result in cases when it does not perform well

We will consider adaptive boosting „**AdaBoost**” algorithm !!!

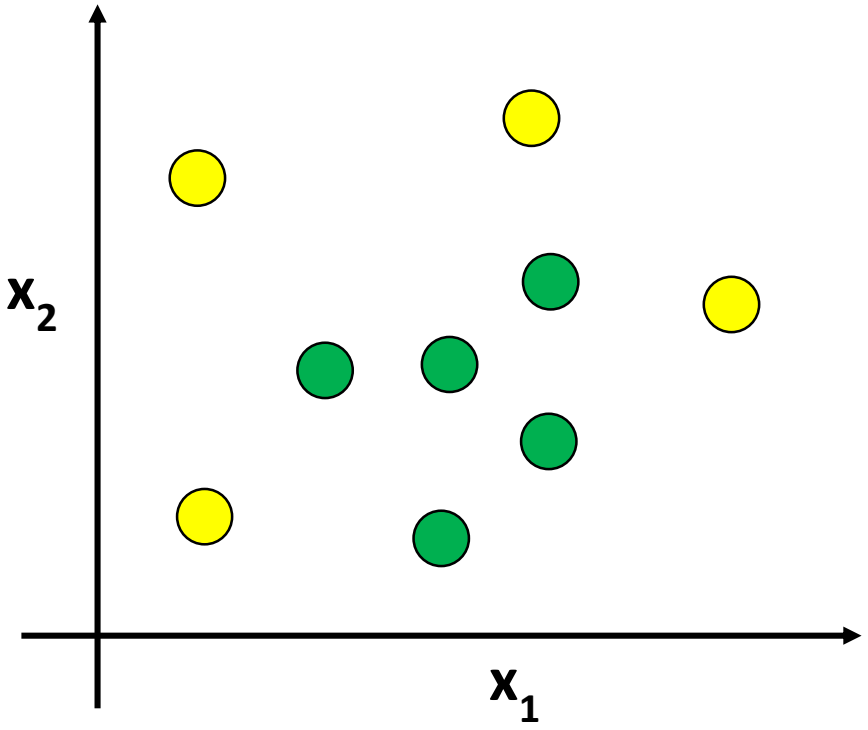
Boosting Applications

Viola-Jones Face Detection algorithm uses boosting
~ combines decision stumps to detect faces

→ weak learners decide whether the given section of the image contains a face or not

→ extremely accurate and fast algorithm

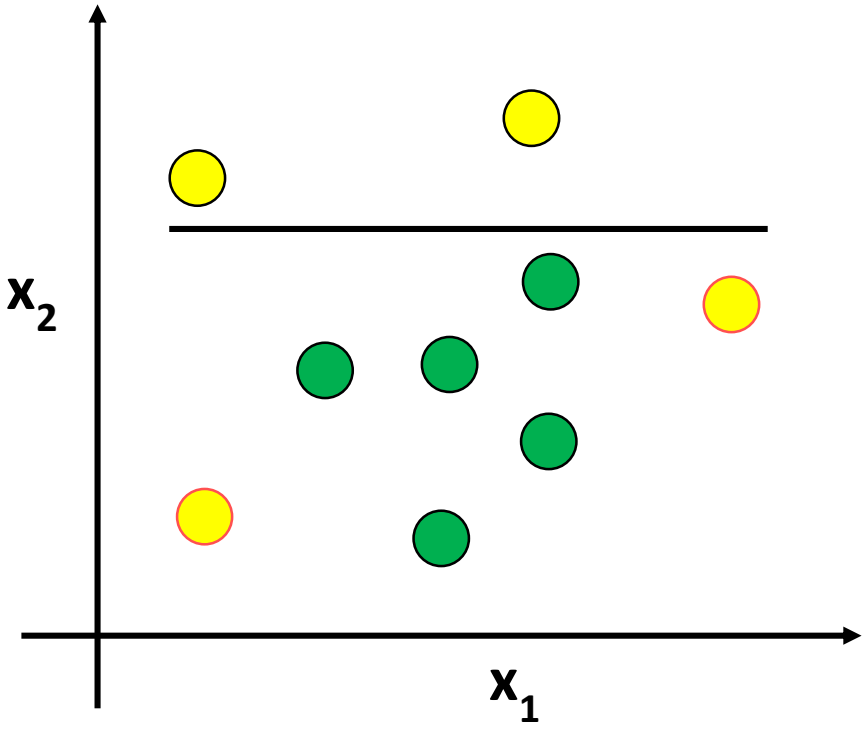
Boosting



For example: we want to classify the dots
~ two **x** features + two output classes (yellow and green)

Boosting: combines very simple weak learners such as
decision trees with depth **1** (capable of linear classification)

Boosting



For example: we want to classify the dots
~ two **x** features + two output classes (yellow and green)

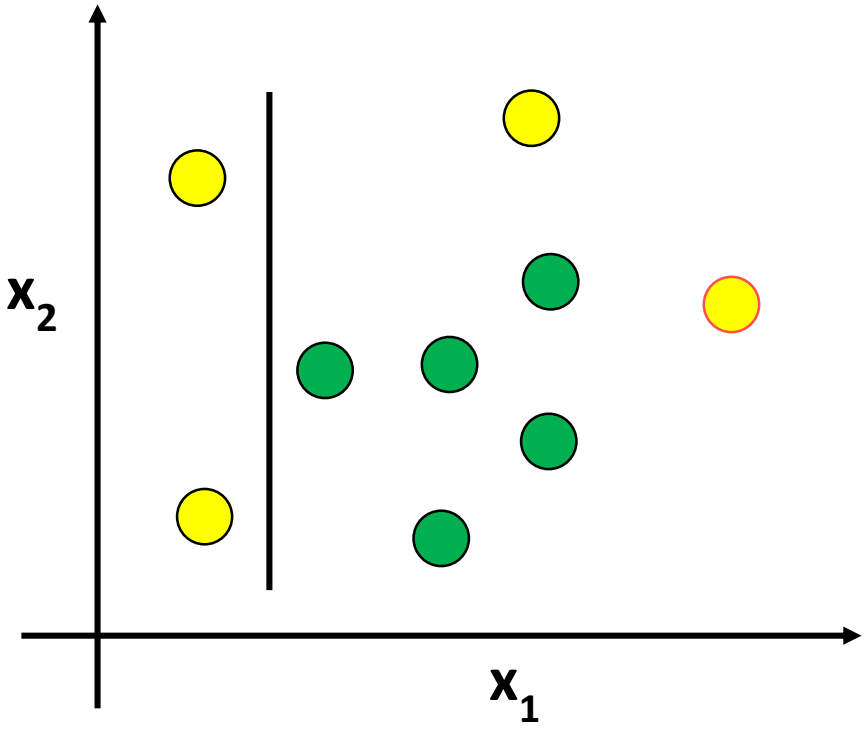
Boosting: combines very simple weak learners such as **decision trees** with depth **1** (capable of linear classification)

→ the classifier made **2** mistakes: two yellow dots are misclassified

→ boosting algorithm: in the next iteration it will focus on the misclassified items

Increase the **w** weights: for misclassified items
Decrease the **w** weights: for correctly classified items

Boosting



For example: we want to classify the dots
~ two **x** features + two output classes (yellow and green)

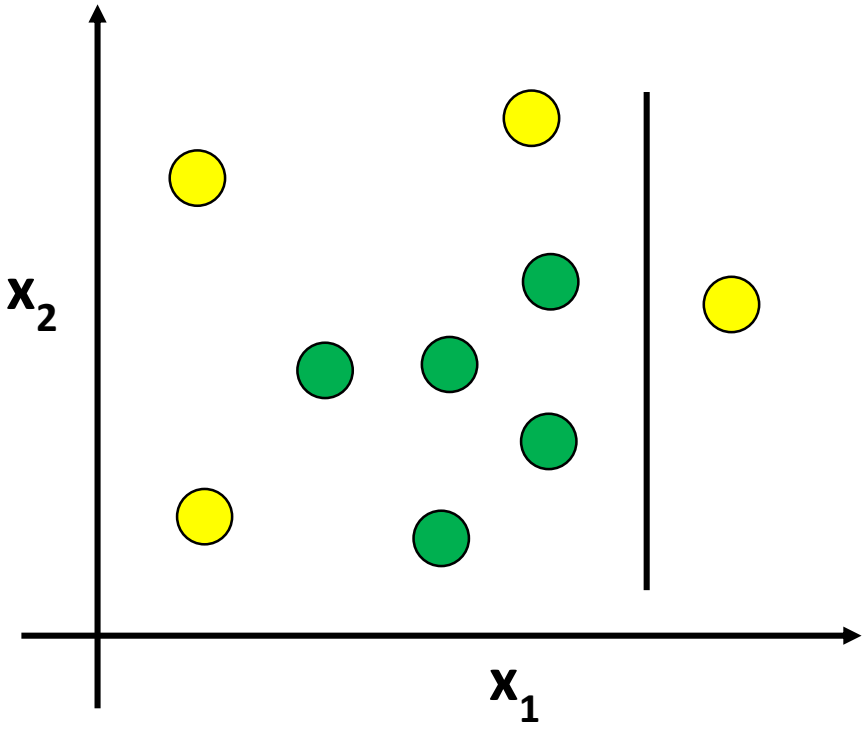
Boosting: combines very simple weak learners such as **decision trees** with depth **1** (capable of linear classification)

→ the classifier made **2** mistakes: two yellow dots are misclassified

→ boosting algorithm: in the next iteration it will focus on the misclassified items

Increase the **w** weights: for misclassified items
Decrease the **w** weights: for correctly classified items

Boosting



For example: we want to classify the dots
~ two **x** features + two output classes (yellow and green)

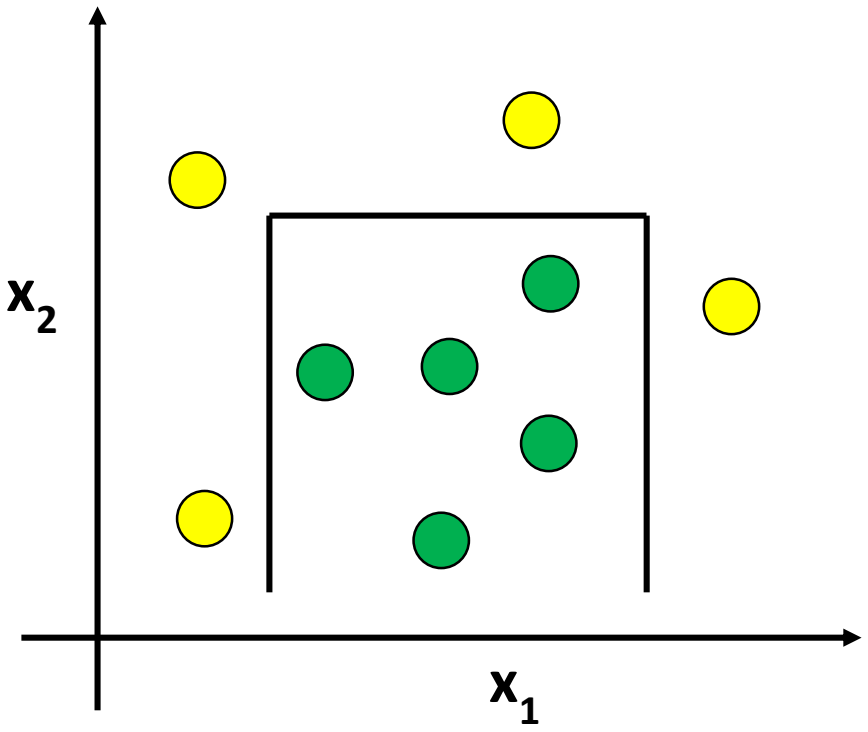
Boosting: combines very simple weak learners such as **decision trees** with depth **1** (capable of linear classification)

→ the classifier made **2** mistakes: two yellow dots are misclassified

→ boosting algorithm: in the next iteration it will focus on the misclassified items

Increase the **w** weights: for misclassified items
Decrease the **w** weights: for correctly classified items

Boosting



For example: we want to classify the dots
~ two **x** features + two output classes (yellow and green)

Boosting: combines very simple weak learners such as **decision trees** with depth **1** (capable of linear classification)

→ the classifier made **2** mistakes: two yellow dots are misclassified

→ boosting algorithm: in the next iteration it will focus on the misclassified items

Increase the **w** weights: for misclassified items
Decrease the **w** weights: for correctly classified items

WE JUST HAVE TO COMBINE THESE WEAK CLASSIFIERS !!!

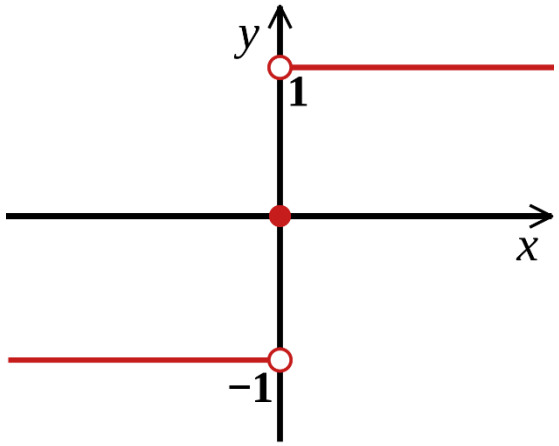
Boosting

we keep combining **h(x)**
weak learners (each learner knows just a little fraction of the space)

$$H(x) = \text{sign} \sum_{i=1}^{n} \alpha_i h_i(x)$$

final **H(x)** model which is a strong classifier

// we assign **+1** and **-1** for the output classes (yellow and green)

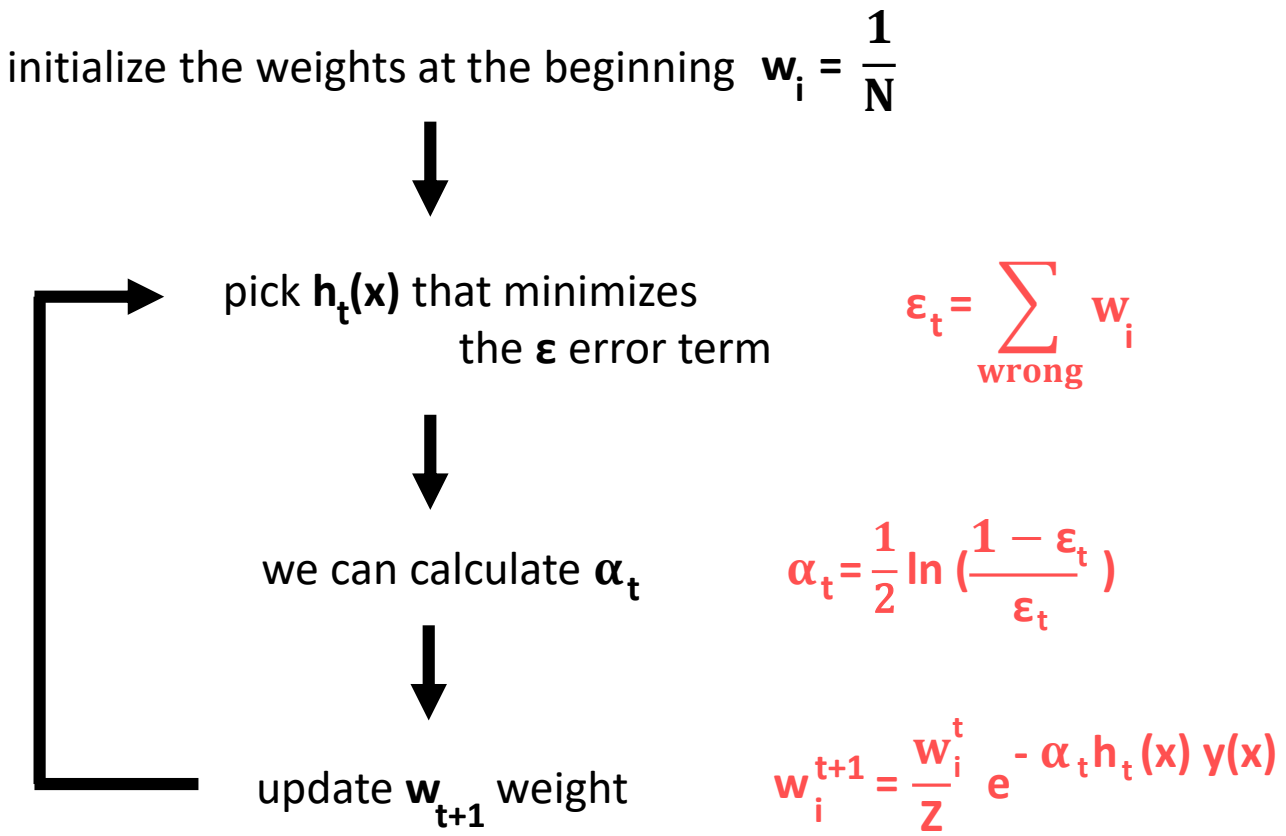


→ we initialize the weights parameters at the beginning $w_i = \frac{1}{N}$

$\sum_{i=1}^{n} w_i = 1$ we make sure this is a distribution

$\varepsilon = \sum_{\text{wrong}} w_i$ error is the sum of the misclassified weights

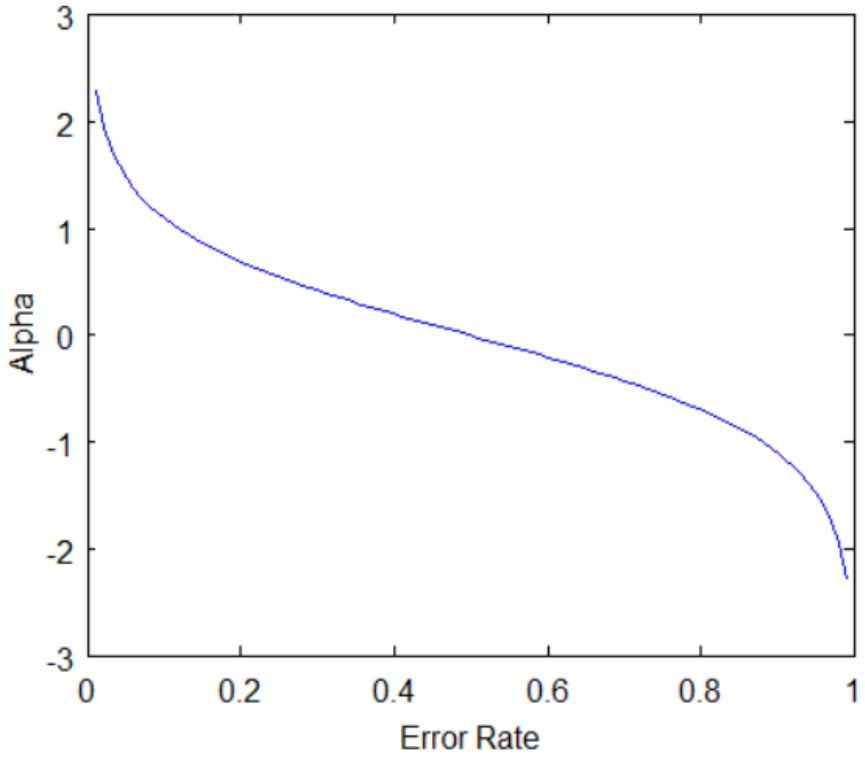
Boosting



On every iteration we add a new **h(x)** weak learner to the final model

Boosting

$$\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \varepsilon_t}{\varepsilon_t} \right)$$



→ given **h(x)** classifier **α** value increases as the error converges to **0**: of course, good classifiers are given more weight

→ given **h(x)** classifier **α** value is **0** if the error is **0.5**
Why? Because it is a random guess (coin toss)
~ we do not want our algorithm to rely on random guesses

→ we give negative **α** value for **h(x)** classifiers that are worse than random guesses
~ we do the opposite thats the best action

This **α** parameter as something to do with the **h(x)** learners !!!

Boosting

$$w_i^{t+1} = \frac{w_i^t}{Z} e^{-\alpha_t h_t(x)\, y(x)}$$

this is how we update the weights in every iteration

→ the **w** weights have something to do with the dataset

→ we set higher weights to more important samples and lower weight values to less important ones

→ the **Z** makes sure **w** is a distribution so the sum is **1**

→ **y(x)** flips the sign of the exponent if **h(x)** is wrong

Why is it good? It makes sure to assign smaller weights to samples that are correctly classified and bigger weights for misclassification

~ in the next iteration the next **h(x)** learner can focus on those samples with higher weights

Why to use **α** in the formula? This is how we make sure that stronger classfiers' decisions are more important

If a weak classifier misclassifies an input we do not take that as seriously as a strong classifier's mistake

Boosting and Bagging

BAGGING	BOOSTING
both bagging and boosting use **N** learners + yields more stable models	
every item has the same probability to appear in a new dataset	the samples are weighted so some of them will occur more often
parallel training stage	builds learners in sequential way
final decision is the average of the **N** learners	final decision is the weighted average of the **N** learners (better classifiers have higher weights)
reduces variance and solves over-fitting	reduces bias but increases over-fitting a bit
PREFERRED	

Computer Vision

So far we have been dealing with known datasets

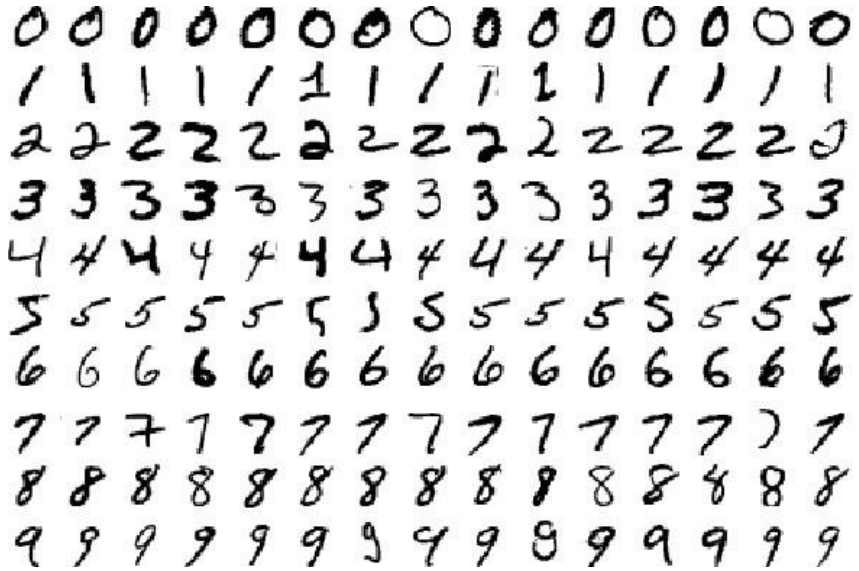
MNIST handwritten digit dataset: we know that there are **28x28** pixels image
+ we know that there are **10** output classes

Olivetti Faces dataset: contains **64x64** pixels images

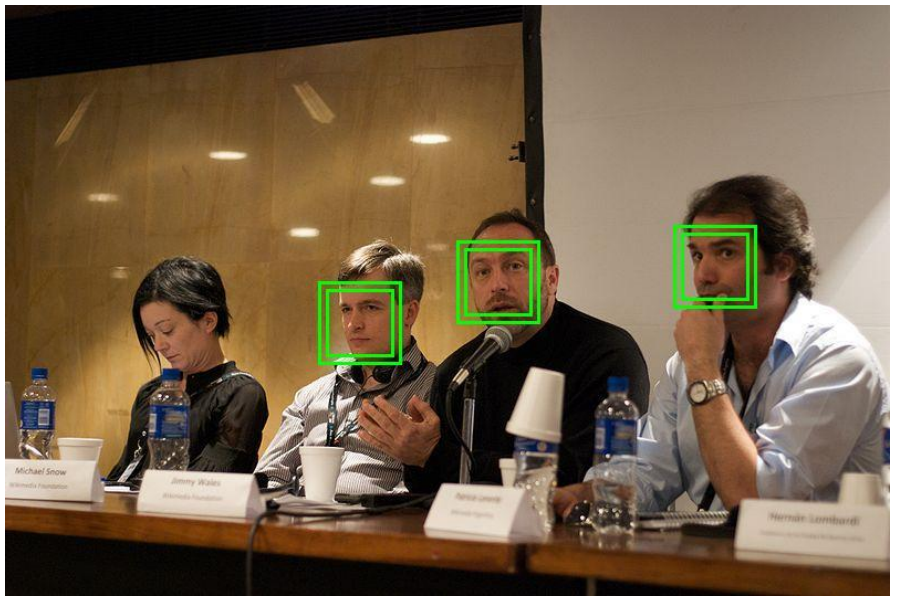
WE KNOW WHAT WE ARE LOOKING FOR

Computer vision: we are after a general solution
If we have a given image → we want to make sure the algorithm is able to detect and recognize objects (faces, cars or horses)

Computer Vision



we can tune the given machine learning algorithm or neural networks to deal with this specific task



we have to find the object on the image and make sure our detection algorithm is correct

Viola-Jones Algorithm

This algorithm tries to find the most relevant
features for a human face (we are dealing with face detection)

→ what are relevant features for detecting faces?

→ two eyes, nose, lips, forehead ...

→ we can construct an algorithm based on the most relevant
features: if the algorithm does not find one of these
features it comes to the conclusion that there is no
human face on that region of the image

THIS ALGORITHM HANDLES GRAYSCALE IMAGES
~ so the first step is to convert the image into grayscale

Viola-Jones Algorithm

The algorithm was formulated by **Paul Viola** and **Michael Jones** in **2001**

→ it is a machine learning algorithm: so it needs training as well
~ training phase + testing (detecting the objects)

→ the algorithm needs positive images (images of faces) and
negative images (images without faces) as well
~ this is how it learns the most relevant feature



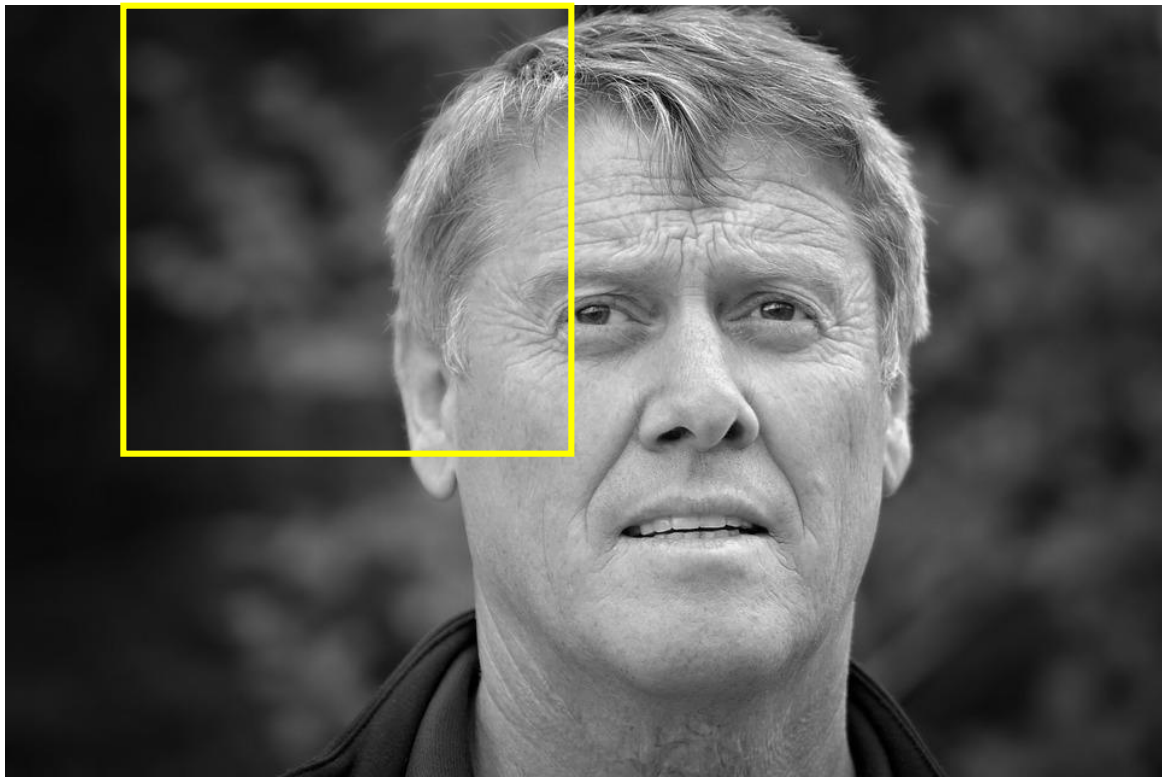




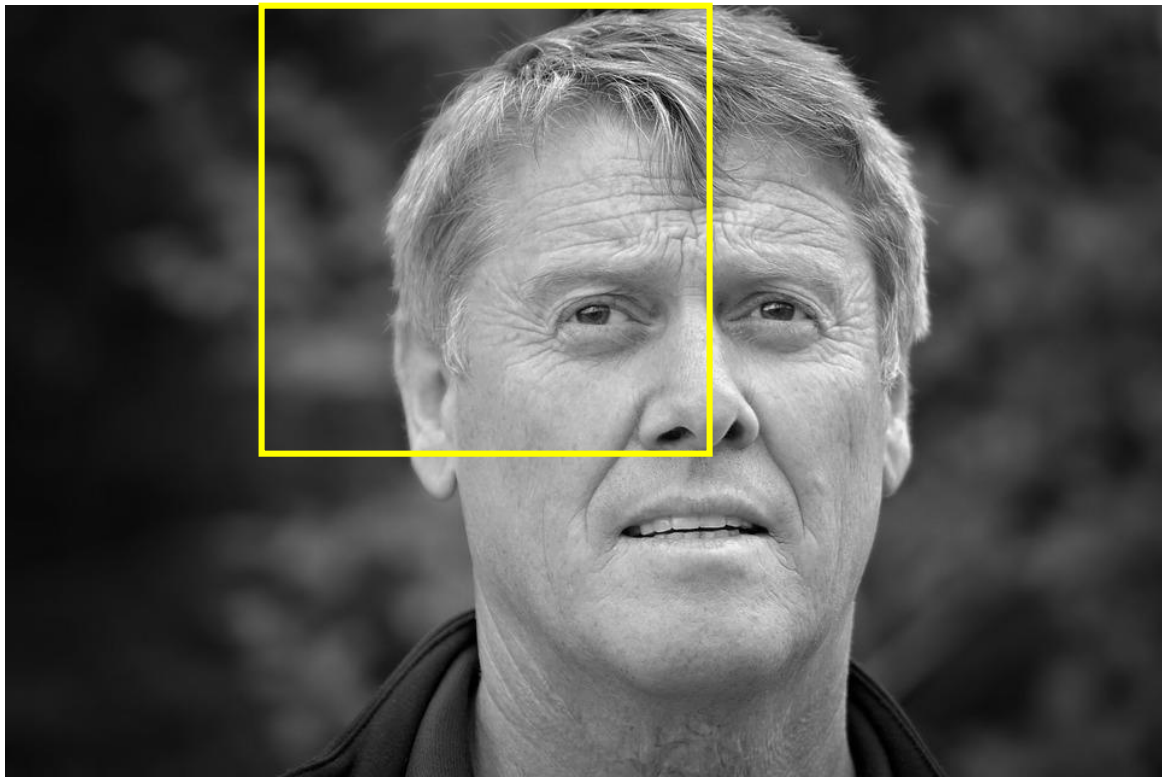
Viola-Jones Algorithm

Viola-Jones Algorithm

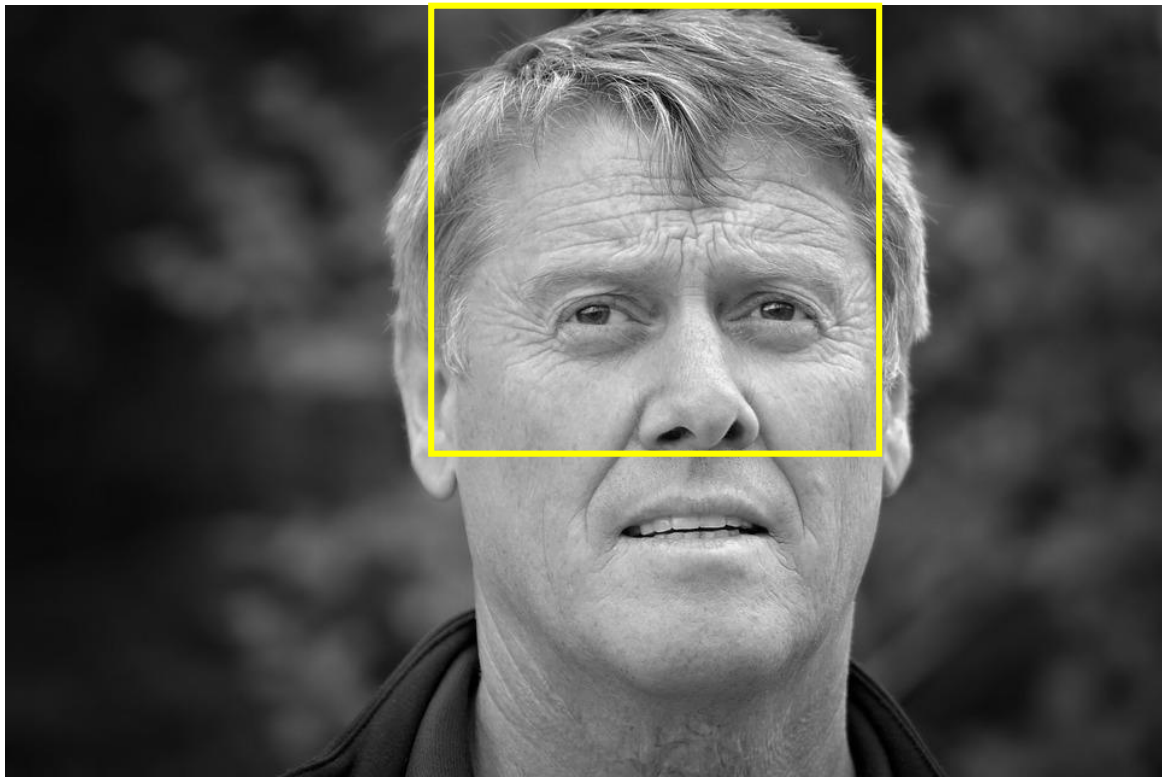
Viola-Jones Algorithm



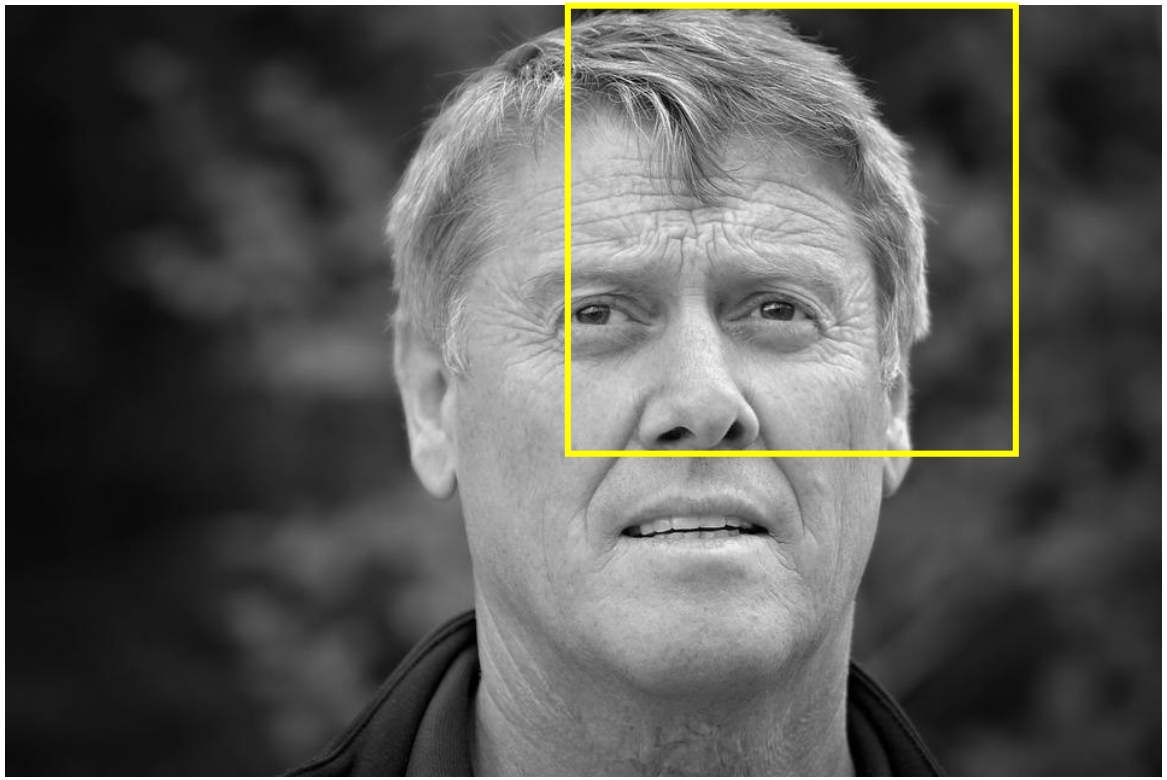
Viola-Jones Algorithm



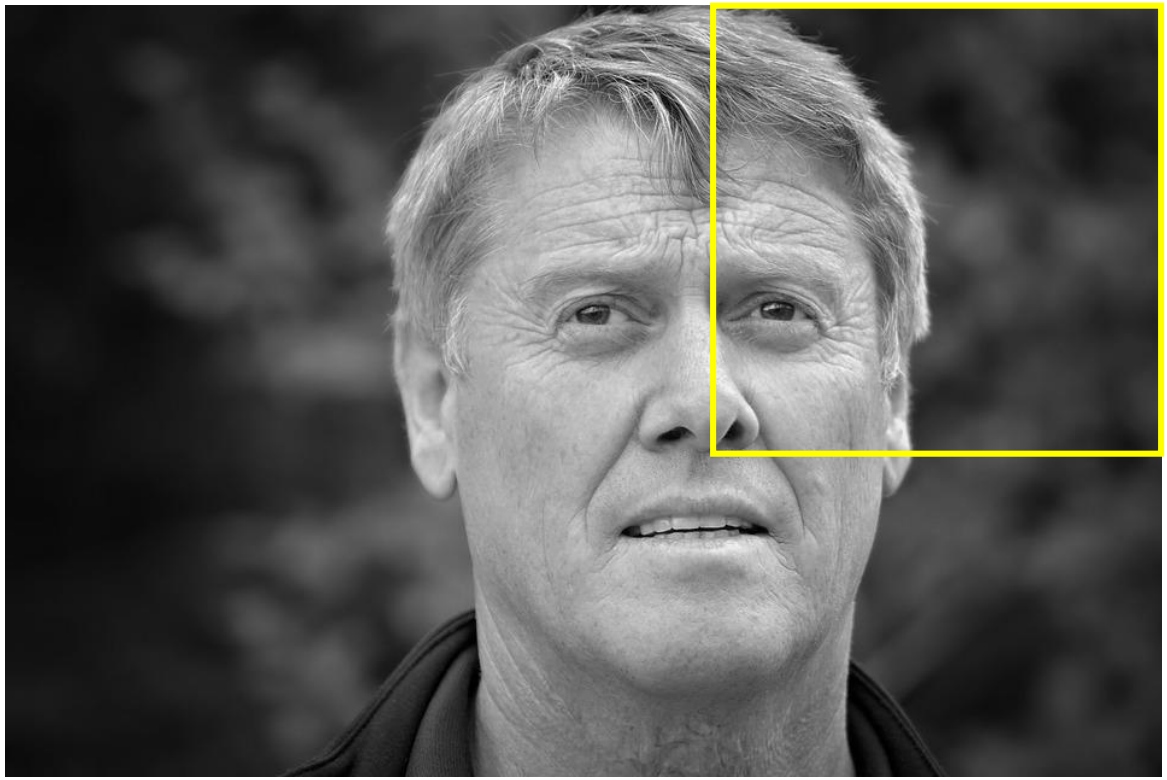
Viola-Jones Algorithm



Viola-Jones Algorithm

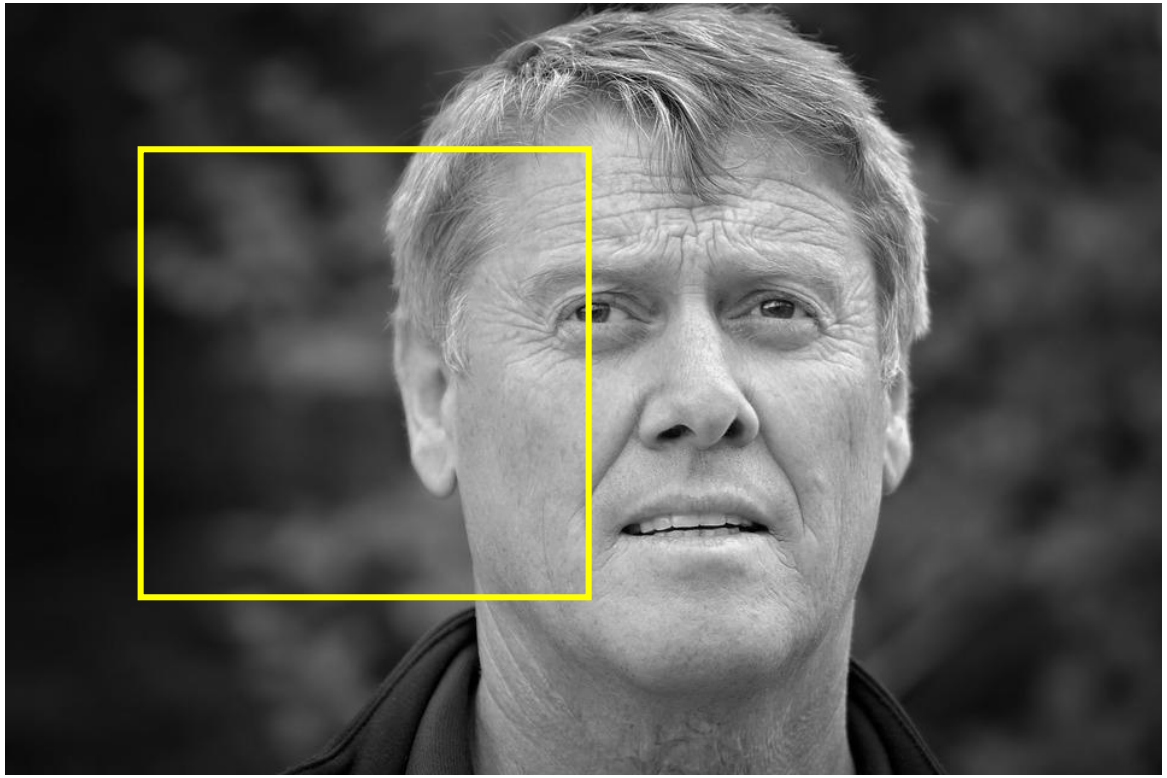


Viola-Jones Algorithm

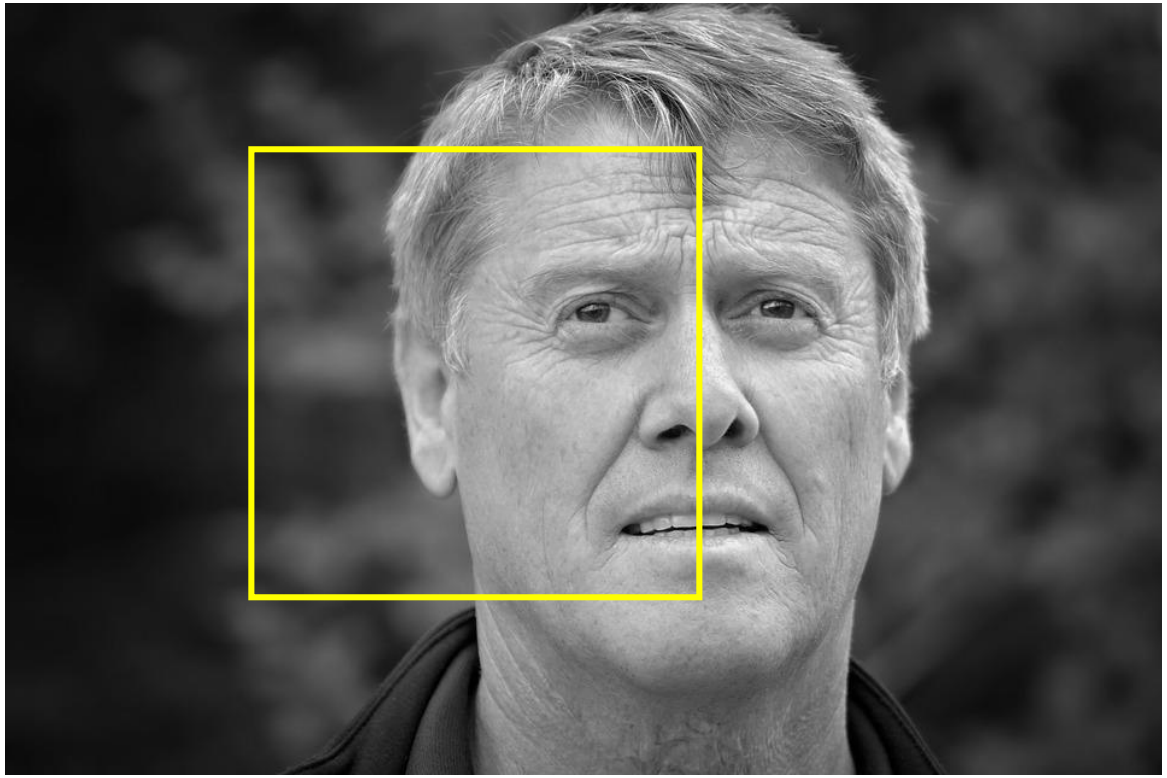


Viola-Jones Algorithm

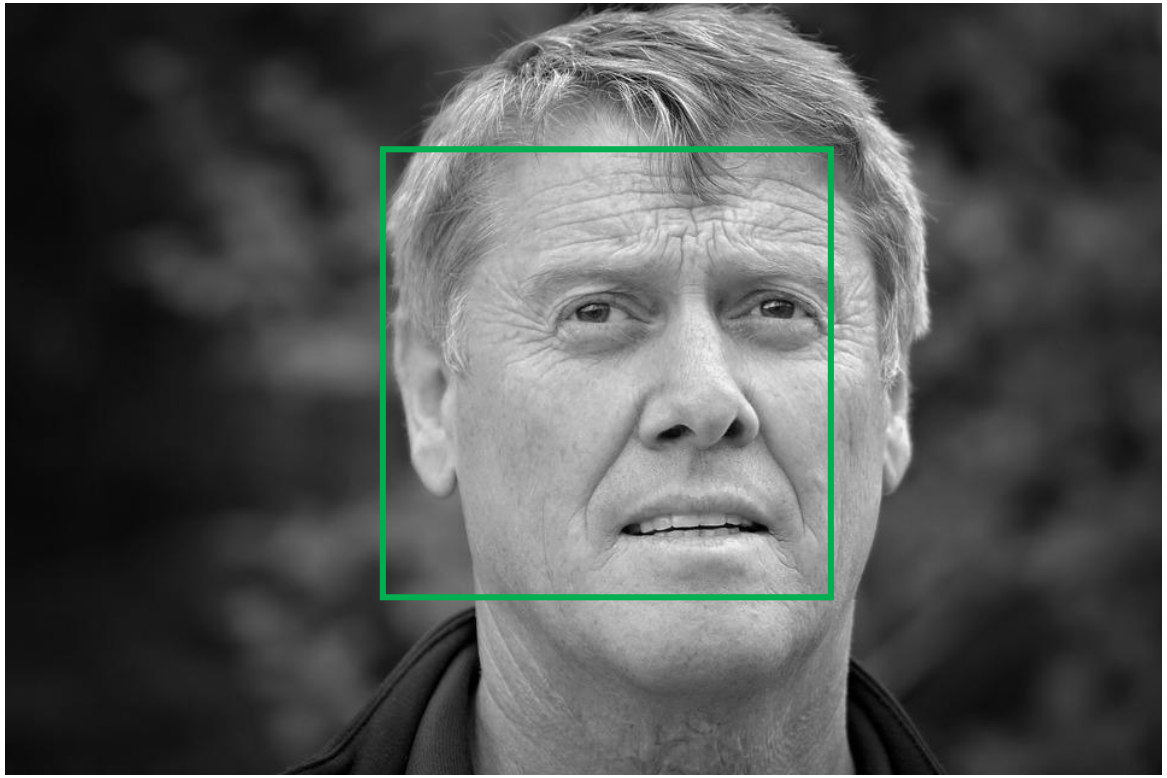
Viola-Jones Algorithm



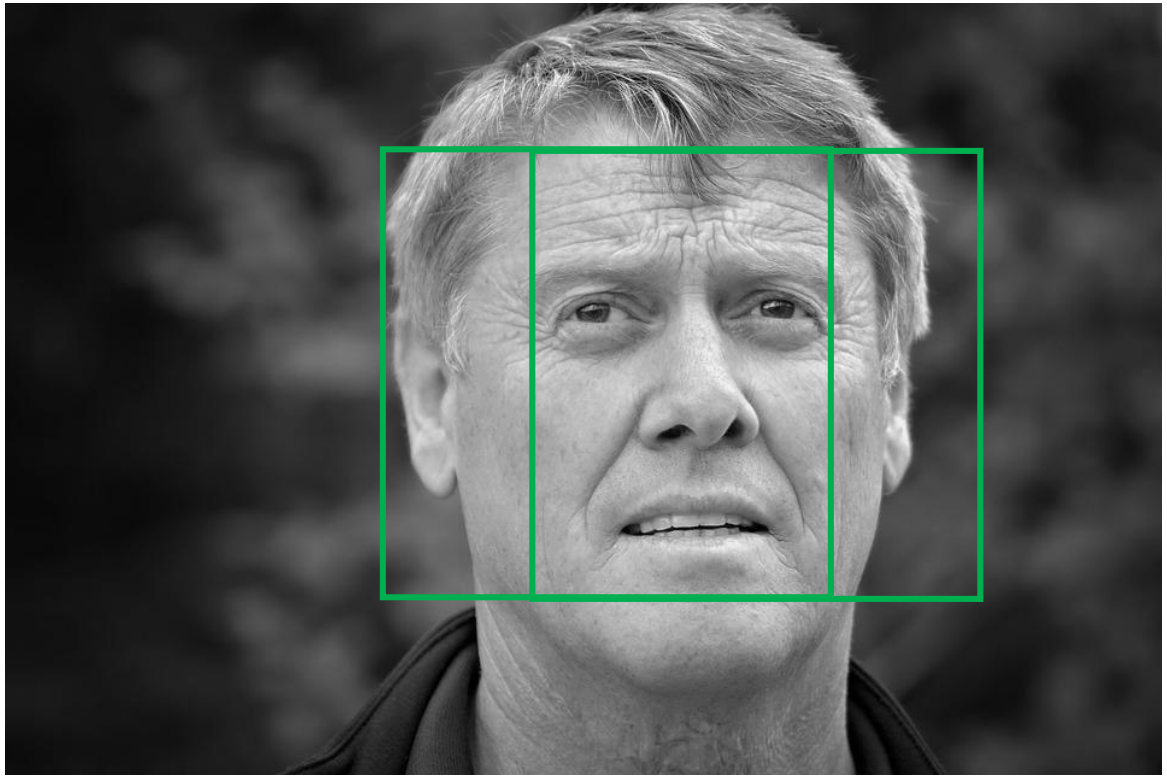
Viola-Jones Algorithm



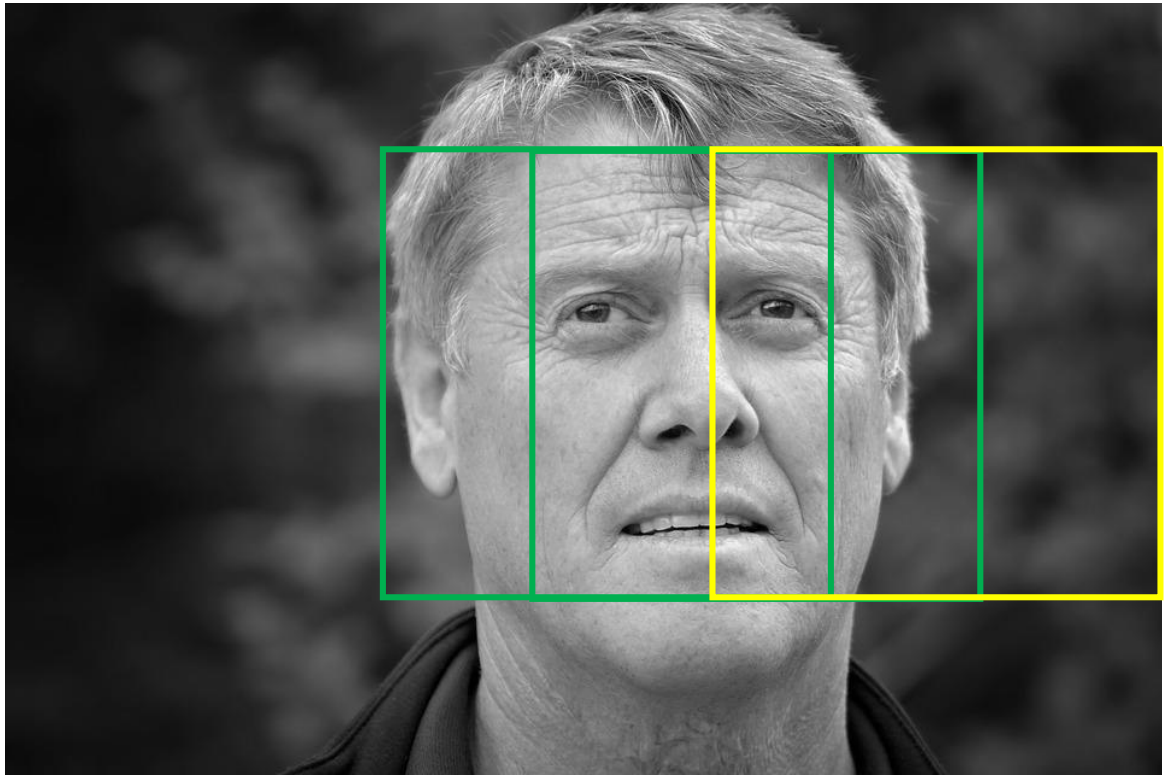
Viola-Jones Algorithm



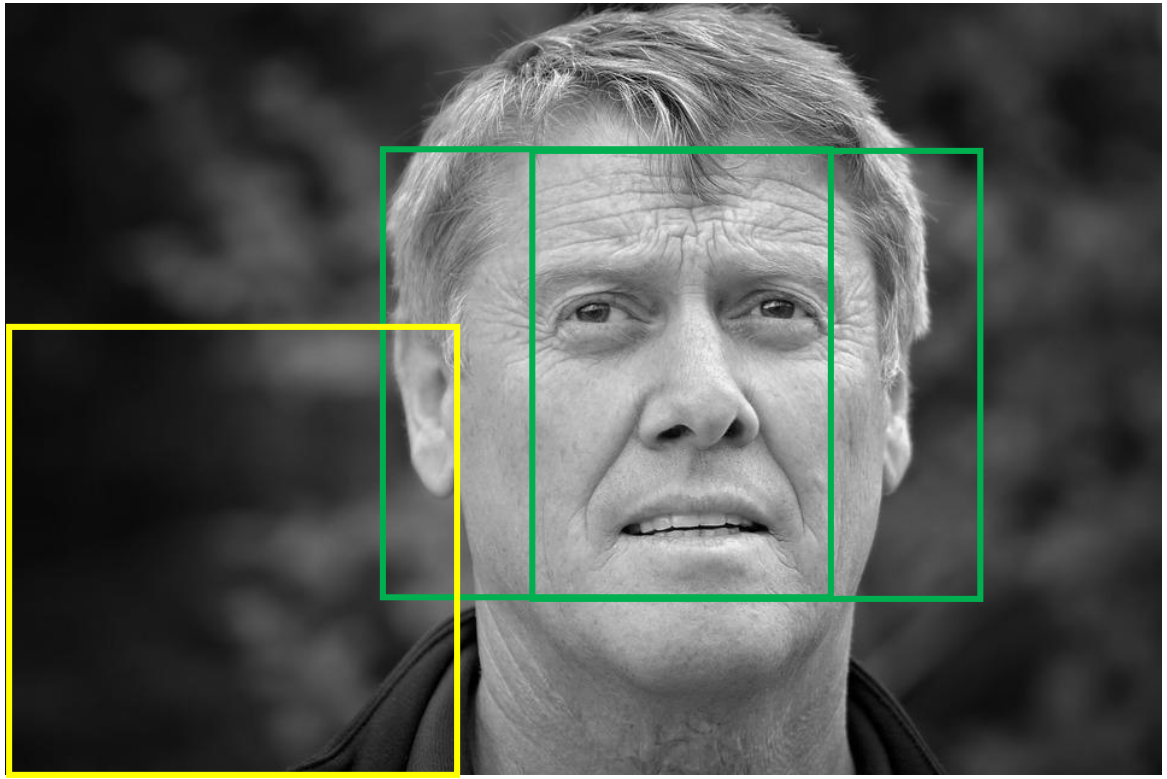
Viola-Jones Algorithm



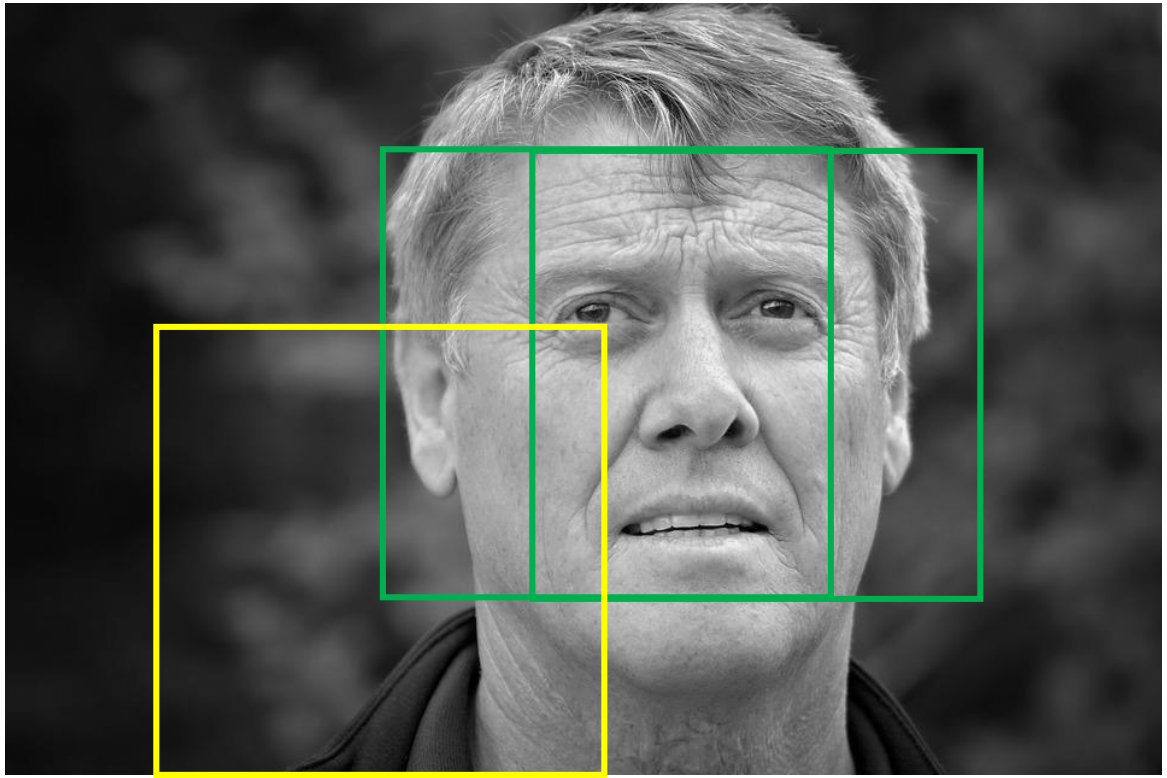
Viola-Jones Algorithm



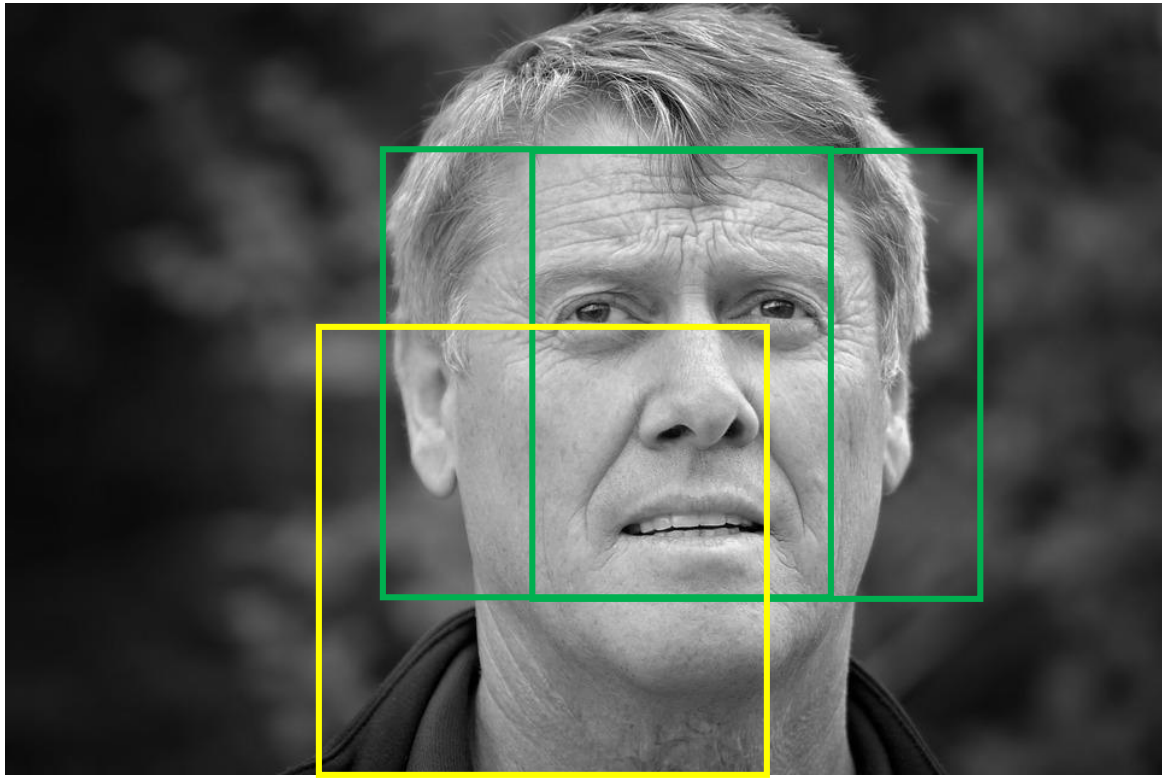
Viola-Jones Algorithm



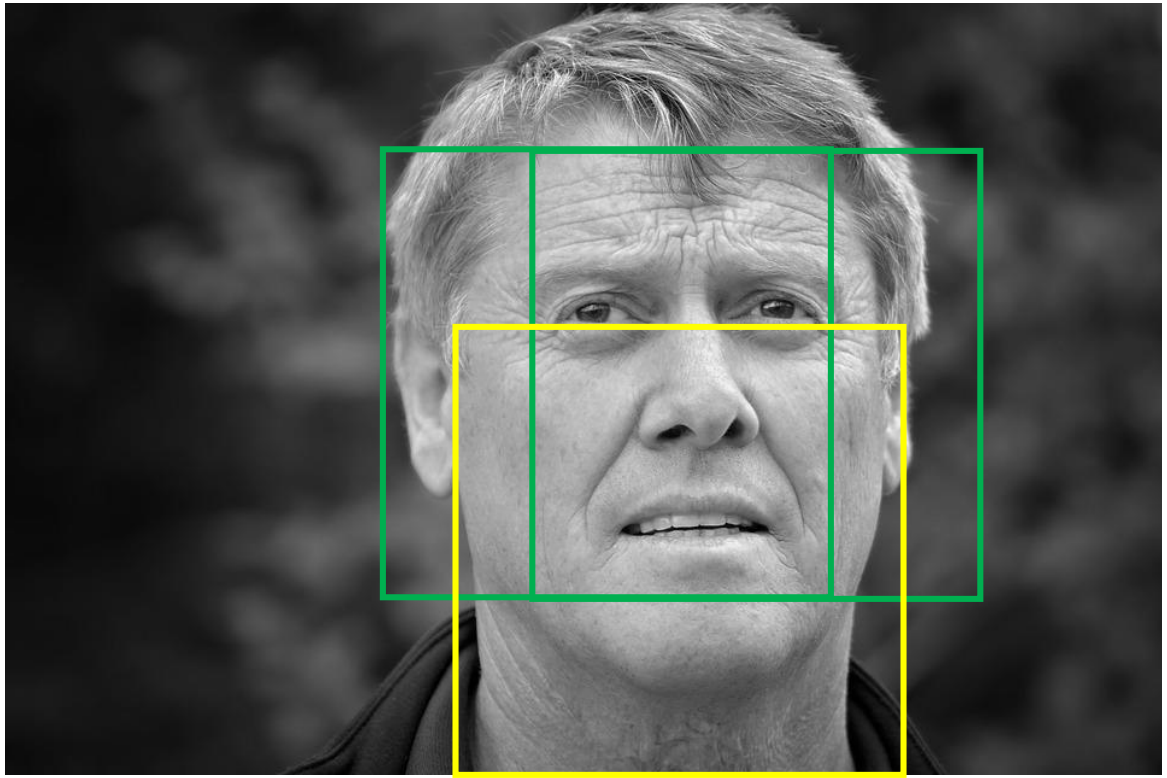
Viola-Jones Algorithm



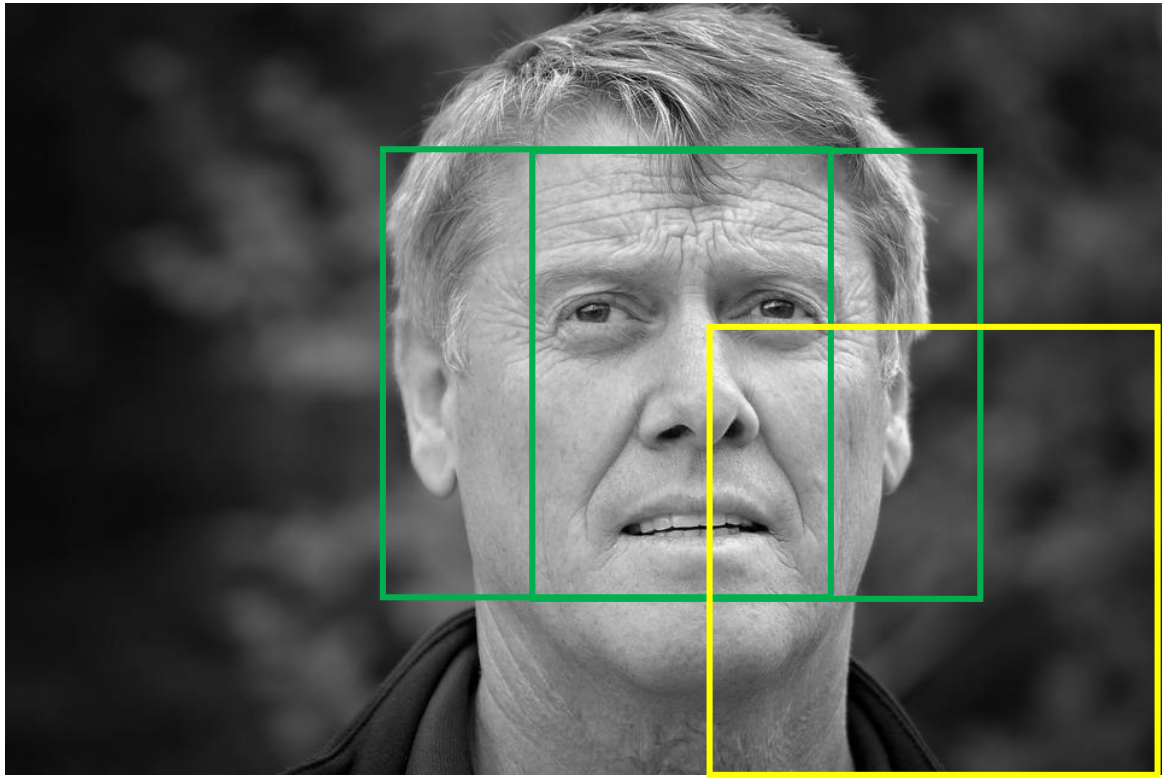
Viola-Jones Algorithm



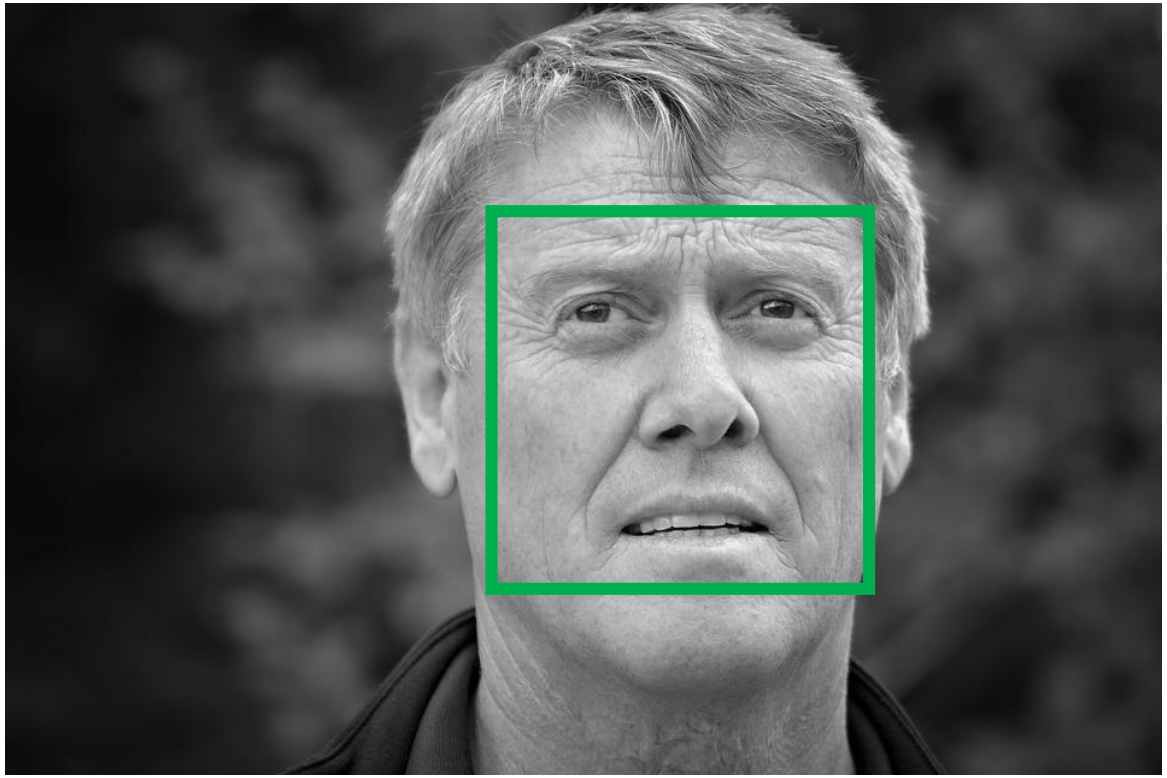
Viola-Jones Algorithm



Viola-Jones Algorithm



Viola-Jones Algorithm



Viola-Jones Algorithm

We have a huge problem with this approach: size of faces may differ
(window size is too small to include all the relevant features of a face
so the algorithm will not work fine)

~ some faces may be closer to the camera which means they
appear bigger than other faces in the background

scaleFactor: this feature in **OpenCV** compensates for this issue

The training uses **24x24** pixels images so we have to rescale the input image
~ so we can resize a larger face to a smaller one

Haar-features

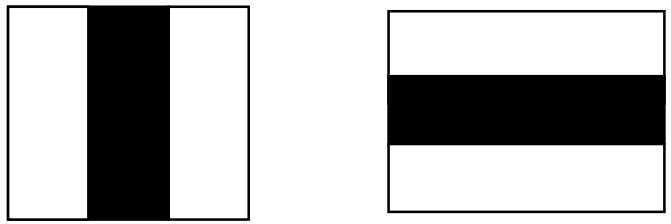
Haar-wavelet is a sequence of rescaled square-shaped „functions”
~ very similar to Fourier-analysis

→ it was proposed by **Alfréd Haar** in **1909**

→ they are like convolutional kernels

HAAR FEATURES ARE THE RELEVANT FEATURES FOR FACE DETECTION

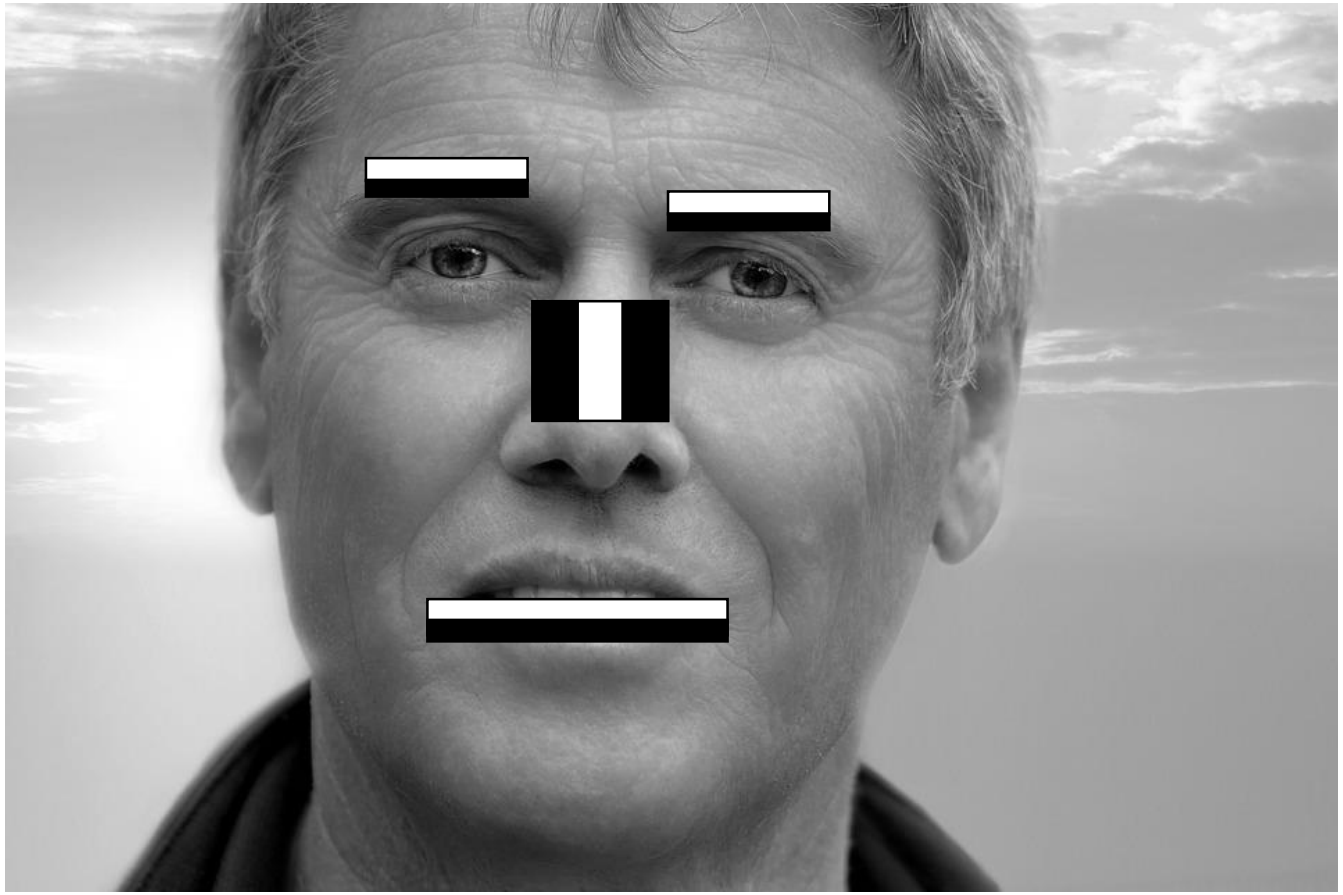
edge features can detect edges
quite effectively



line features can detect lines
quite effectively

Haar-features

Haar-features



WE CAN REPRESENT THE MOST RELEVANT FEATURES WITH HAAR-FEATURES !!!

Haar-features

0	0	1	1
0	0	1	1
0	0	1	1
0	0	1	1

ideal **Haar-feature**
pixel intensities
0: white
1: black

0.1	0.2	0.6	0.8
0.2	0.3	0.8	0.6
0.2	0.1	0.6	0.8
0.2	0.1	0.8	0.9

these are real values
detected on an image

Viola-Jones algorithm will compare how close the real scenario is to the ideal case

1.) let's sum up the white pixel intensities

2.) calculate the sum of the black pixel intensities

$$\Delta = \text{dark} - \text{white} = \frac{1}{n}\sum_{\text{dark}}^{n} I(x) - \frac{1}{n}\sum_{\text{white}}^{n} I(x)$$

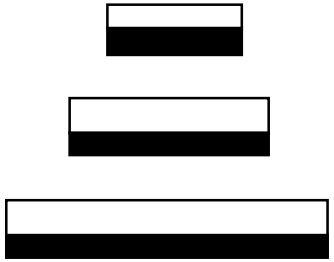
Δ for ideal Haar-feature is **1**

Δ for the real image: **0.74 – 0.18 = 0.56**

The closer the value to **1**, the more likely we have found a **Haar-feature** !!!
(of course we will never get **0** or **1**: there are thresholds)

Integral Image

0.1	0.1	0.2	0.1	0.7	0.1
0.2	0.3	0.2	0.7	0.8	0.2
0.1	0.4	0.3	0.3	0.1	0.3
0.1	0.5	0.1	0.1	0.2	0.8
0.1	0.4	0.8	0.5	0.6	0.5



these are the same kernels but with different sizes

The problem is that we have to calculate the average of a given region several times

~ the time complexity of these operations are **O(N^2)**

WE CAN USE INTEGRAL IMAGE APPROACH TO ACHIEVE O(1) RUNNING TIME

Why are there so many operations?

Because we have to use Haar-features with all possible sizes and locations

~ **200k** features to calculate !!!

→ every time we have to use a quadratic algorithm

~ we have to find a better approach

Integral Image

0.1	0.1	0.2	0.1	0.7	0.1
0.2	0.3	0.2	0.7	0.8	0.2
0.1	0.4	0.3	0.3	0.1	0.3
0.1	0.5	0.1	0.1	0.2	0.8
0.1	0.4	0.8	0.5	0.6	0.5

original image

0.1	0.2	0.4	0.5	1.2	1.3
0.3	0.7	1.1	1.9	3.4	3.7
0.4	1.2	1.9	3.0	4.6	5.2
0.5	1.7	2.5	3.7	5.3	6.7
0.6	2.3	3.9	5.6	8.0	9.9

integral image

Integral Image

0.1	0.1	0.2	0.1	0.7	0.1
0.2	0.3	0.2	0.7	0.8	0.2
0.1	0.4	0.3	0.3	0.1	0.3
0.1	0.5	0.1	0.1	0.2	0.8
0.1	0.4	0.8	0.5	0.6	0.5

original image

0.1	0.2	0.4	0.5	1.2	1.3
0.3	0.7	1.1	1.9	3.4	3.7
0.4	1.2	1.9	3.0	4.6	5.2
0.5	1.7	2.5	3.7	5.3	6.7
0.6	2.3	3.9	5.6	8.0	9.9

integral image

Integral Image

0.1	0.1	0.2	0.1	0.7	0.1
0.2	0.3	0.2	0.7	0.8	0.2
0.1	0.4	0.3	0.3	0.1	0.3
0.1	0.5	0.1	0.1	0.2	0.8
0.1	0.4	0.8	0.5	0.6	0.5

original image

0.1	0.2	0.4	0.5	1.2	1.3
0.3	0.7	1.1	1.9	3.4	3.7
0.4	1.2	1.9	3.0	4.6	5.2
0.5	1.7	2.5	3.7	5.3	6.7
0.6	2.3	3.9	5.6	8.0	9.9

integral image

Integral Image

0.1	0.1	0.2	0.1	0.7	0.1
0.2	0.3	0.2	0.7	0.8	0.2
0.1	0.4	0.3	0.3	0.1	0.3
0.1	0.5	0.1	0.1	0.2	0.8
0.1	0.4	0.8	0.5	0.6	0.5

original image

0.1	0.2	0.4	0.5	1.2	1.3
0.3	0.7	1.1	1.9	3.4	3.7
0.4	1.2	1.9	3.0	4.6	5.2
0.5	1.7	2.5	3.7	5.3	6.7
0.6	2.3	3.9	5.6	8.0	9.9

integral image

Integral Image

0.1	0.1	0.2	0.1	0.7	0.1
0.2	0.3	0.2	0.7	0.8	0.2
0.1	0.4	0.3	0.3	0.1	0.3
0.1	0.5	0.1	0.1	0.2	0.8
0.1	0.4	0.8	0.5	0.6	0.5

original image

0.1	0.2	0.4	0.5	1.2	1.3
0.3	0.7	1.1	1.9	3.4	3.7
0.4	1.2	1.9	3.0	4.6	5.2
0.5	1.7	2.5	3.7	5.3	6.7
0.6	2.3	3.9	5.6	8.0	9.9

integral image

SUM = 3.7

Integral Image

0.1	0.1	0.2	0.1	0.7	0.1
0.2	0.3	0.2	0.7	0.8	0.2
0.1	0.4	0.3	0.3	0.1	0.3
0.1	0.5	0.1	0.1	0.2	0.8
0.1	0.4	0.8	0.5	0.6	0.5

original image

0.1	0.2	0.4	0.5	1.2	1.3
0.3	0.7	1.1	1.9	3.4	3.7
0.4	1.2	1.9	3.0	4.6	5.2
0.5	1.7	2.5	3.7	5.3	6.7
0.6	2.3	3.9	5.6	8.0	9.9

integral image

SUM = 3.7 – 0.5

Integral Image

0.1	0.1	0.2	0.1	0.7	0.1
0.2	0.3	0.2	0.7	0.8	0.2
0.1	0.4	0.3	0.3	0.1	0.3
0.1	0.5	0.1	0.1	0.2	0.8
0.1	0.4	0.8	0.5	0.6	0.5

original image

0.1	0.2	0.4	0.5	1.2	1.3
0.3	0.7	1.1	1.9	3.4	3.7
0.4	1.2	1.9	3.0	4.6	5.2
0.5	1.7	2.5	3.7	5.3	6.7
0.6	2.3	3.9	5.6	8.0	9.9

integral image

SUM = 3.7 – 0.5 + 0.2

Integral Image

0.1	0.1	0.2	0.1	0.7	0.1
0.2	0.3	0.2	0.7	0.8	0.2
0.1	0.4	0.3	0.3	0.1	0.3
0.1	0.5	0.1	0.1	0.2	0.8
0.1	0.4	0.8	0.5	0.6	0.5

original image

0.1	0.2	0.4	0.5	1.2	1.3
0.3	0.7	1.1	1.9	3.4	3.7
0.4	1.2	1.9	3.0	4.6	5.2
0.5	1.7	2.5	3.7	5.3	6.7
0.6	2.3	3.9	5.6	8.0	9.9

integral image

SUM = 3.7 – 0.5 + 0.2 – 1.7 = 1.7

Why is it good? We can achieve **O(1)** running time for handling **Haar-features**
~ we assume these features are rectangles

Computer Vision - Boosting

→ we can boost some part of the algorithm with the help of integral image approach

→ **BUT** there are way too many features
~ most of the features are irrelevant and not important at all

HOW TO SELECT THE BEST FEATURES? WITH BOOSTING !!!

we keep combining **h(x)**
weak learners (weak learner with
a single **Haar-feature**)

$$H(x) = \text{sign} \sum_{i=1}^{n} \alpha_i h_i(x)$$

final **H(x)** model which
is a strong classifier

every **h(x)** week learner
make a prediction
based on a single **Haar-feature**

Computer Vision - Boosting

At the beginning all the **h(x)** weak learners have the same weight
~ all of them contribute to the final decision (face is detected or not)



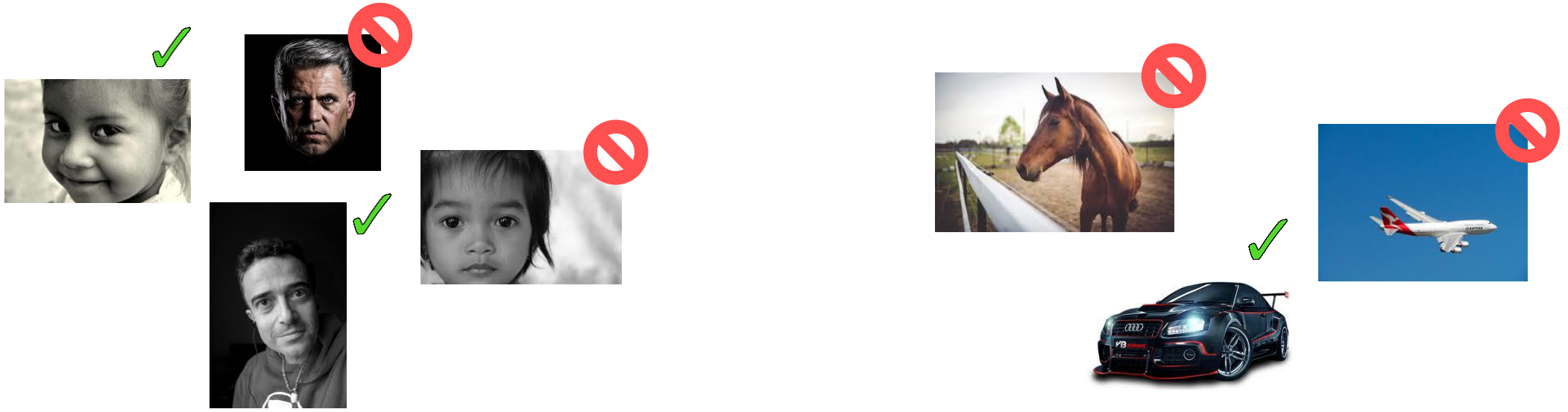




During the training phase **Viola-Jones algorithm** update
the weights for the **h(x)** weak learners (the features)
and finally we have the relevant features with higher weights

Computer Vision - Boosting

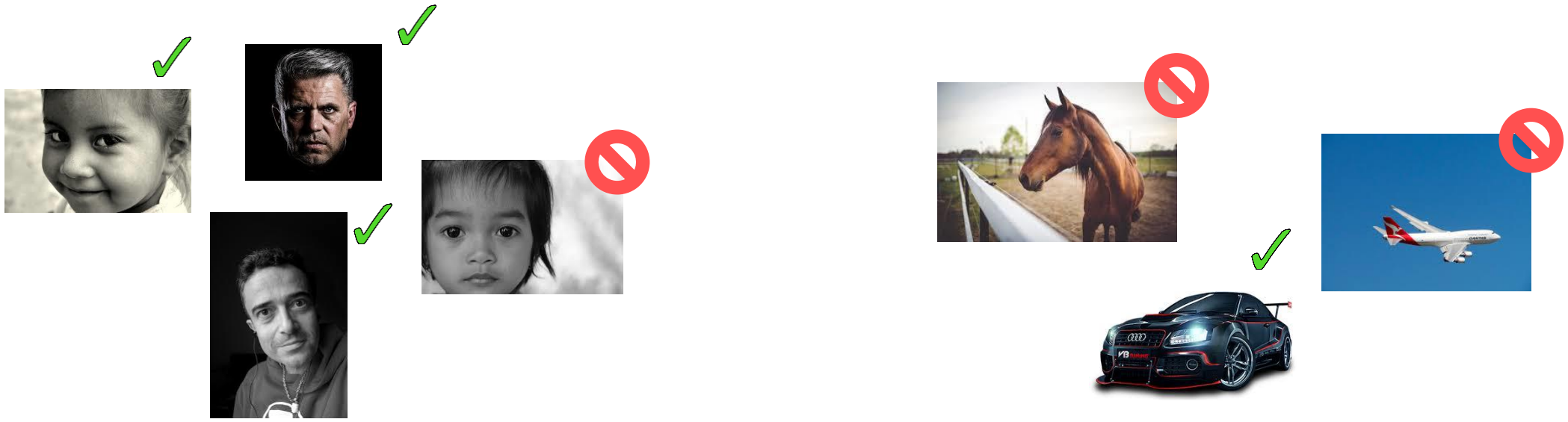
At the beginning all the **h(x)** weak learner have the same weight
~ all of them contribute to the final decision (face is detected or not)



During the training phase **Viola-Jones algorithm** update
the weights for the **h(x)** weak learners (the features)
and finally we have the relevant features with higher weights

Computer Vision - Boosting

At the beginning all the **h(x)** weak learner have the same weight
~ all of them contribute to the final decision (face is detected or not)



During the training phase **Viola-Jones algorithm** update
the weights for the **h(x)** weak learners (the features)
and finally we have the relevant features with higher weights

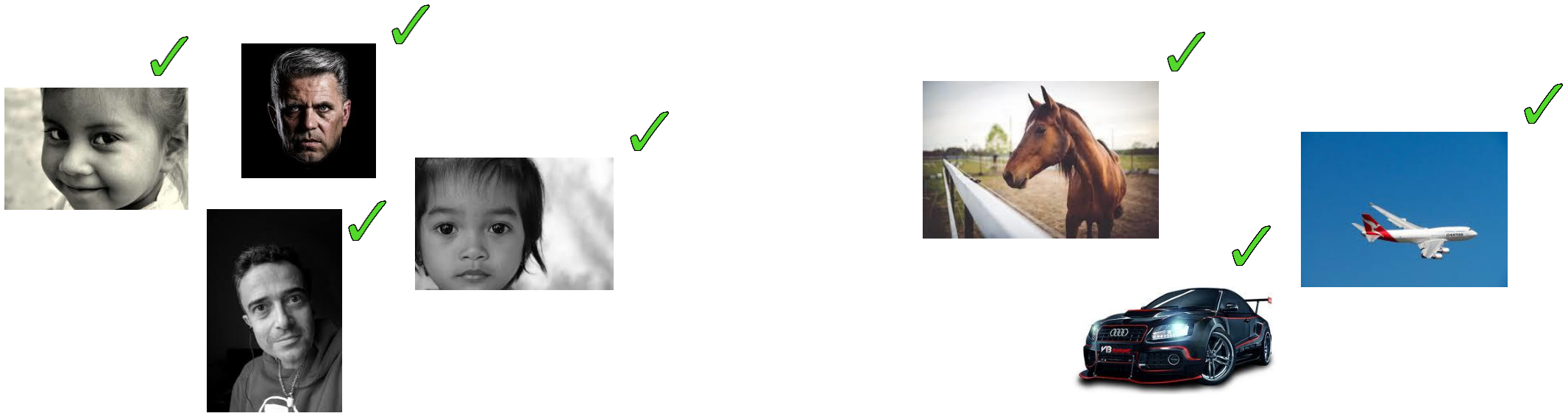
Computer Vision - Boosting

At the beginning all the **h(x)** weak learner have the same weight
~ all of them contribute to the final decision (face is detected or not)

During the training phase **Viola-Jones algorithm** update
the weights for the **h(x)** weak learners (the features)
and finally we have the relevant features with higher weights

Computer Vision - Boosting

At the beginning all the **h(x)** weak learner have the same weight
~ all of them contribute to the final decision (face is detected or not)



During the training phase **Viola-Jones algorithm** update
the weights for the **h(x)** weak learners (the features)
and finally we have the relevant features with higher weights

Cascading

With boosting we can make the algorithm quite fast ...
But can we do even better?

→ we know that most of the image region is non-face region

→ it is a better idea to have a simple method to check if a window is not a face region: if it is not, discard it in a single shot

→ do not process unnecessary regions: focus on regions where there can be a face instead

THIS IS WHY WE USE THE CASCADE CLASSIFIER CONCEPT

Instead of applying all the ~**6000 Haar-features**: we use the most relevant ones in the first iteration (first stages contain very less features: **1, 10, 15, 50**...)

Cascading

How to find the most relevant features?

~ boosting algorithm has already found the best features

we keep combining **h(x)** weak learners (weak learner with a single **Haar-feature**)

$$H(x) = \text{sign} \sum_{i=1}^{n} \alpha_i h_i(x)$$

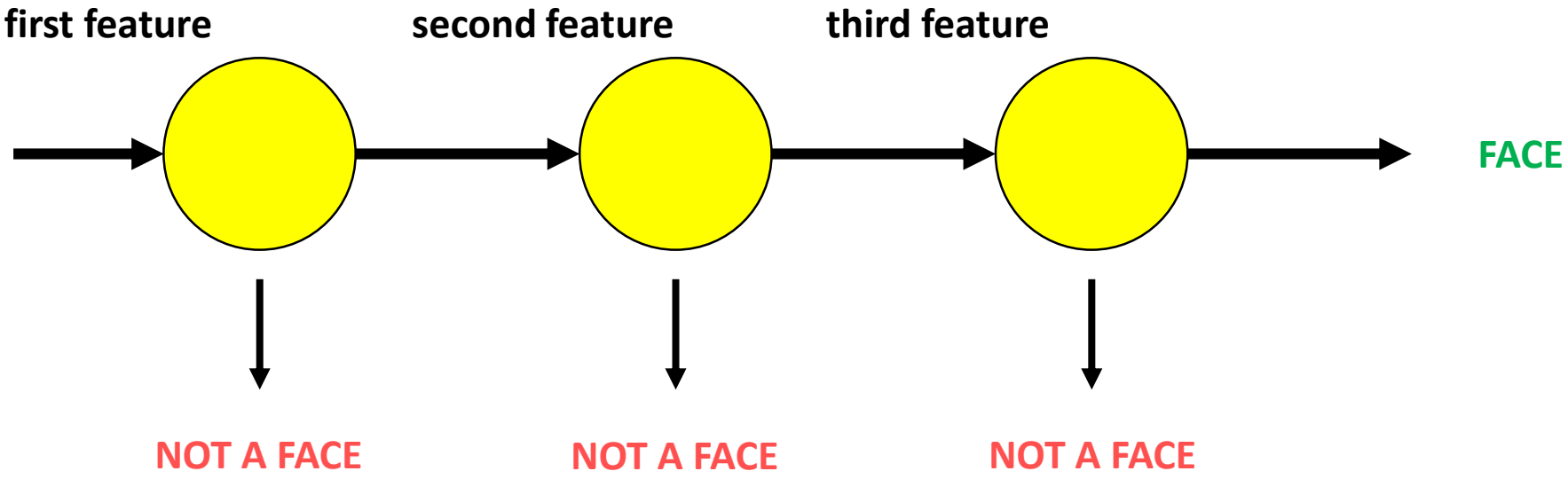
final **H(x)** model which is a strong classifier

every **h(x)** week learner make a prediction based on a single **Haar-feature**

THE h(x) CLASSIFIERS WITH HIGHER α VALUES ARE THE RELEVANT FEATURES !!!

→ if the window does not contain the most relevant features we can consider the next region on the image

Cascading


first feature
second feature
third feature
FACE
NOT A FACE
NOT A FACE
NOT A FACE