# Introduction to Machine Learning

R. Pastor-Satorras

Departament de Física
UPC

Eines Informàtiques Avançades / EIA

# What is Machine Learning (ML)?

## Aurélien Géron, 2024

The science (and art) of programming computers so they can *learn from data*

# What is Machine Learning (ML)?

## Aurélien Géron, 2024

The science (and art) of programming computers so they can *learn from data*

## Arthur Samuel, 1959

The field of study that gives computers the ability to learn without being explicitly programmed

# What is Machine Learning (ML)?

## Aurélien Géron, 2024

The science (and art) of programming computers so they can *learn from data*

## Arthur Samuel, 1959

The field of study that gives computers the ability to learn without being explicitly programmed

## Tom Mitchell, 1997

A computer program is said to learn from experience $E$ with respect to some task $T$ and some performance measure $P$, if its performance on $T$, as measured by $P$, improves with experience $E$

# The Working of ML

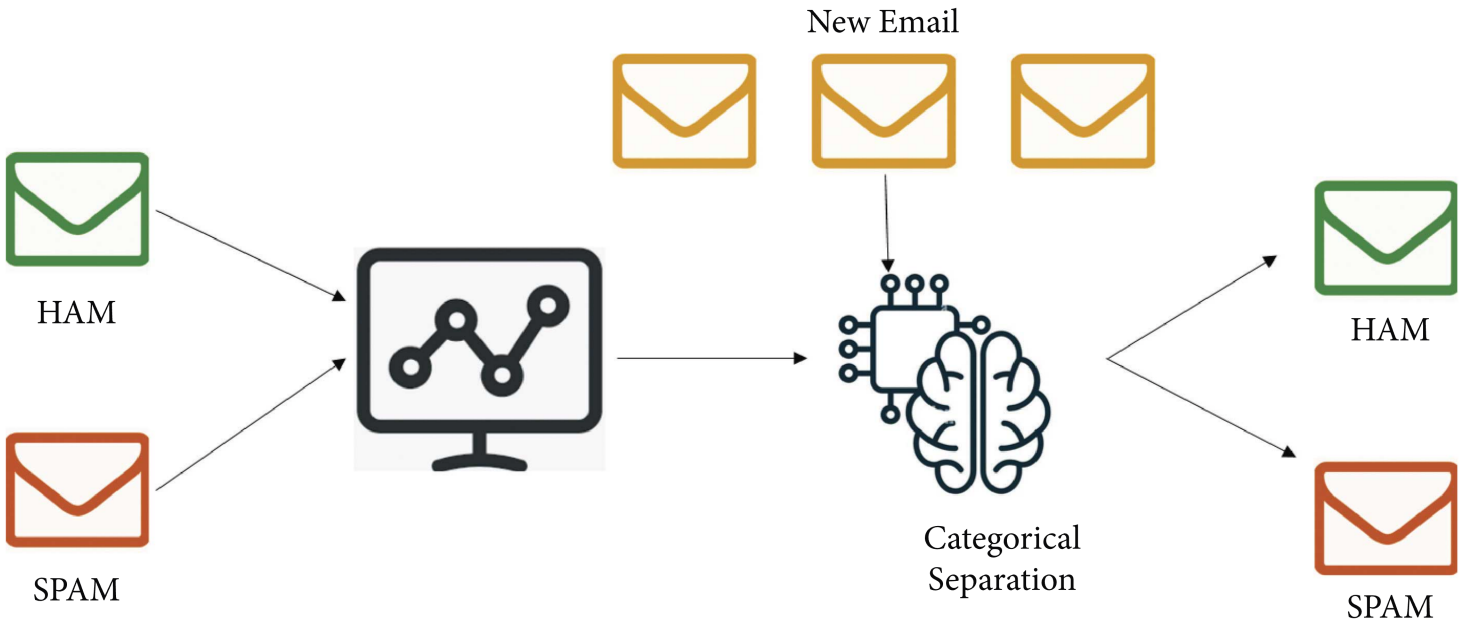
In simple terms:

- Find a statistical model (or mathematical formula) that, applied to some set of initial input data (the *training data*), produces some desired output result
- When applied to other input data, different from the training set, also generates correct outputs
  - Necessary condition is that the other input data has the same "statistical properties" than the training data
- Finding the statistical model usually implies the minimization of some *cost* or *objective function*
- Minimizing the objective function is usually hard, so efficient computational methods are usually required

# Examples of ML: Classification

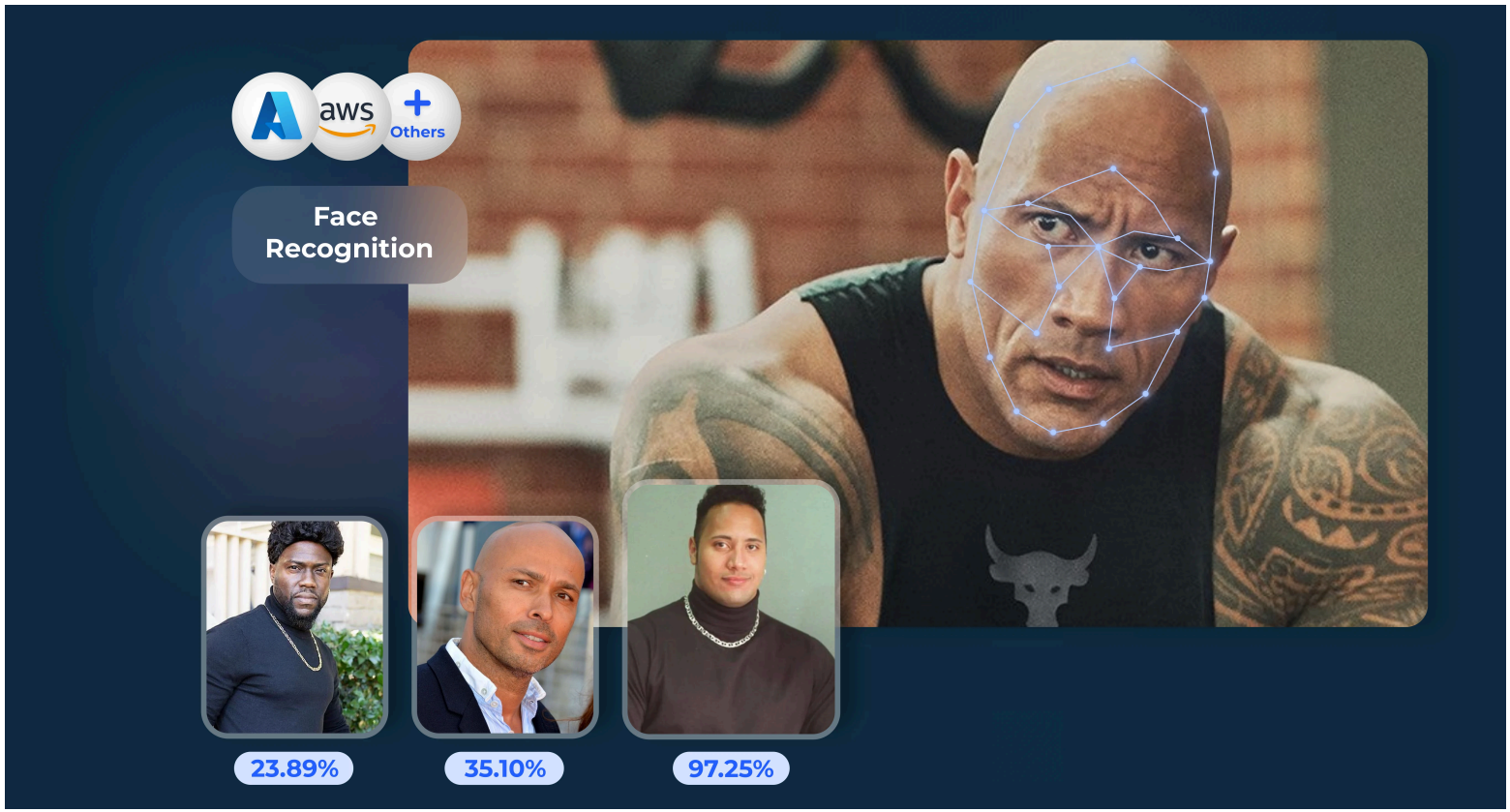
Learn to classify things into known classes (labels)

Eg: Spam detector



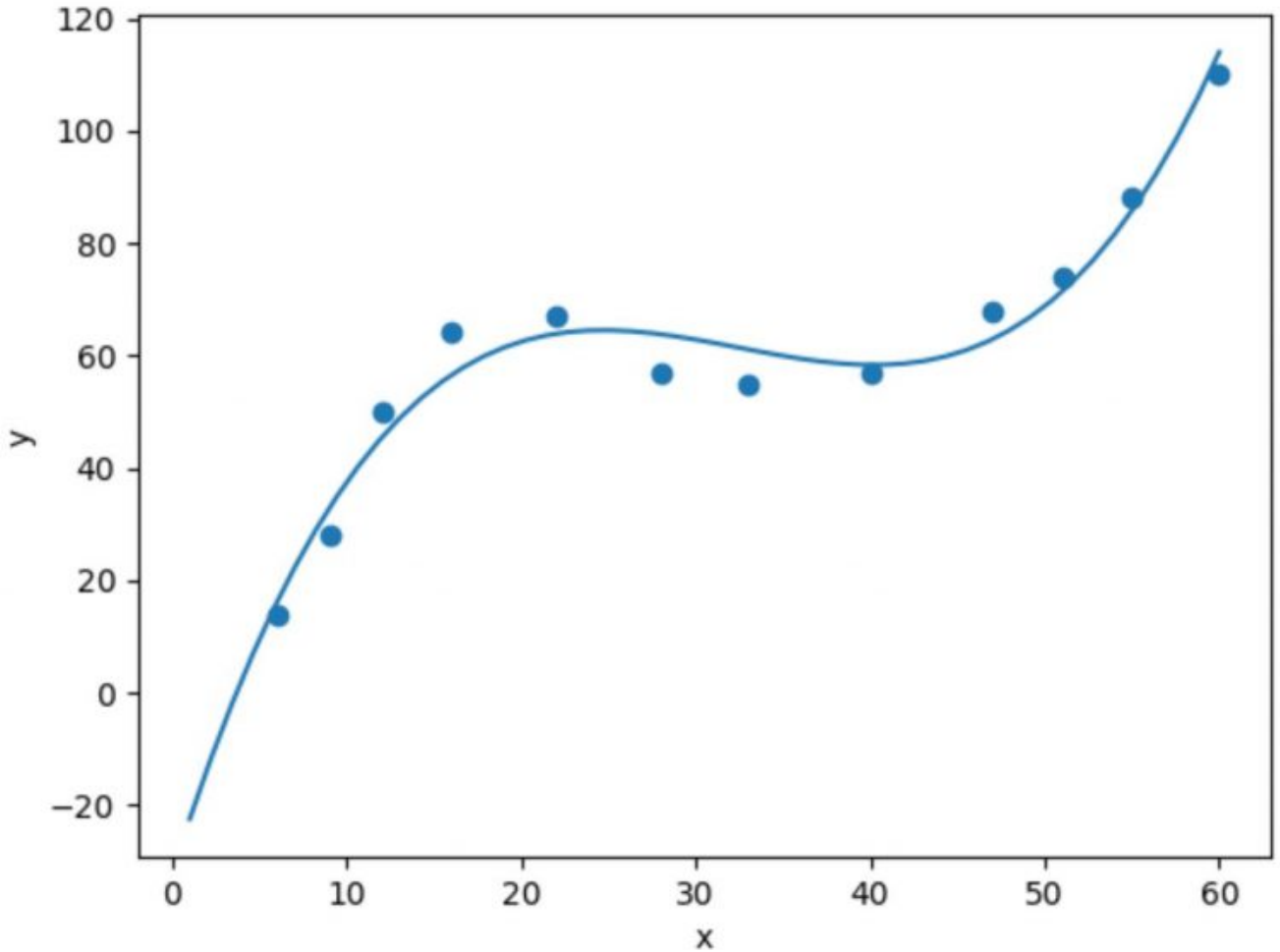
# Examples of ML: Face recognition

Learn to recognize a face from different angles and with some changes (Eg. Unlock you phone with face ID)



# Examples of ML: Regression

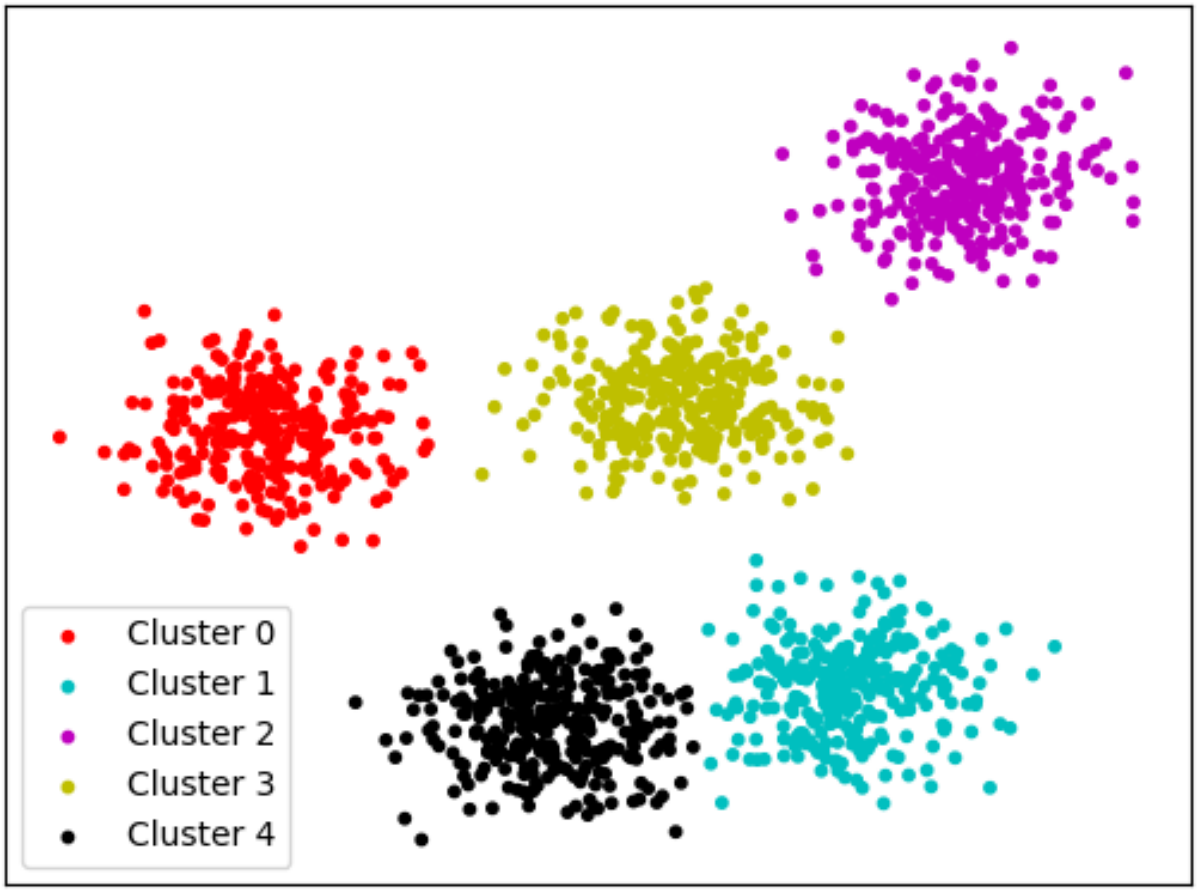
Predicting a numerical value from some examples



# Examples of ML: Clustering

Detect the presence of structures or categories in datasets

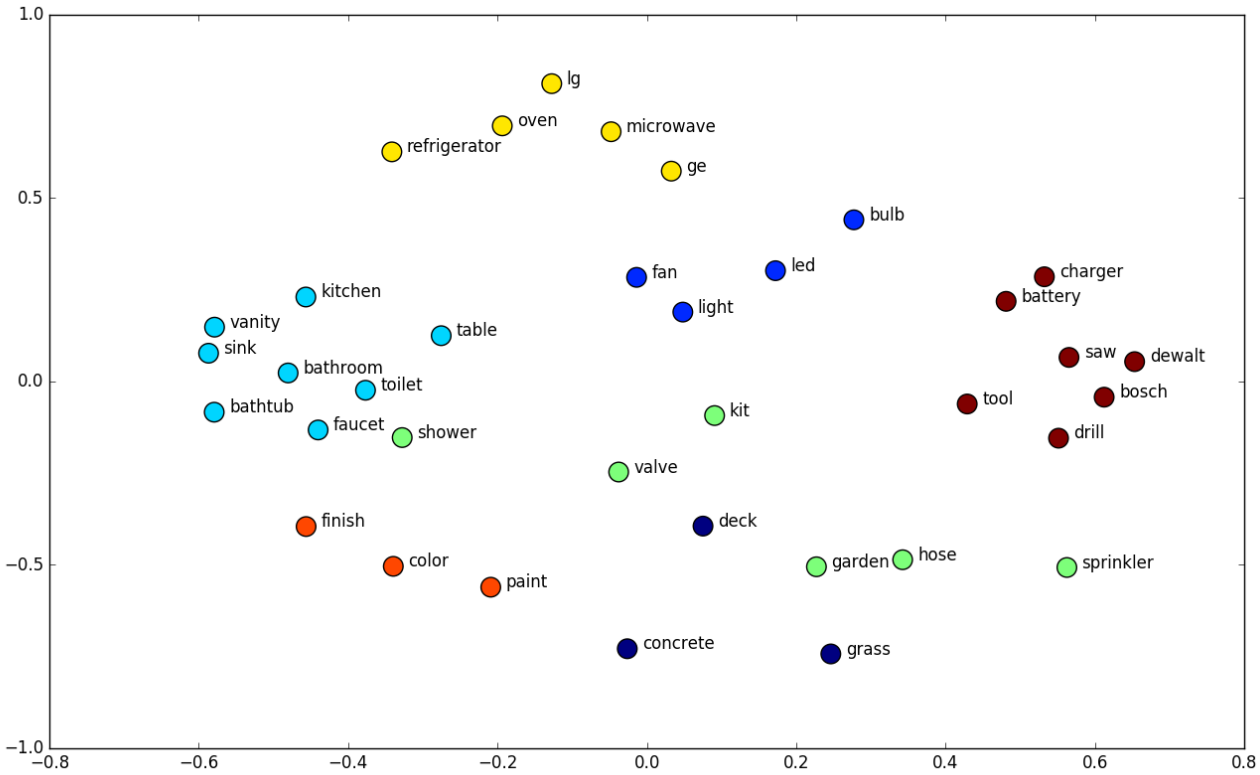
Kind of classification, but we don't know beforehand the classes



# Examples of ML: Embedding

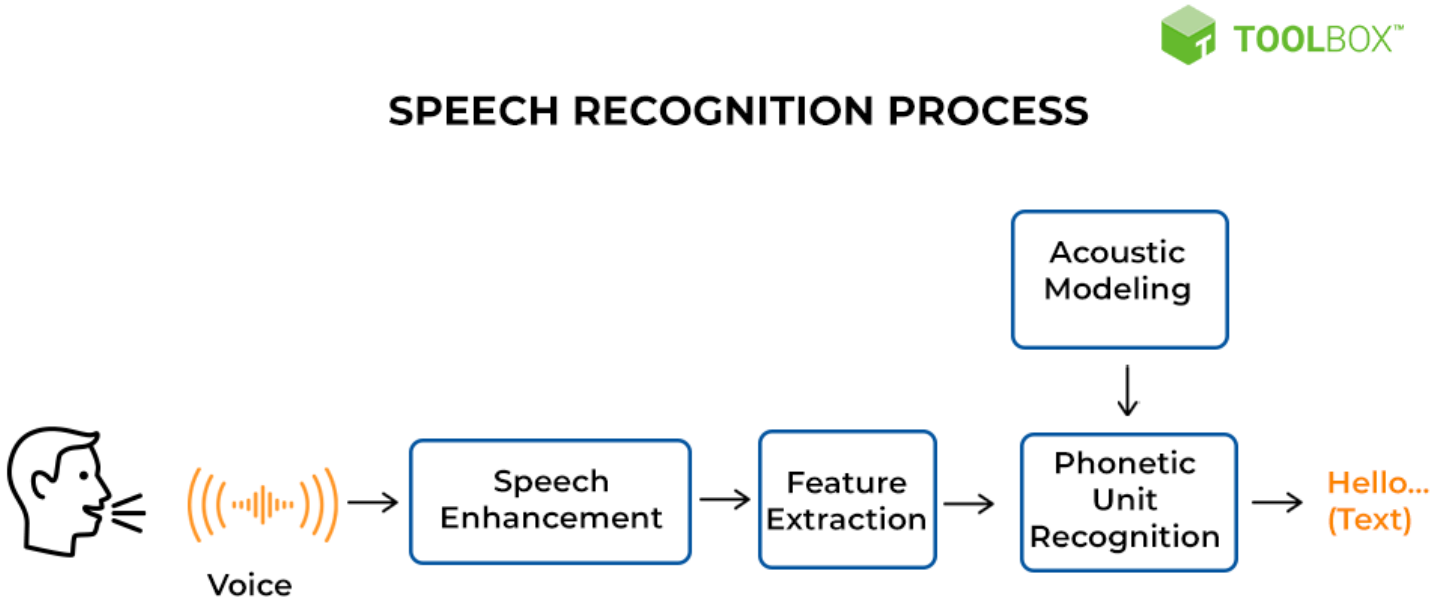
Learn a low-dimensional representation in Euclidean space of complex, non-geometric data, in such a way that similar datapoints are close to each other

Eg. Word embeddings



# Examples of ML: Speech recognition

Learn to identify the meaning of spoken words (Eg. Siri, Alexa)



# ML *is* Data

We need *data* to solve a problem with Machine Learnein

But we cannot use *any* kind of data

What computer algorithms like is data in terms of $D$-dimensional real-valued vectors, $\vec{x} = \{x^{(1)}, x^{(2)}, \ldots, x^{(D)}\}$, called *feature vectors*

Each *feature* describes one property of the data point.

Eg. for human beings:

- $x^{(1)} =$ Age
- $x^{(2)} =$ Weight
- $x^{(3)} =$ Height

A *dataset* is composed by collection of *feature vectors* $\{\vec{x}_i\}$, $i = 1, \ldots, N$

Additionally, a data point can be characterized by a *label*, $y_i$, containing information that characterizes in some way the data point

The label can be an integer, if it classifies the data point in a finite set of *classes*, a real number, a string of text, or any more complex structure

# Converting complex data into feature vectors

Some datasets are readily given in terms of feature vectors

Some other must be transformed

- Text (e-mails)
- Images
- Sound

There are different ways of perform this transformation, called *encoding*

# Encoding categorical data

Categories, usually in the form of strings of characters, that categorize the datapoints. Usually finite in number

We have different options:

- Ordinal encoding
  Assign to each category a number

| Original Encoding | Ordinal Encoding |
|---|---|
| Poor | 1 |
| Good | 2 |
| Very Good | 3 |
| Excellent | 4 |

# Encoding categorical data

- Ordinal encoding
  Problem: It implicitly assigns a distance, with sometimes can make sense, but sometimes doesn't

# Encoding categorical data

- Ordinal encoding
  Problem: It implicitly assigns a distance, with sometimes can make sense, but sometimes doesn't

| Original Encoding | Ordinal Encoding |
|---|---|
| Poor | 1 |
| Good | 2 |
| Very Good | 3 |
| Excellent | 4 |

| Original Encoding | Ordinal Encoding |
|---|---|
| Cat | 1 |
| Dog | 2 |
| Turtle | 3 |
| Fish | 4 |

# Encoding categorical data

- One-hot encoding
  Better option
  Map each category to a vector that contains 0's and 1's, denoting the presence of the feature or not

| Pet |
|---|
| Cat |
| Dog |
| Turtle |
| Fish |

→

| Cat | Dog | Turtle | Fish |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 |

# Encoding categorical data

Many other possibilities:

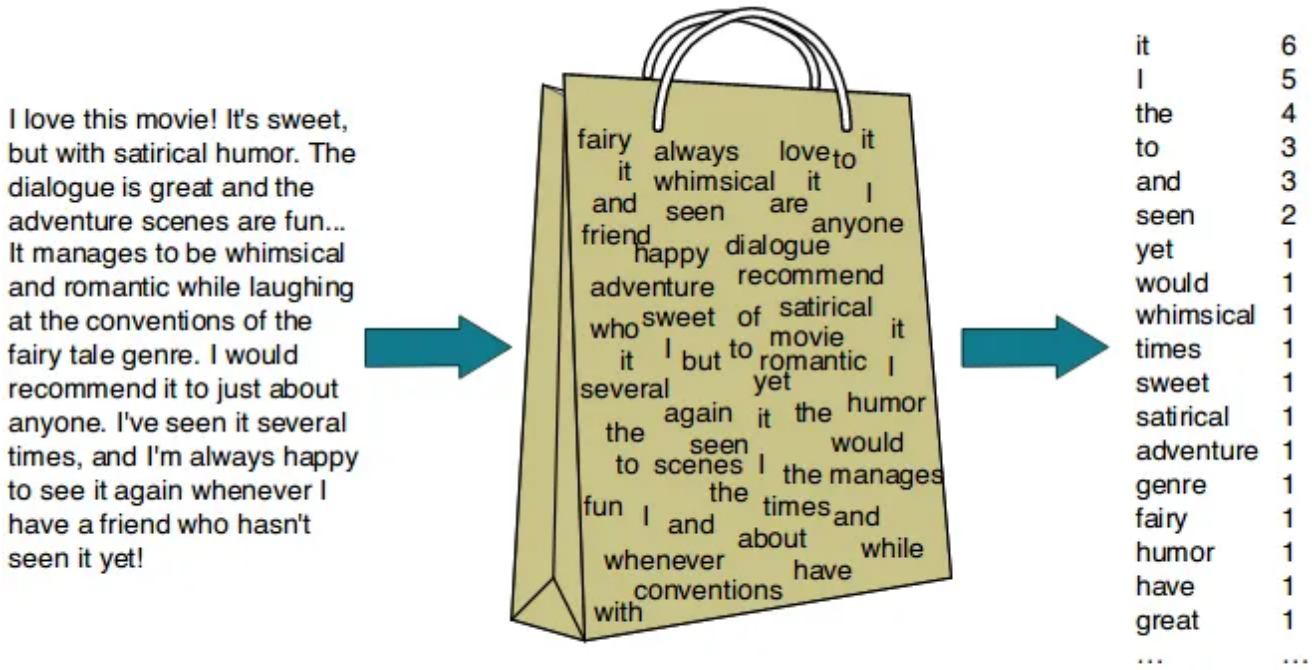
- Binary encoding: Use ordinal encoding and transform it to binary code, using the binary digits as separate features
- Hash encoding: Transform the string of characters of the categorical data into a hash string, using a MD5 algorithm
- Frequency encoding: If the categorical data appear many times, use their frequency as encoding
  - If the word "apple" appears 50 times, "banana" 30 times, and "cherry" 25 times, we replace:

| fruit | frequency-encoded fruit |
|---|---|
| apple | 50 |
| banana | 30 |
| cherry | 20 |

# Encoding text

## Bag of Words

- Take a dictionary of, say, $N = 10000$ ordered words
- For a given text we assign a feature vector of dimension $D = N$ as follows
  - If the text contains the $n$-th word of the dictionary, the feature $x^{(n)} = 1$
  - Otherwise, set $x^{(n)} = 0$.
- Each word is thus assigned a feature vector of dimension $D = 10000$



# Encoding text: Bag-of-Words

Example: Dictionary of 8 words:

- "a" : 1
- "be" : 2
- "case" : 3
- "man" : 4
- "nice" : 5
- "not" : 6
- "or" : 7
- "to" : 8

Text = "To be or not to be"

$\vec{x}$ has $D = 8$ dimensions

Different words: "to":8, "be":2, "or":7, "not":6

$\vec{x} = \{0, 1, 0, 0, 0, 1, 1, 1\}$

# Encoding images

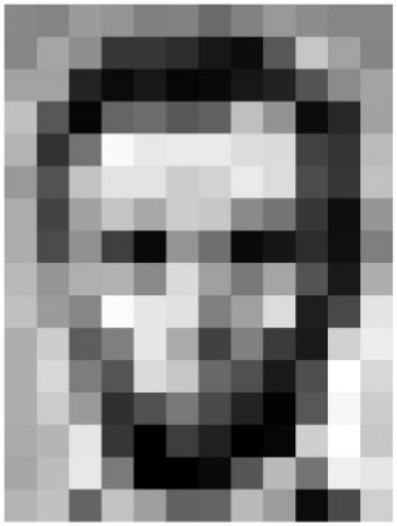
Images can be defined in a low resolution raster format, made up of pixels

Each pixel is assigned a position and a color

The color can be encoded in a single number, real or integer

An image of size $N \times M$ pixels can thus be defined by a matrix of $N \times M$

Flattening the matrix, we can assign to each image a feature vector of $N \times M$ dimensions



| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 157 | 153 | 174 | 168 | 150 | 152 | 129 | 151 | 172 | 161 | 155 | 156 |
| 155 | 182 | 163 | 74 | 75 | 62 | 33 | 17 | 110 | 210 | 180 | 154 |
| 180 | 180 | 50 | 14 | 34 | 6 | 10 | 33 | 48 | 106 | 159 | 181 |
| 206 | 109 | 5 | 124 | 131 | 111 | 120 | 204 | 166 | 15 | 56 | 180 |
| 194 | 68 | 137 | 251 | 237 | 239 | 239 | 228 | 227 | 87 | 71 | 201 |
| 172 | 105 | 207 | 233 | 233 | 214 | 220 | 239 | 228 | 98 | 74 | 206 |
| 188 | 88 | 179 | 209 | 185 | 215 | 211 | 158 | 139 | 75 | 20 | 169 |
| 189 | 97 | 165 | 84 | 10 | 168 | 134 | 11 | 31 | 62 | 22 | 148 |
| 199 | 168 | 191 | 193 | 158 | 227 | 178 | 143 | 182 | 106 | 36 | 190 |
| 205 | 174 | 155 | 252 | 236 | 231 | 149 | 178 | 228 | 43 | 95 | 234 |
| 190 | 216 | 116 | 149 | 236 | 187 | 86 | 150 | 79 | 38 | 218 | 241 |
| 190 | 224 | 147 | 108 | 227 | 210 | 127 | 102 | 36 | 101 | 255 | 224 |
| 190 | 214 | 173 | 66 | 103 | 143 | 96 | 50 | 2 | 109 | 249 | 215 |
| 187 | 196 | 235 | 75 | 1 | 81 | 47 | 0 | 6 | 217 | 255 | 211 |
| 183 | 202 | 237 | 145 | 0 | 0 | 12 | 108 | 200 | 138 | 243 | 236 |
| 195 | 206 | 123 | 207 | 177 | 121 | 123 | 200 | 175 | 13 | 96 | 218 |


| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 157 | 153 | 174 | 168 | 150 | 152 | 129 | 151 | 172 | 161 | 155 | 156 |
| 155 | 182 | 163 | 74 | 75 | 62 | 33 | 17 | 110 | 210 | 180 | 154 |
| 180 | 180 | 50 | 14 | 34 | 6 | 10 | 33 | 48 | 106 | 159 | 181 |
| 206 | 109 | 5 | 124 | 131 | 111 | 120 | 204 | 166 | 15 | 56 | 180 |
| 194 | 68 | 137 | 251 | 237 | 239 | 239 | 228 | 227 | 87 | 71 | 201 |
| 172 | 105 | 207 | 233 | 233 | 214 | 220 | 239 | 228 | 98 | 74 | 206 |
| 188 | 88 | 179 | 209 | 185 | 215 | 211 | 158 | 139 | 75 | 20 | 169 |
| 189 | 97 | 165 | 84 | 10 | 168 | 134 | 11 | 31 | 62 | 22 | 148 |
| 199 | 168 | 191 | 193 | 158 | 227 | 178 | 143 | 182 | 106 | 36 | 190 |
| 205 | 174 | 155 | 252 | 236 | 231 | 149 | 178 | 228 | 43 | 95 | 234 |
| 190 | 216 | 116 | 149 | 236 | 187 | 86 | 150 | 79 | 38 | 218 | 241 |
| 190 | 224 | 147 | 108 | 227 | 210 | 127 | 102 | 36 | 101 | 255 | 224 |
| 190 | 214 | 173 | 66 | 103 | 143 | 96 | 50 | 2 | 109 | 249 | 215 |
| 187 | 196 | 235 | 75 | 1 | 81 | 47 | 0 | 6 | 217 | 255 | 211 |
| 183 | 202 | 237 | 145 | 0 | 0 | 12 | 108 | 200 | 138 | 243 | 236 |
| 195 | 206 | 123 | 207 | 177 | 121 | 123 | 200 | 175 | 13 | 96 | 218 |

# Types of Machine Learning

- Supervised Learning
- Unsupervised Learning
- Semi-supervised Learning
- Reinforcement Learning

# Supervised Learning

Dataset is a collection of labeled datapoints $\{\vec{x}_i, y_i\}$

The goal is to use the dataset to build a model that takes as input a feature vector $\vec{x}_i$ and produces an output that helps to decide the value of the label $y_i$

Information comes from the interplay between feature vectors and their associated labels

Example: Spam mail classification

# Dividing the data

In supervised (and general) learning, the dataset is usually divided into three different sets:

- *Training set*
- *Validation set*
- *Test set*

The three sets are chosen at random from the dataset with a typical proportion 70%/15%/15% (or 90%/5%/5% if your dataset is really very large)

- The training set is used to build the stochastic model
- The validation set is used to select the appropriate parameters (*hyper-parameters*) of the learning algorithm, if there are some that can be tuned by hand
- The test set is finally used to check that the validated model really works

# Unsupervised Learning

Dataset is a collection of unlabeled datapoints $\{\vec{x}_i\}$

The goal is to use the dataset to build a model that can solve some practical problem

An example is *clustering*, consisting in partitioning a dataset into different groups (clusters) that share similar properties or characteristics

Information comes from the feature vectors alone

# Semi-supervised Learning

Dataset is a collection of labeled and unlabeled datapoints, usually much more unlabeled than labeled

The goal is the same as in supervised learning, with the hope that the unlabeled datapoints will help the learning algorithm to produce a better model

# Reinforcement Learning

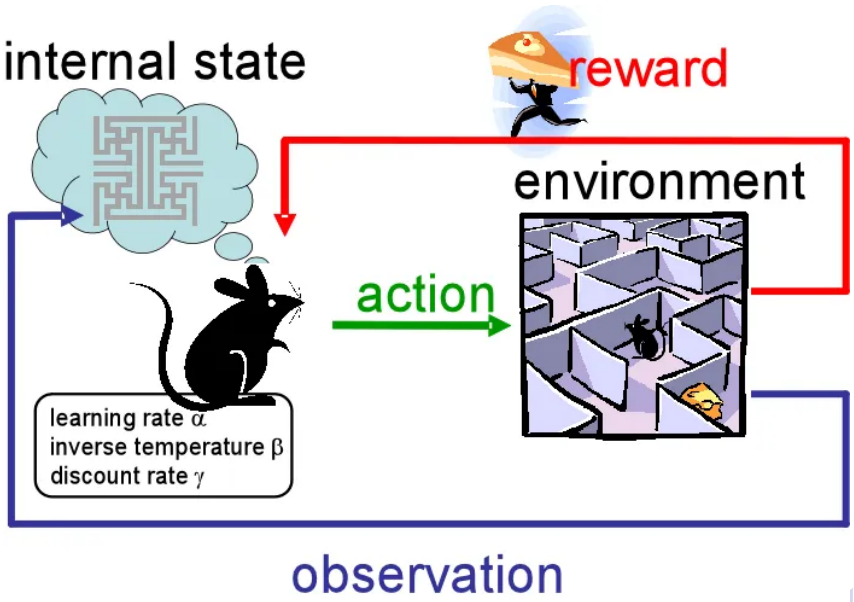
In Reinforcement Learning the machine evolves in a set of different states

The machine can execute *actions* in every state, that produce a *reward* and can move the machine to another state

The goal is to learn a *policy*: A function that takes an state as input and outputs the next optimal action to perform in that state

The action is *optimal* it maximizes the expected average reward

Example: Solve problems where decision-making is sequential, such as game-playing



# Model-based vs Instance-based learning

In *Model Based learning* (MB), the algorithm learns a statistical model (with some mathematical formulation) that represents the statistical underlying properties/patterns of the training data. The model can be used later to make predictions on new input data

In *Instance Based learning* (IB) the algorithm learns the training data, keeping it in memory, to discover relationships between the training datapoints. Predictions can be made by comparing new input data with the stored training data

Differences:

- **Generality**: MB learns a generalizable model; IB learns the training datapoints
- **Scalability**: IB requires much more memory: not as scalable as MB
- **Interpretability**: MB is easier to interpret in terms of a mathematical model

# The simplest ML algorithm: Regression

Supervised, model based learning technique used to predict continuous values

Dataset is a set of labeled datapoints $\{\vec{x}_i, y_i\}$

Labels are real numbers

The scope is to build a model composed by a linear combination of functions of the features of the training set

$$f(\vec{x}; b_1, b_2, \ldots, b_M) = \sum_{j=1}^{M} b_j \; g_j(\vec{x}) \tag{1}$$

where $g_j(\vec{x})$ are functions of the feature vector

The goal is to learn the best values $b_j^*$ that can predict with accuracy the real-valued labels of the feature vectors

$$f(\vec{x}_i; b_1^*, b_2^*, \ldots, b_M^*) \simeq y_i \tag{2}$$

# The simplest ML algorithm: Regression

The goal can be achieved by minimizing the *distance* between $f(\vec{x}_i; b_1^*, b_2^*, \ldots, b_M^*)$ and $y_i$

To do so we define the *cost function*

$$Q = \frac{1}{N} \sum_{i=1}^{N} [f(\vec{x}_i; b_1, b_2, \ldots, b_M) - y_i]^2 \tag{3}$$

(least squares minimization)

Analytically, this implies to solve the set of equations

$$\left. \frac{\partial Q}{\partial b_i} \right|_{b_j = b_j^*} = 0 \tag{4}$$

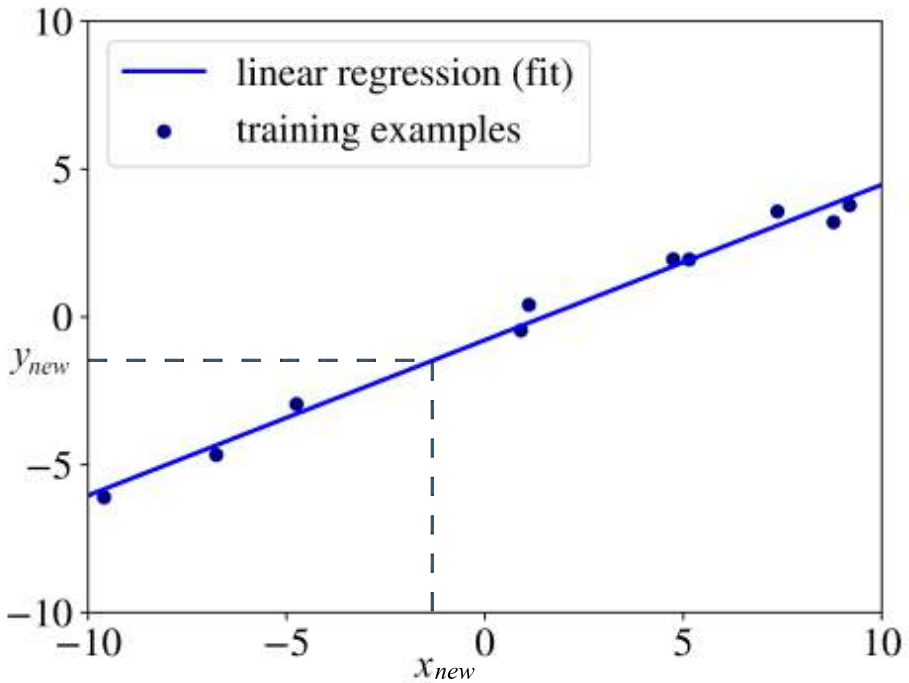
When there are many parameters $b_i$, the minimization becomes complex and it must be done using numerical techniques

# Linear regression

Linear regression takes place when the feature vectors are one-dimensional and we use the fitting function of the form

$$f(x; b_1, b_2) = b_2 x + b_1 \tag{5}$$

Classical fit to a linear function, which can be analytically solved



# Underfitting and overfitting

In some cases, a more complex function is required

$$f(x; b_1, b_2, \ldots, b_M) = \sum_{i=1}^{M} b_i \, x^{i-1} \tag{6}$$

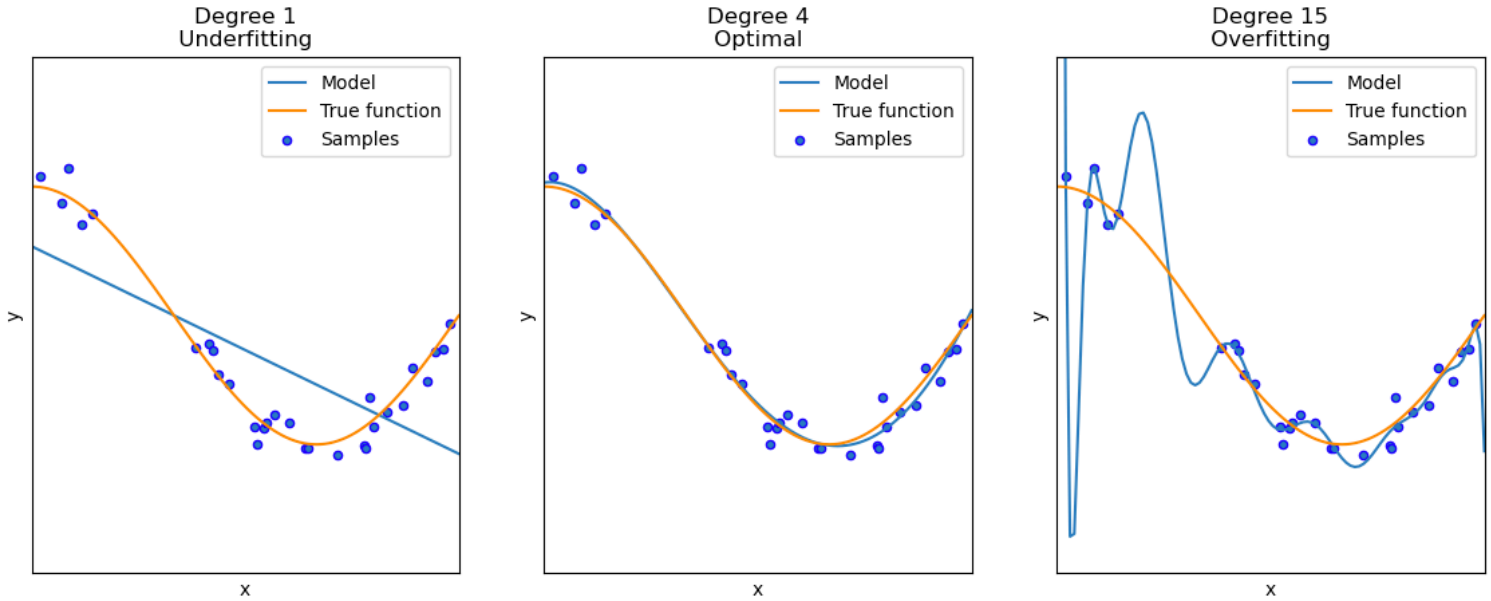
$M$ is now a *hyper-parameter*

# Underfitting and overfitting

In some cases, a more complex function is required

$$f(x; b_1, b_2, \ldots, b_M) = \sum_{i=1}^{M} b_i \, x^{i-1} \tag{6}$$

$M$ is now a *hyper-parameter*

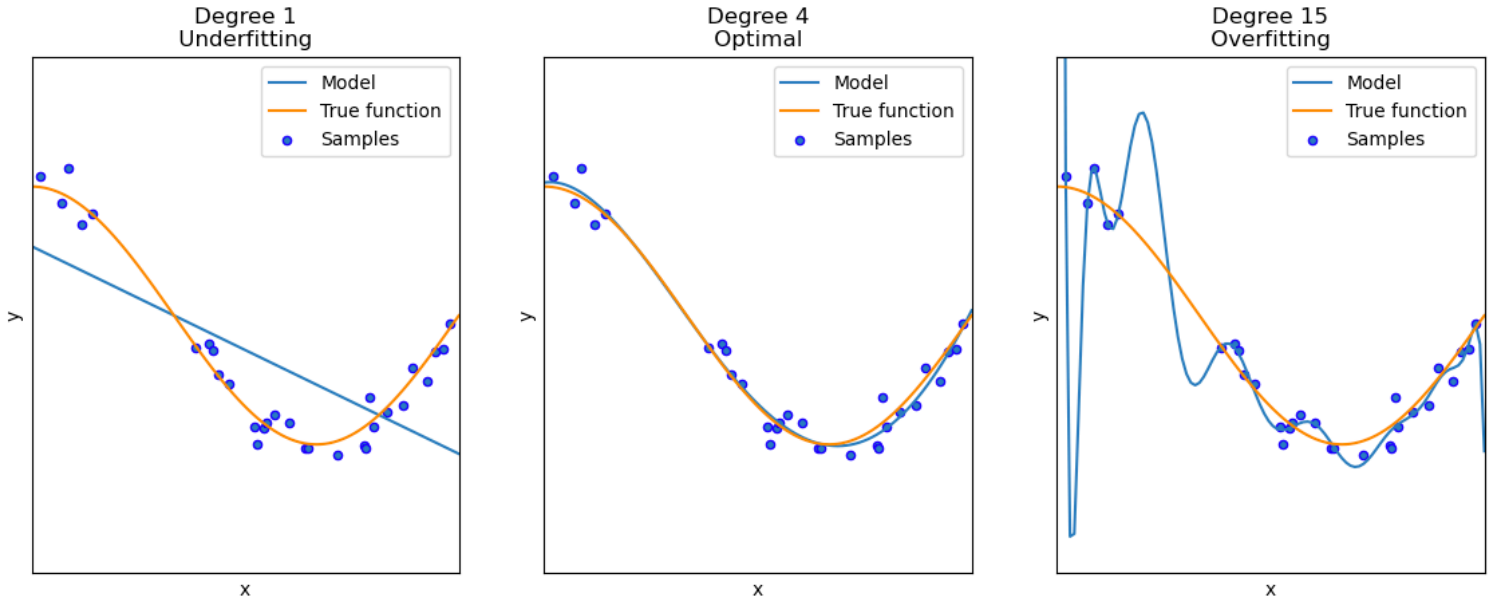


# Underfitting and overfitting

In some cases, a more complex function is required

$$f(x; b_1, b_2, \ldots, b_M) = \sum_{i=1}^{M} b_i \, x^{i-1} \tag{6}$$

$M$ is now a *hyper-parameter*




Judiciously choose number of terms to balance between *underfitting* and *overfitting*

# Overfitting

Overfitting (or *high variance problem*) is a typical problem of supervised learning, in which the model predicts very well the training dataset but poorly the test dataset

Reason for overfitting

- The model is too complex (Eg, too many terms in a regression)
- There are many features but a small number of training examples

Solutions

- Use a simpler model
- Reduce the dimensionality of the features (more on this later)
- Add more training datapoints
- *Regularize* the model

# Regularization

*Regularization* consists in modifying the model (through the cost function) adding a penalty for complex models

In the case of regression, the cost function is

$$Q = \frac{1}{N}\sum_{i=1}^{N}[f(\vec{x}_i;\{b_j\}) - y_i]^2$$

The *L2 regularization*, defined by the new cost function

$$Q_r = \alpha||\vec{b}||^2 + \frac{1}{N}\sum_{i=1}^{N}[f(\vec{x}_i;\{b_j\}) - y_i]^2$$

forces many of the components of $\vec{b}$ to be zero, thus simplifying the model (*feature selection* of the most relevant features)

$\alpha$ is another *hyper-parameter*, that can be tuned for better performance

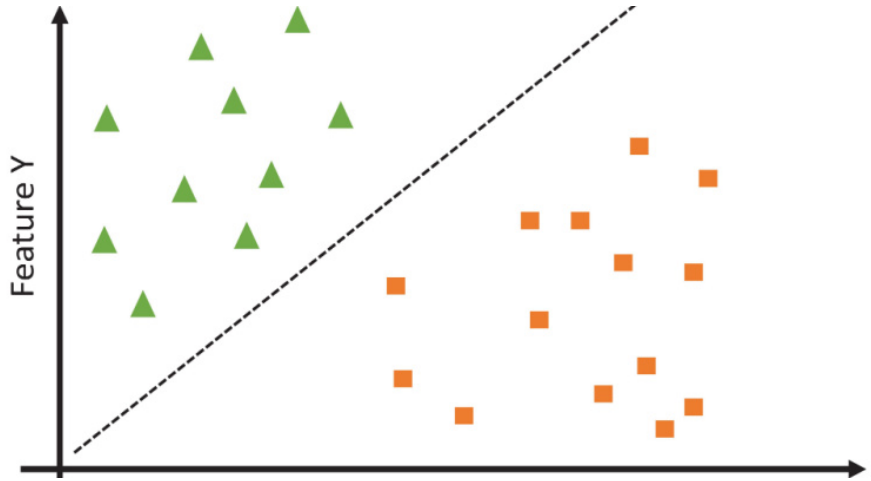
# Support Vector Machine (SVM)

Supervised, model based learning technique used to classify datapoints according to their label

Dataset is a set of labeled datapoints $\{\vec{x}_i, y_i\}$

Labels take discrete values. Ex: ["Spam", "Not Spam"] = { +1, -1 }

For labels taking two values, the *linear* algorithm places the feature vectors in a $D$-dimensional space and draws a hyper-plane that separates the datapoints with positive label from those with negative labels



# Support Vector Machine (SVM)

The equation of the hyper-plane is

$$\vec{w} \cdot \vec{x} - b = 0 \tag{7}$$

where $\vec{w}$ is a real-valued vector

The prediction for the label is

$$y = \text{sign}(\vec{w} \cdot \vec{x} - b) \tag{8}$$

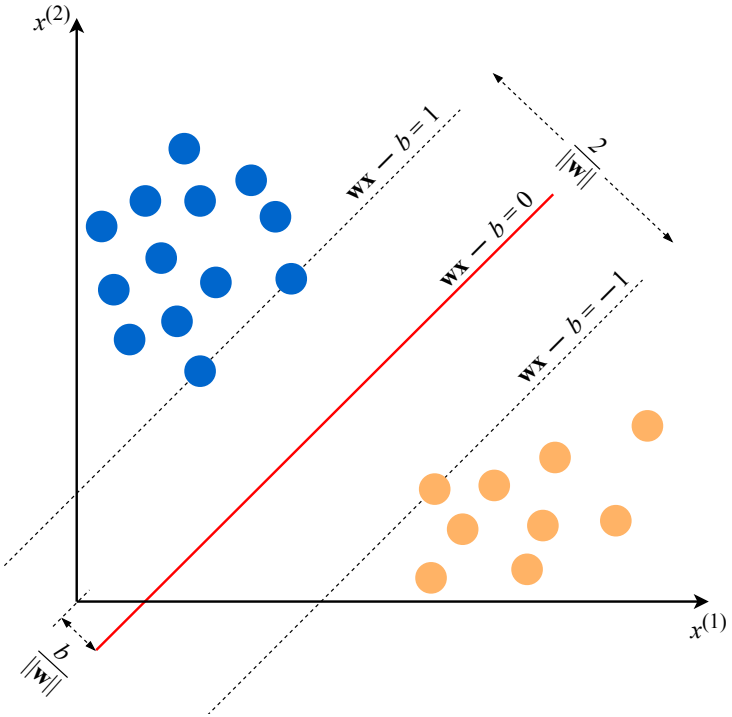
The goal of the algorithm is to learn the optimal values $\vec{w}^*$ and $b^*$, subject to the constraint, for maximal differentiation between classes

$$\vec{w} \cdot \vec{x}_i - b \geq +1 \quad \text{if} \quad y_i = +1$$
$$\vec{w} \cdot \vec{x}_i - b \leq -1 \quad \text{if} \quad y_i = -1$$

# Support Vector Machine (SVM)

The optimal values $\vec{w}^*$ and $b^*$ are such that hyper-plane separates positives datasets from negative ones by the largest margin

The margin is the distance between the closest examples of the two classes




Cost function

$$Q = ||\vec{w}|| = \sqrt{\sum_{j=1}^{D} [w^{(j)}]^2}$$

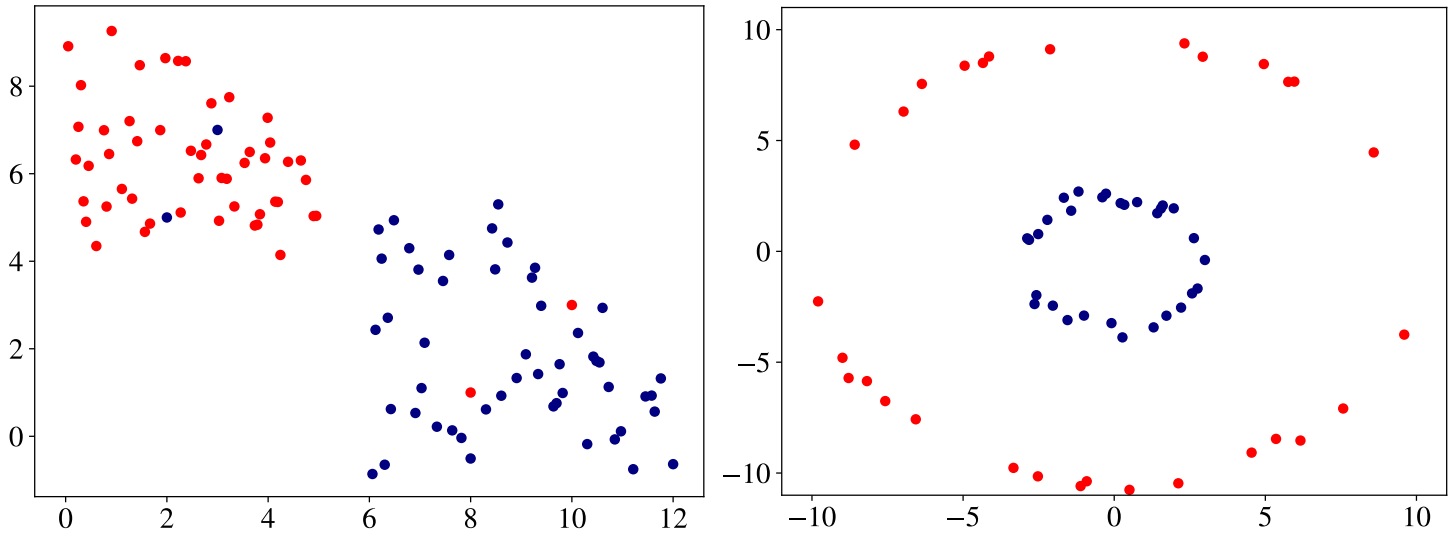
Subject to the constraint

$$y_i(\vec{w} \cdot \vec{x}_i) \geq +1$$

Minimization of the cost function, subject to the constraints, performed numerically

# Non-linear Support Vector Machine

In some cases, a hyper-plane cannot separate the classes adequately, either due to noise or to a structure inherently non-linear




In this case, one trick (*kernel trick*) consists in transforming the original $D$-dimensional feature space in a space of dimension $D' > D$, with the hope that in higher dimension it can become linearly separable

# Performance assessment

The performance of SVM, and in general supervised learning models, constructed with a training set, can be assessed by applying the model to the test set and checking how it works

In the results of the test set, we can observe four types of results (with labels $\{+1, -1\}$)

- **True positive** (TP) : A test datapoint $+1$ classified as $+1$
- **True negative** (TN) : A test datapoint $-1$ classified as $-1$
- **False positive** (FP) : A test datapoint $-1$ classified as $+1$
- **False negative** (FN) : A test datapoint $+1$ classified as $-1$

Think of results of a serological test: $+1$ meaning infected, -1 meaning non infected

The performace of the learning model can be represented in different ways

# Confusion matrix

Matrix representation of the results

$$\mathsf{C} = \begin{pmatrix} \mathrm{TP} & \mathrm{FN} \\ \mathrm{FP} & \mathrm{TN} \end{pmatrix}$$

Can be extended to the case in which there are $M$ classes.

Matrix $\mathsf{C} = \{C_{ij}\}$ where $C_{ij}$ is the number of test datapoints belonging to the class $i$ that were assigned to the class $j$

# Precision/Recall

*Precision* is the ratio of the correct positive predictions with respect to the total number of positive predictions

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

*Recall* is the ratio of the correct positive predictions with respect to the total number of positive examples in the test dataset

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

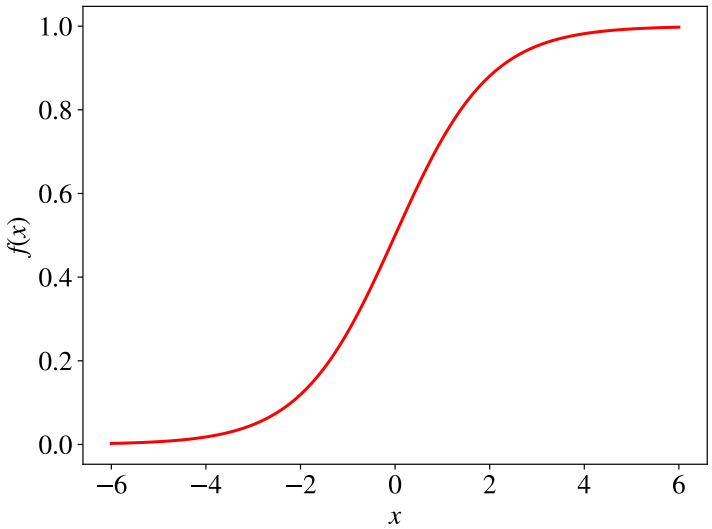
Ideally we want high precision and high recall, but it is usually impossible to achieve both at the same time

In a trade off, we usually prefer high precision (no legit mail is classified as spam) and allow for low recall (we tolerate some spam)

# Logistic Regression

Supervised, model based learning technique with the same goal as SVM, aiming a classifying datapoints according to a binary label $y_i \in \{1, 0\}$

Nothing to do with linear (or non-linear) regression. The names stems from the use of the *logistic function* in the creation of the cost function




Logistic function

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

The output is interpreted as the probability of the label being 1 (if it is higher than the threshold 0.5)

# Logistic Regression

Logistic regression model

$$f(x; \vec{w}, b) = \frac{1}{1 + e^{-(\vec{w} \cdot \vec{x} + b)}}$$

After learning the optimal parameters $\vec{w}^*$ and $b^*$, the model $f(x; \vec{w}^*, b^*)$ gives the probability that the feature vector $\vec{x}$ has a 1 label

Instead of minimizing a cost function, we maximize the *likelihood* of the training set

$$L = \prod_{i=1}^{N} f(x_i; \vec{w}, b)^{y_i} [1 - f(x_i; \vec{w}, b)]^{(1-y_i)}$$

Alternatively, the *log-likelihood* (easier to compute)

$$\mathrm{Log}L = \ln L = \sum_{i=1}^{N} [y_i \ln f(x_i; \vec{w}, b) + (1 - y_i) \ln(1 - f(x_i; \vec{w}, b))]$$

Purely numerical evaluation

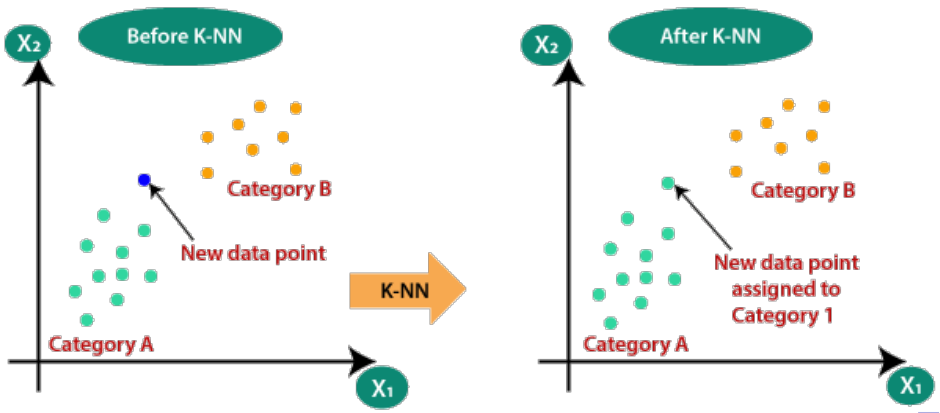
# $k$-Nearest Neighbors (kNN)

The $k$-Nearest Neighbors (kNN) is an unsupervised, instance based classification technique that classifies datapoints according to the similarity of their feature vectors or labels

The kNN algorithms stores the entire training database in memory as a reference

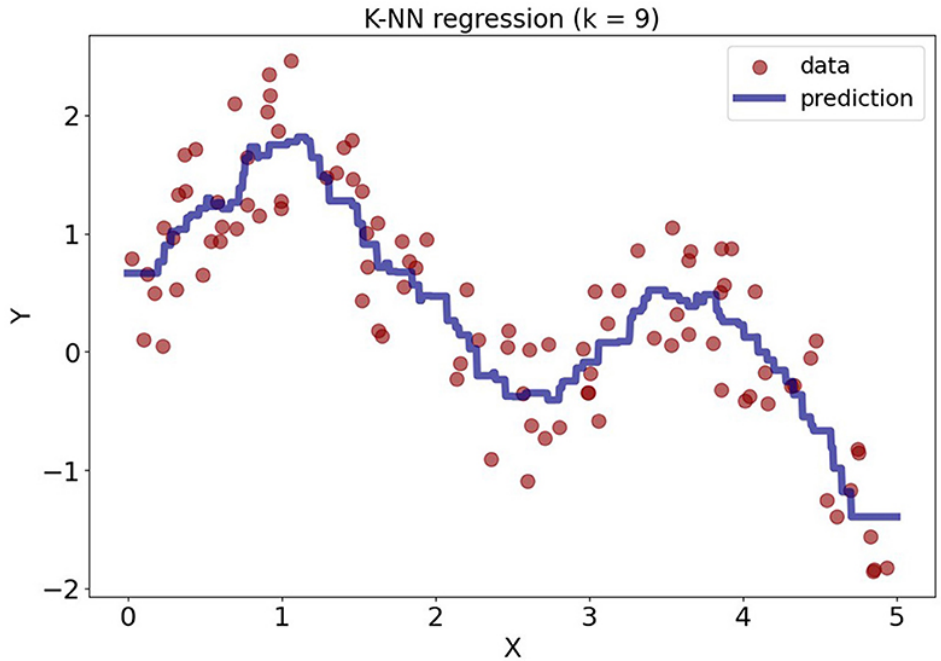
When making a prediction for a new input data, it computes the distance of the new input point to all the training examples, at the level of feature vectors

The algorithm identifies the $K$ nearest neighbors of the new input data and assigns to it the most common class label among the $K$ neighbors



# kNN as a regressor

kNN can be used as a regressor, assigning a value to a new datapoint that is the average value of the closest points used for training



# The Curse of Dimensionality

In kNN and other algorithms, we must measure the distance between feature vectors

The usual option is the Euclidean distance

$$d(\vec{x}_i, \vec{x}_j) = \sqrt{\sum_{n=1}^{D} (x_i^{(n)} - x_j^{(n)})^2}$$

The problem is when the feature vectors have a large dimension $D$. In this case, Euclidean distance become less meaningful, with the difference in distance between datapoints becoming negligible

- *The Curse of Dimensionality*

# Cosine Similarity

When data has a large dimension, a popular (negative) distance is the *cosine similarity*

$$s(\vec{x}_i, \vec{x}_j) = \frac{\sum_{n=1}^{D} x_i^{(n)} x_j^{(n)}}{\sqrt{\sum_{n=1}^{D} (x_i^{(n)})^2} \sqrt{\sum_{n=1}^{D} (x_j^{(n)})^2}}$$

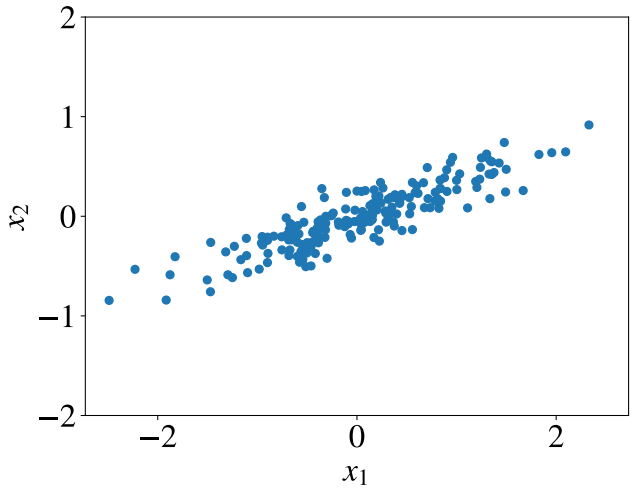
i.e. the cosine of the angle formed by the feature vectors $\vec{x}_i$ and $\vec{x}_j$

- Negative distance:
  Close vectors form a small angle and thus the cosine similarity is close to 1; Vectors pointing in opposite directions form an angle close to $\pi$, and this the cosine similarity is close to -1
- To use the cosine similarity as a distance we have to multiply it by -1

# Dimensional reduction

In real world applications, datasets can have feature vectors with many features, up to thousands or millions. This can present a problem in the efficiency of learning algorithms

In some cases, the features can have redundant information, in the sense that most of information (*variance*) of the data lies in a lower dimensional space




A lower dimensional approximation can be sufficient to represent the data

In this sense, dimensional reduction is used as a *preprocessing* technique to reduce computational complexity

# Dimensional reduction

*Dimensional reduction* is a procedure to reduce the number of features in a dataset, while still keeping as much information as possible of the datapoints

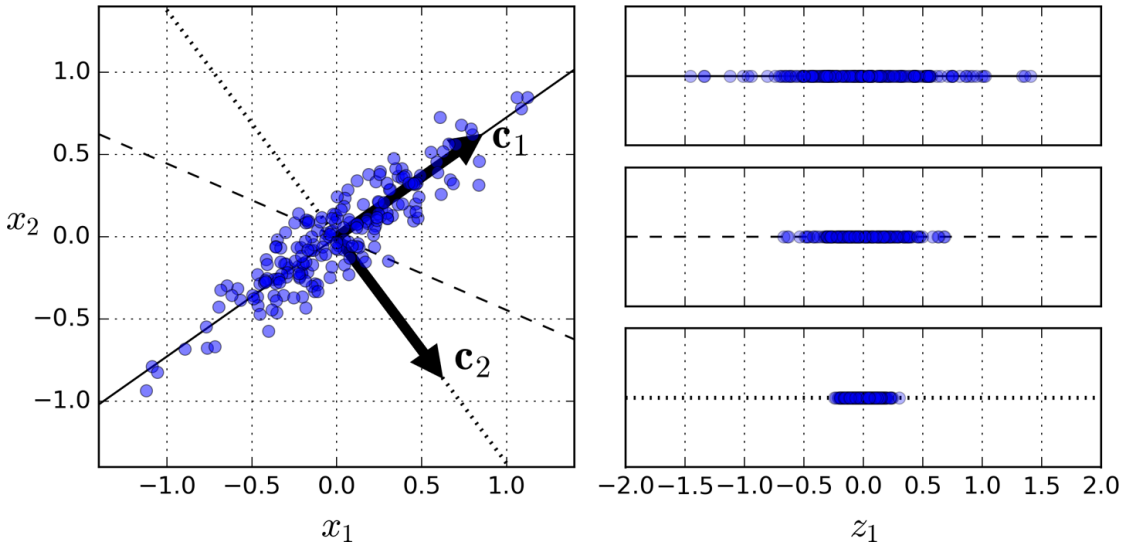
Several techniques are available for dimensional reduction.

Simplest and oldest one:

- *Principal component analysis* (PCA)

# Principal Component Analysis (PCA)

PCA is a model-based, linear transformation that tries to find the hyperplane that lies closest to the data, and then projects the data onto it




For $D = 2$, the hyperplane will be a line. For each line, the projection of the data will be different

PCA will identify the line in which the variance of the projected data is largest, and also a second, perpendicular to the first one, that accounts for the remaining variance

# Principal Component Analysis (PCA)

In $D$ dimensions, PCA finds the first *principal component* as the line in the $D$ dimensional space that best approximates the data in the least-squares sense

The second principal component is defined as the line that best approximates the data after we subtract the first principal component of the dataset

Iteratively, we can define the $k$-th principal component as the line that best fits data after subtracting the $k-1$-th principal component

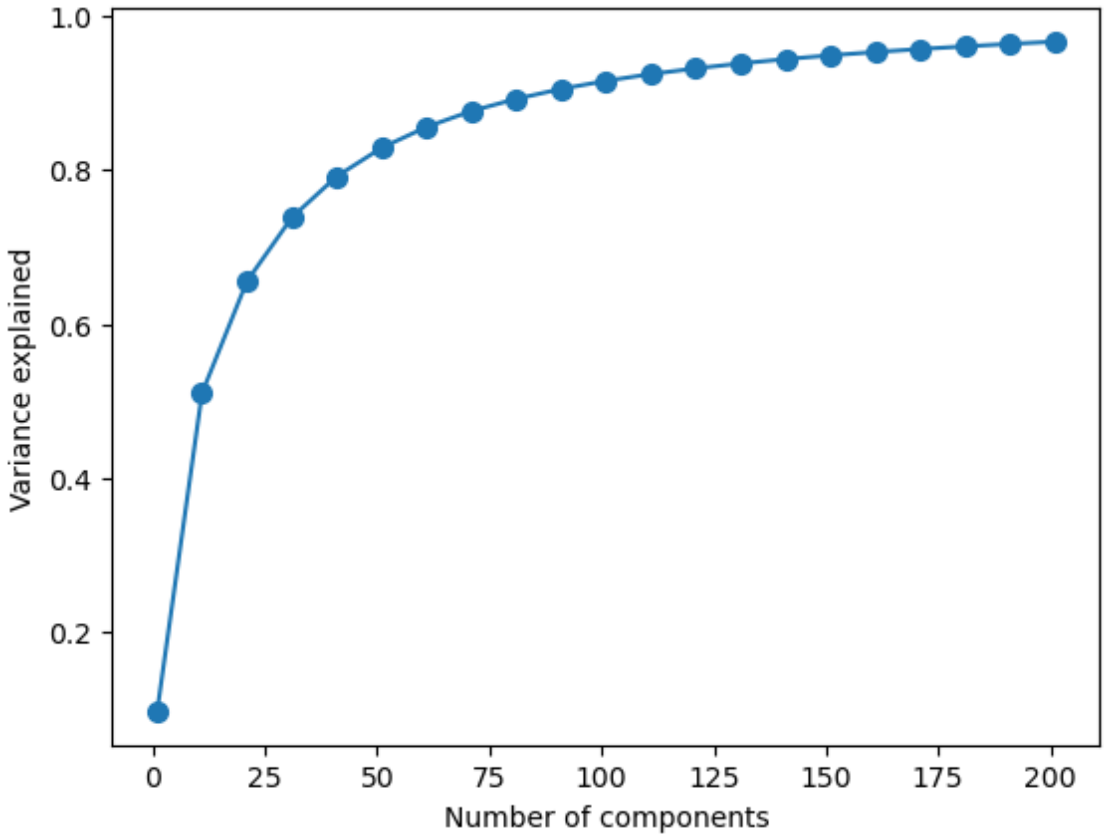
The process can be stated in terms of a simple eigenvalue problem of the *covariance matrix* of the feature vectors

The $n$ largest eigenvalues represents the $n$ principal components, the direction being the eigenvector and the explained variance the eigenvalue

# Principal Component Analysis (PCA)

The number of principal components is chosen by the user

Best approach: Choose the number of components that leads to a reasonable explained variance (90% or more)

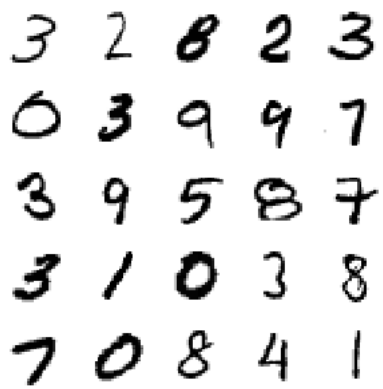


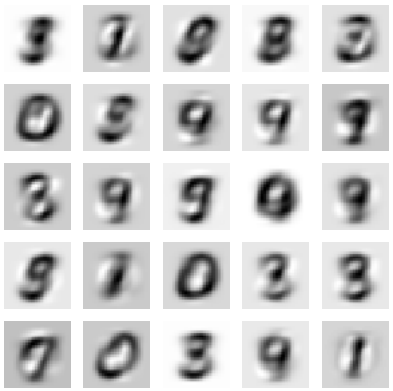
# Principal Component Analysis (PCA)

MNIST database of handwritten digits

Original

784 features



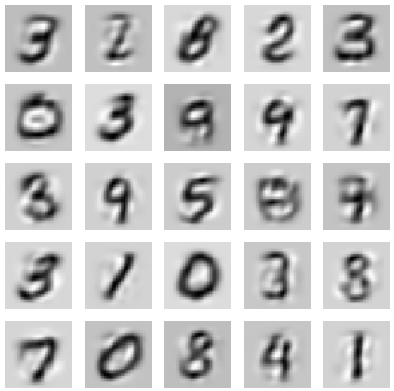


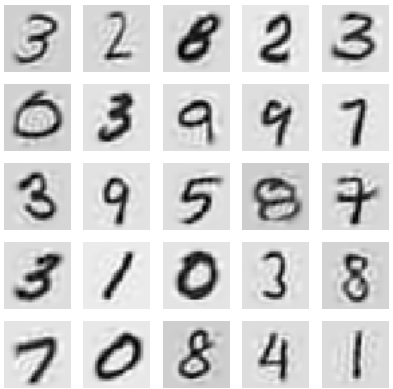
5 components

33% variance

25 components

69% variance





100 components

91% variance

# t-distributed Stochastic Neighbor Embedding (t-SNE)

t-SNE is an instance-based, unsupervised, non-linear dimensionality reduction technique specially suited for visually exploring high-dimensional data

It is not good as a preprocessing technique, but as a way to get an intuition on how data is arranged in the real high dimensional state

It can also be used as a *classification* technique, as it can identify *clusters* of datapoints

# t-SNE

t-SNE works by performing a mapping from the high dimensional space into a low dimensional space

It starts by calculating the distance (or cosine similarity) between all pairs of points in the high dimensional space and computing its probability distribution

Then it places the data points in the low dimensional space, trying to keep the distance probability distribution in the low dimensional space similar to the distance distribution on the high dimensional space

This is done by minimizing the *Kullback-Leibler divergence* (a measure of separation) of the low and high dimensional distance distributions

$$D_{\mathrm{DL}}(P||Q) = \sum_i P(i) \ln \frac{P(i)}{Q(i)}$$

# Data pre-processing

Data pre-processing is an initial step, prior to applying any ML algorithm, to prepare raw data to be suitable as an input of a learning model

Additionally, data in the real world can be messy

- With features of labels not taking a numerical value
- With too many features to be practical
- With values missing either in the features or in the labels
- With features expressing quantities that can range on widely different scales

We have already seen how to encode non-numerical data (ordinal, one-hot) and how to reduce the number of features (PCA)

Look at the other problems

# Dealing with missing data

In some datasets, datapoints can have missing values, due to different reasons

In this case, we can:

- Remove the datapoints with missing features. Acceptable if your dataset is big enough
- Use a *data imputation* technique

# Data imputation

Simple approaches:

- Replace the missing features by the average value of the rest of the present features. Eg: If the feature $i$ of the datapoint $j$, $x_j^{(i)}$, is missing, assign
$$x_j^{(i)} = \frac{1}{N-1} \sum_{k \neq j} x_k^{(i)}$$
- Replace the missing features by a value very different from the rest of the present features. Eg: If the feature $i$ of the datapoint $j$, $x_j^{(i)}$, is missing, and the present $x_k^{(i)}$, $k \neq j$, take values in, say $[0, 1]$, assign $x_j^{(i)} = 10$ or $100$
The hope is that the learning algorithm will learn what to do when one point has a feature significantly different from the others

# Data imputation

More advanced techniques:

- Use the missing features as a target for a regression problem
  For each datapoint, consider the $D-1$ feature vector $\hat{x}$, constructed with the features $x_j^{(q)}$, $q \neq i$, with a label $\hat{y}_j = x_j^{(i)}$
  Build a regression model to predict the value of $\hat{y}$ as a function of $\hat{x}$
  Use the model to predict the missing values of $\hat{y}$
- Use the missing features as a target for a $k$-NN classification
  Proceeding analogously as in the previous technique, use $\hat{x}$ and $\hat{y}$ of the complete datapoints as training set of a $k$-NN, and use it to assign the missing $\hat{y}$

# Data rescaling

Some learning algorithms can have problems if the features range on different scales. Eg: Some on the range $[0, 1]$, others in the range $[10^3, 10^5]$

Specially important in algorithms that weight the features of the inputs, like $k$-NN algorithms

It is useful thus to *rescale* the features into a common range

Different techniques can be used

# Data rescaling

- Normalization: Reduce all features to the same scale, say $[0, 1]$

$$\bar{x}_j^{(i)} = \frac{x_j^{(i)} - \min_j \left( x_j^{(i)} \right)}{\max_j \left( x_j^{(i)} \right) - \min_j \left( x_j^{(i)} \right)}$$

- Standardization: Rescale the features such that they have a zero mean and a unit standard deviation

$$\bar{x}_j^{(i)} = \frac{x_j^{(i)} - \mu^{(i)}}{\sigma^{(i)}}$$

Where

$$\mu^{(i)} = \frac{1}{N} \sum_j x_j^{(i)}, \quad \sigma^{(i)} = \sqrt{\frac{1}{N} \sum_j (x_j^{(i)} - \mu^{(i)})^2}$$

# Neural Networks

Neural networks can be understood as a mechanism to create *embeddings* for categorical data (particularly text, as in the case of **ChatGPT**)

- By *embedding* we understand a low dimensional, *learned*, continuous vector representation of discrete variables
- Their value reside in that, being learned, they reduce the dimensionality of categorical variables while *representing categories in the transformed space in a meaningful way*
  - Similar objects are placed close together in the transformed space

In contrast, one-hot encoding, or Bag of Words encoding, while representing an embedding in terms of vectors, they do not keep categories in the transformed space

- Close objects are randomly scattered

# Neural Networks

A neural network (NN) is, at its most basic, a mathematical function of the form

$$y = f_{\mathrm{NN}}(\vec{x}),$$

where $f_{\mathrm{NN}}(\vec{x})$ is a *nested* function of the form

$$f_{\mathrm{NN}}(\vec{x}) = f_n(\vec{f}_{n-1}\vec{f}_{n-2}(\cdots(\vec{f}_2(\vec{f}_1(\vec{x})))),$$

where $n$ is the number of *layers* of the NN, and the function $\vec{f}_i$,, $i = n-1, n-2, \ldots, 1$ are vector functions of the form

$$\vec{f}_i(\vec{x}) = \vec{g}_i(\mathrm{A}_i \cdot \vec{x} + \vec{b}_i)$$

The functions $\vec{g}_i$ are called *activation functions* and the matrices $\mathrm{A}_i$ and vectors $\vec{b}_i$ are parameters that are learned by minimizing some cost function.

The functions $\vec{g}_i$, $i = 1, \ldots, n-1$, can have different dimensions

The last function $f_n(\vec{x}) = g_n(\vec{x})$ is a scalar function

# Neural Networks

Typical choices for the activation function are:

- The logistic function

$$g(z) = \frac{1}{1 + e^{-z}}$$

- The hyperbolic tangent function

$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

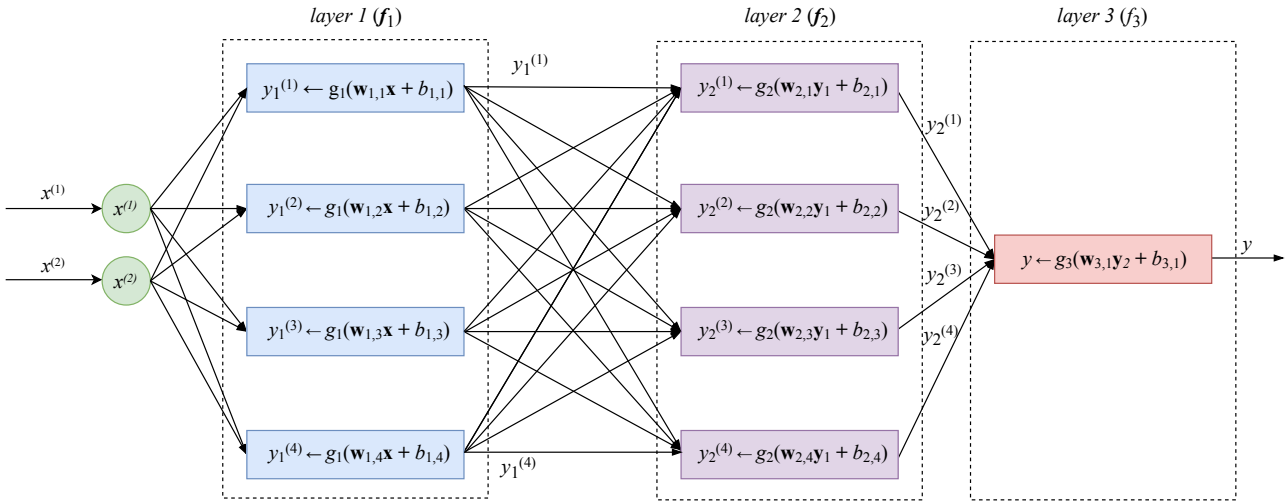
- The rectified linear unit function (`relu`)

$$g(z) = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$

- And many others

Different forms are good for different objectives.

# The Multilayer Perceptron (MLP)




Simplest neural network

This example has three layers.

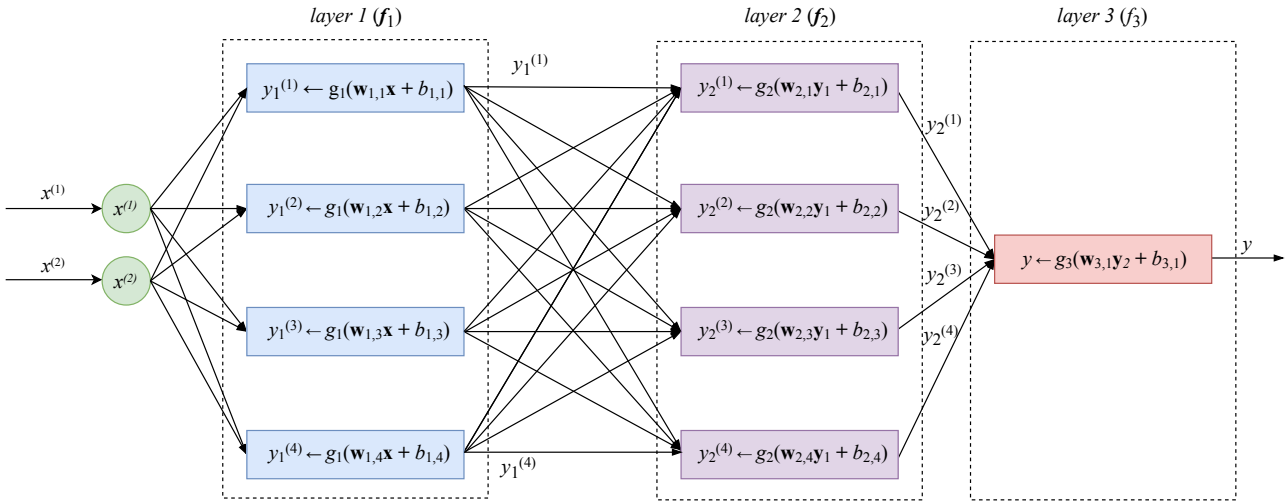
The initial input is a feature vector of two dimensions

Each layer is composed by four *logical units* or *nodes*, corresponding to activation vectors $\vec{g}_i$ of dimension four

The output of each logical unit in each layer is sent as input to the four logical units of the next layer (*fully connected* architecture)

The final output is here a scalar

# The Multilayer Perceptron (MLP)



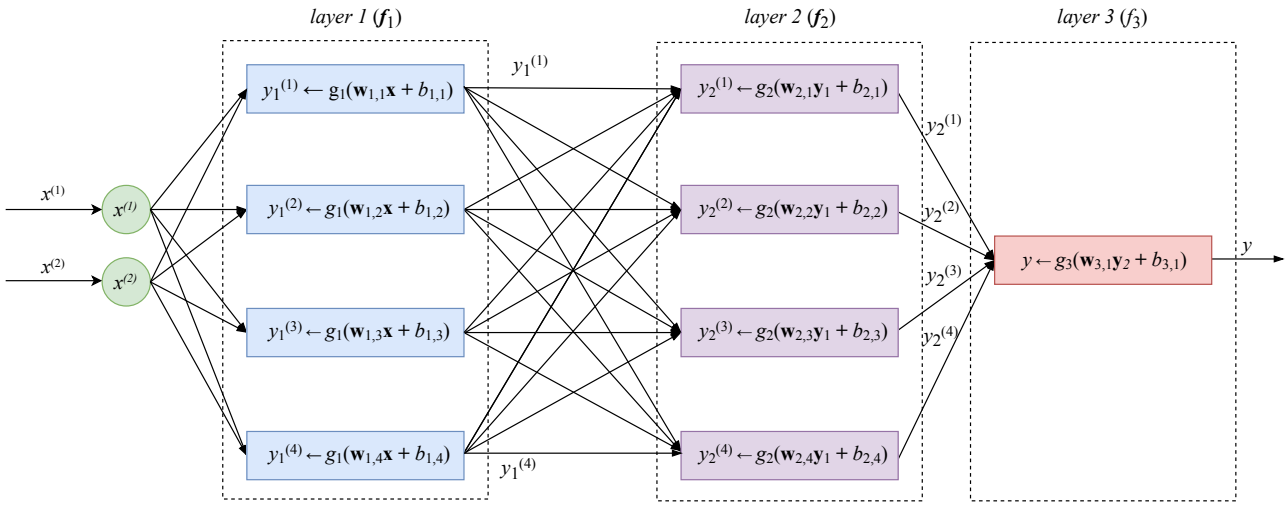

If the output is a binary number 0 or 1, the MLP works as a Support Vector Machine to classify datapoints

The largest complexity of the layers allow for classifying non-linearly separated structures

If the output is a number between 0 and 1, the MLP works as a logistic regression, again with better efficacy

In this cases, the cost function associated to the training is based in supervised learning

# The Multilayer Perceptron (MLP)




The layers that are not the output one are called *hidden layers* (two hidden layers in the example)

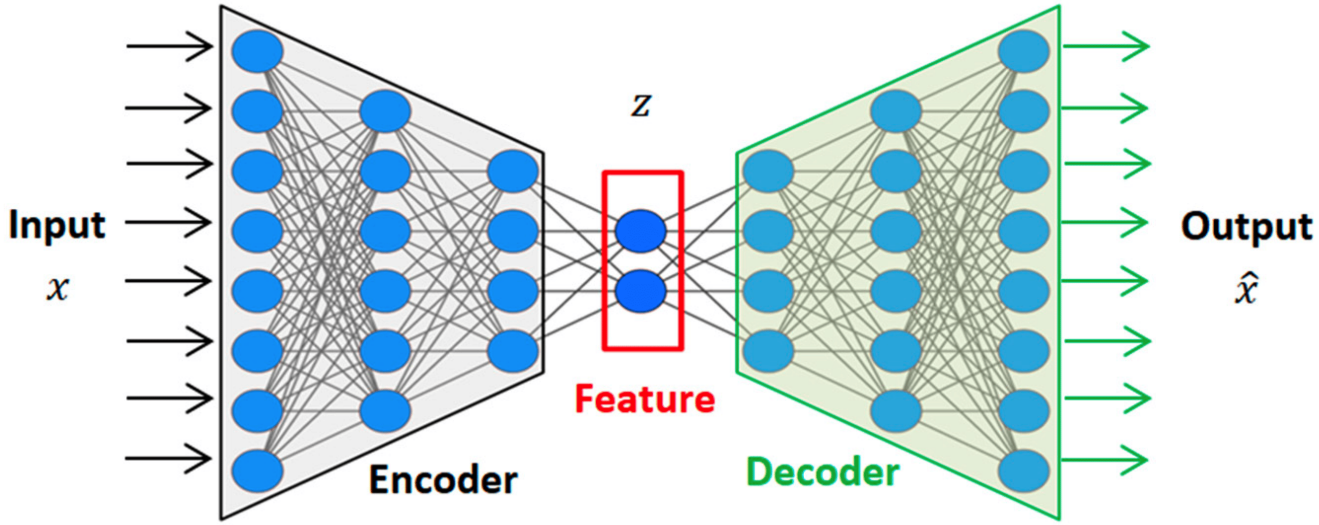
When there is a single hidden layer, the NN is referred a *shallow learning*

Where there is more than one hidden layer, the NN is referred as *deep learning*

The increased depth of a deep learning NN allows to capture more complex patterns in the input data

Deep learning NN were not feasible until recently due to technical problems in the learning process

# Autoencoders




NN that generates a *compressed knowledge representation* of the original input

The features in some hidden layer (*feature layer*) represent the embedding of the original data

The learning is unsupervised, applying *back-propagation*, with the cost function being the *reconstruction error* $\mathrm{L}(x, \hat{x})$ between the original input and the reconstruction of it from the decoder

Autoencoders are at the basis of ChatGPT, combined with language, sentiment and dialogue models, based in the embedding

# High performance computing and Machine Learning

Machine learning algorithms can be very costly computationally

- Data with thousands of features
- Models, such as autoencoders, with hundreds of thousands or millions of parameters to adjust minimizing a cost function

To accomplish this task, high performance computing is needed, in particular parallel computing

At this respect, GPUs are particularly useful

GPU cores have less performance than CPU cores but they are much more in number (e.g. NVIDIA RTX 4090: 16,384 CUDA cores, 516 tensor cores), so GPUs can manage many more mathematical calculation in parallel with greater efficiency

# Practical Machine Learning

For practical application of Machine Learning, the Python programming language is particularly well suited, based in some external packages

- `numpy` and `scipy` for mathematical manipulation
- `theano` for numerical computations
- `pandas` for high-level dataset manipulation
- `seaborn` for data visualization
- `scikit-learn` for basic machine learning tasks
- `tensorflow` or `pytorch` to construct and train neural networks

Many of this packages include high performace capabilities, with parallel computing in the CPUs and the GPUs of the computer

# Applications of Machine Learning in Physics and Chemistry

- Image analysis to detect gravitational lensing (black holes)
- Finding patterns in high energy collisions
- Detection of gravitational waves
- Learning computing demanding numerical models in order to predict outcomes without rerunning the code
- Detecting phase transitions from image analysis of system's configurations
- Representation and discovery of atomic potentials
- Discovering relation between minima on complex energy landscapes
- Discovery of new potentials in density functional theory