

Neural Methods for NLP

Master LiTL --- 2024-2025

chloe.braud@irit.fr

https://gitlab.irit.fr/melodi/andiamo/teaching_cbraud/master_litl



Goals

- Understand what are Neural Networks: the motivations and steps from linear to neural models
- Being able to train and evaluate a deep learning model
 - understand the hyper-parameters, being able to optimize a DL model
 - understand the varied architectures, their underlying motivations, their use
 - understand the input: what are (word) embeddings?
 - how to build a model for a specific application: classification, sequence labelling, generation
- Having an idea of the limitations and current challenges

Practical sessions / Assignments:

- Libraries: PyTorch, HuggingFace
- Environment: Google Colaboratory

Schedule 2024-2025

	1	26.11	13h-16h	3	(C1) ML Reminder + Intro DL	TP1-POO
	2	03.12	13h-16h	3	(C2) Intro DL + Embeddings	TP2-FFNN
	3	10.12	13h-16h	3	(C3) Embeddings + start projects	TP3-Embed
	-	17.12	-	-	BREAK	
	(holidays)					
→Part 1 due	4	07.01	13h-16h	3	(C4) Training a NN	TP5-HFData
						TP6-TrainFFNN
	5	14.01	13h-16h	3	(C5) CNN, RNN	TP7-LSTM
						TP8-HFTrain
	6	21.01	13h-16h	3	Projects	
	7	28.01	13h-16h	3	(C6) Encoder-decoder, transformer	TP9-Biais
	-	04.02	-	-	BREAK	
→Part 2 due	8	11.02	13h-16h	3	(C7) Current challenges	→ project defences

Projects

System:

- Read a research paper on the chosen task
- (Implement a non neural baseline system)
- Compare with a neural architecture
- Augment the system within a multilingual / cross-domain setting

Topics:

- Text classification: sentiment analysis, fake news detection, ...
- Sequence labelling: named entity recognition, POS tagging, ...

Assignments (groups 2-3):


- **07/01** : Pre-processing code + report part 1
 - Code : data pre-processing
 - Report Part 1 : description of the data and related work
- **09/02** : Code + report part 2 (+ corrected Part 1)
 - Code for training and evaluating the system
 - Report Part 2 : describe the system and present the results
- **11/02** : Oral presentation (10-15 mn)

Bibliography and resources

There are many many good resources for more details.

- *Neural Network Methods for NLP*, Y. Goldberg
- Online courses:
 - [Neural Nets for NLP, G. Neubig](#),
 - [Stanford courses with C. Manning](#) (official website: <https://web.stanford.edu/class/cs224n/>)
 - En français : <https://gricad-gitlab.univ-grenoble-alpes.fr/talks/fidle/-/wikis/home>
- J. Eisenstein course:
<https://github.com/jacobeisenstein/gt-nlp-class/blob/master/notes/eisenstein-nlp-notes.pdf>

I used other resources to build this course, I'll try to give all the sources used.



Neural Methods for NLP

Master LiTL --- 2024-2025

chloe.braud@irit.fr

https://gitlab.irit.fr/melodi/andiamo/teaching_cbraud/master_litl

Course 1: Machine Learning (reminder)



Neural methods for NLP

- 1980's: Symbolic NLP
 - rule-based approach, hand-written rules
 - advantages: based on linguistics expertise, very precise
 - inconvenients: lack of coverage, time consuming
- 1990's: 'Statistical' NLP
 - learn rules automatically = (mostly linear) functions, with high-dimensional, sparse feature vectors
 - large annotated corpora
 - handcrafted features
 - rather fast to train, still good baselines
- ≈ 2010: 'Neural' NLP
 - combine linear and non-linear functions, over dense inputs
 - (very) large annotated corpora and very large unannotated corpora
 - improved performance (in general), no feature engineering
 - harder to interpret ("black box")



Neural methods for NLP

- 1980's: Symbolic NLP
 - rule-based approach, hand-written rules
 - advantages: based on linguistics expertise, very precise
 - inconvenients: lack of coverage, time consuming
- **1990's: 'Statistical' NLP**
 - learn rules automatically = (mostly linear) functions, with high-dimensional, sparse feature vectors
 - large annotated corpora
 - handcrafted features
 - rather fast to train, still good baselines
- ≈ 2010: 'Neural' NLP
 - combine linear and non-linear functions, over dense inputs
 - (very) large annotated corpora and very large unannotated corpora
 - improved performance (in general), no feature engineering
 - harder to interpret ("black box")



Machine Learning for NLP

Applications:

- NLP applications: spam filtering, spell checking, machine translation, summarization, web search, recommendation systems, sentiment analysis, hate speech detection...
- NLP tasks: sentence splitting, tokenization, POS tagging, NER, syntactic parsing, semantic parsing, discourse parsing, event identification, detecting language change, representation learning, speech recognition...

Data investigation:

- Looking at how works your model could help understanding your data/problem:
 - e.g. **Age or Gender bias in models**: *Gender Bias in Part-of-Speech Tagging and Dependency Parsing Data*, A. Garimella, C. Banea, D. Hovy, & R. Mihalcea. ACL 2019
- (Linguistic) Hypothesis checking
 - e.g. Scientific fraud: **specific writing style?** *Is writing style predictive of scientific fraud?*, C. Braud and A. Søgaard. EMNLP 2017
- For 'fun': e.g. see T. Van de Cruys' book of **poetry generated** via ML

Content

Statistical Learning

1. Learning problems
2. Workflow and terminology
3. Linear classification
4. Representation function
5. Basics of POO

Practical session 0: basics of POO, implement a ML model with Scikit

Content

Statistical Learning

ML \approx DL

1. Learning problems
2. Workflow and terminology
3. Linear classification
4. Representation function
5. Basics of POO

Practical session 0: basics of POO, implement a ML model with Scikit

Content

Statistical Learning

ML \approx DL

ML \neq DL

1. Learning problems
2. Workflow and terminology
3. **Linear classification**
4. **Representation function**
5. Basics of POO

Practical session 0: basics of POO, implement a ML model with Scikit

Learning problems and scenarios

Most common learning problem in NLP: **classification**

Most common scenario: **supervised** learning

→ using pre-trained word embeddings is in fact doing semi-supervised learning

The different tasks

- **Classification:** predict a categorical label for each item
 - single label: each instance is assigned a single label
 - binary: 2 labels, e.g. an email is either a spam or not
 - multi-class: > 2 labels, e.g. sentiment is either positive, negative or neutral
 - multi-label: each instance is assigned multiple labels, e.g. *The Lord of the Ring* is classified as: Adventure, Fantasy, Drama
- **Sequence labeling / structured prediction:** predict a categorical label for each member of a sequence
 - e.g. POS tagging, NER...
 - can be seen as performing independent classification tasks on each item
 - but performance are improved when taking into account the dependence between the elements
- **Regression:** Predict a **real value** for each item
 - e.g.: prediction of stock values, variations of economic variables, house prices..
 - rarer for NLP, but e.g. data with depression “scores” (DAIC)

The different tasks

- **Clustering:** Partition items into homogeneous regions
 - kind of classification but without classes known a priori
 - can be useful if you don't have manual labels or want to explore your data
 - often used for very large data sets
 - e.g.: in social network analysis, attempt to identify “communities” within large groups of people.
- **Ranking:** Order items according to some criterion (e.g. Web search)
- Dimensionality reduction or manifold learning: Transform an initial representation of items into a lower-dimensional representation (for pre- processing or visualisation)

The learning scenarios

Depend on the **annotations** you have:

Supervised learning:

- we have a set of **labeled** examples as training data
- most common for classification and regression

The learning scenarios

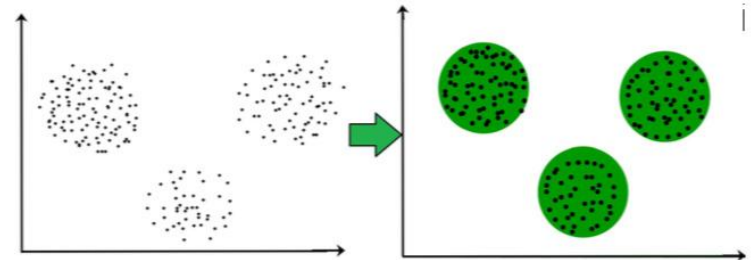
Depend on the **annotations** you have:

Supervised learning:

- we have a set of **labeled** examples as training data
- most common for classification and regression

Unsupervised learning:

- we only have **unlabeled** training data
 - e.g.: Clustering and dimensionality reduction
 - often hard to evaluate



The learning scenarios

Semi-supervised learning: the training sample consists of both **labeled** and **unlabeled** data

The learning scenarios

Semi-supervised learning: the training sample consists of both **labeled** and **unlabeled** data



easier to obtain!

The learning scenarios

Semi-supervised learning: the training sample consists of both **labeled and unlabeled** data

Very hard in practice, but many variations:

- **labeled + automatically labeled** data
 - e.g. sentiment analysis with smileys as (noisy) labels
- **labeled + external resource** giving constraints
 - e.g. POS tagging with a dictionary
- **labeled + labeled** data for another task
 - multi-task learning
- **labeled + unlabeled**: pre-trained word embeddings



easier to obtain!

Especially used for **transfer learning** / domain adaptation:

- e.g. building a model for a new language or for a new genre of texts

Supervised classification

Supervised classification:

- the most common scenario for NLP (with supervised structured prediction)
- **supervised**: input = **labeled** data points
- **classification**: assign a category/class to each item, e.g.
 - is a word a VERB or a NOUN?
 - Is a document talking about Sport or Politics or Economy?

Binary vs Multi-class:

2 classes (e.g. positive/negative, comedy/drama) vs more than 2 classes (e.g. positive/negative/neutral, any genre)

Distinction that has an impact

- on the algorithm: various strategies to deal with MC problems
- on evaluation: various metrics
- but rather transparent with scikit: algorithms/functions can be used for both binary and MC problems

Machine Learning: workflow and terminology

The different steps when doing machine learning:

- (1) preparing data,
- (2) learning and tuning,
- (3) predicting and evaluating

Terminology:

- input
- model and parameters
- train/dev/test sets

Machine Learning

Start with:

- a set of labelled data = data points + (gold) labels
- a function that could be used to compute a label for a data point

Learning a model:

- Goal: try to get the best function, i.e. that finds the right/gold label
- Process: iterate over the examples, and adjust the parameters of the function to avoid errors

Evaluating the model:

- Goal: evaluate the performance of the learned model
- Process: once the model is learned / trained, make predictions over unseen data and compute some performance metrics

Machine Learning

Start with:

- a set of labelled data = data points + (gold) labels → **supervised setting**
- a function that could be used to compute a label for a data point → **classification**

Learning a model:

- Goal: try to get the best function, i.e. that finds the right/gold label
- Process: iterate over the examples, and adjust the parameters of the function to avoid errors

Evaluating the model:

- Goal: evaluate the performance of the learned model
- Process: once the model is learned / trained, make predictions over unseen data and compute some performance metrics

Machine Learning

Start with:

- a set of labelled data = **data points + (gold) labels** → **supervised setting**
- a **function** that could be used to compute a label for a data point → **classification**

Learning a model:

- Goal: try to get the best function, i.e. that finds the right/gold label most often
- Process: iterate over the examples, and adjust the **parameters** of the function to avoid errors

Evaluating the model:

- Goal: evaluate the **performance** of the learned model
- Process: once the model is learned / trained, make **predictions** over unseen data and compute some performance metrics

Supervised classification

Data preparation

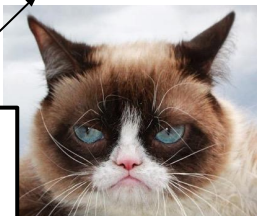
Training

Evaluating

Input Data / Training set
(Labeled examples)

Test set

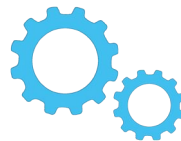
Data points x **gold Labels y**



Cat



Sandwich



learning some
function f with
parameters



output

Cat

20/20

Très bien !



Use f to
predict a
label

Compute
score using
some
performance
metrics

Workflow: (1) Data preparation

- **Define the problem:** task? labels?
- Collect (labeled) data: datasets available online, scikit toy datasets, scrap data...
- Randomly partition your data, i.e. shuffle then split:
 - Train / dev / test: e.g. 80-10-10 (or use pre-defined split), in general train > test
 - Train / test + cross-fold on training set for tuning
- Data description: **you need to know your data!**
 - Number of training/evaluation examples
 - Class distribution: number of examples per class
 - Vocabulary, language, genre, etc...
- **Feature extraction/engineering:** critical step, reflects prior knowledge
 - Possibly linguistic pre-processing: POS tagging, parsing, NER, etc...
 - Vectorization, normalization

Supervised classification

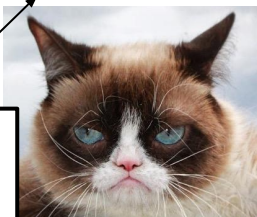
Data preparation

Training

Evaluating

Input Data / Training set
(Labeled examples)

Data points x **gold Labels y**

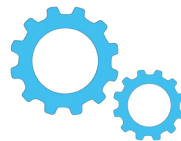


Cat



Sandwich

Represented
by some
features



**learning some
function f with
parameters**

Test set



output

Cat

20/20

Très bien !



Use f to
predict a
label

Compute
score using
some
performance
metrics

Workflow: (2) Learning + Tuning

- **Choose a learning algorithm:** crucial, especially if training is long
 - Advice: try first with a fast algorithm
- Train: at each training step,
 - update values for the parameters
 - the values for the hyper-parameters are fixed
- Tune:
 - identify the tunable hyper-parameters
 - search the best values for the hyper-parameters

Learning algorithms for classification

- Naive Bayes
- **Linear classifiers:**
 - perceptron
 - passive-aggressive
 - Logistic Regression aka MaxEnt
 - linear SVM
- Non linear SVM
- Neural networks

See [the doc on supervised learning](#)

See [the tutorial: working with text](#)

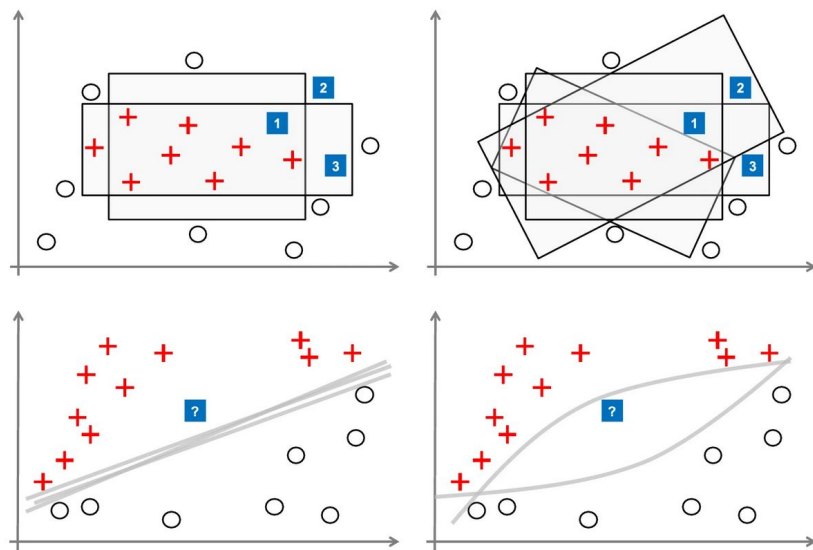
Workflow: (2) Learning + Tuning

- Choose a learning algorithm: crucial, especially if training is long
 - Advice: try first with a fast algorithm
- **Train:** at each training step,
 - update values for the **parameters**
 - the values for the hyper-parameters are fixed
- Tune:
 - identify the tunable hyper-parameters
 - search the best values for the hyper-parameters

Model and learning function

Remember that we are learning a function:

- **Target function**: the true function f we want to learn / approximate.
- **Hypothesis** (sometimes called *model*): a function h that we hope is similar to the target.



Supervised learning :

- Based on a training sample $S = \{(x_i, y_i)\}_{1,m}$
- we look for a on cherche an underlying law of dependence / regularities

e.g. a function h as close as possible to f (the target function) with $y_i = f(x_i)$

many possible hypothesis

Learning a model

Remember that we want to avoid errors, find the best hypothesis:

- **Loss function**: measures the difference, or loss, between a predicted label and a true label.
 - $L: Y \times \hat{Y} \rightarrow \mathbb{R}$ ~ 'count the number of times' $y \neq \hat{y}$ (also called **cost**)
 - zero-one loss, squared loss, hinge loss...

The learning algorithm tries to get the smaller possible loss on the examples it is given:

- if the predicted label is wrong
 - modifies the values of the weights \mathbf{w} so that next time we compute the label, we get the right one i.e. $\hat{y} = \mathbf{w} \cdot \mathbf{x}$
- if the predicted label is right, don't modify anything (except if we want some margin)

The parameters \mathbf{W} and \mathbf{b} are set to minimize L (usually, the sum of the losses over the training examples)

$$L(\Theta) = 1/n \sum_{i=1..n} L(f(\mathbf{x}_i ; \Theta), y_i) \rightarrow \text{The loss should decrease with learning (less and less errors)}$$

Thus training correspond to finding this minimum (= **optimization problem**):

$$\hat{\Theta} = \operatorname{argmin}_{\Theta} L(\Theta) = \operatorname{argmin}_{\Theta} 1/n \sum_{i=1..n} L(f(\mathbf{x}_i ; \Theta), y_i)$$

Workflow: (2) Learning + Tuning

- Choose a learning algorithm: crucial, especially if training is long
 - Advice: try first with a fast algorithm (for DL= smaller layers, smaller input size...)
- Train: at each training step,
 - update values for the parameters
 - the values for the hyper-parameters are fixed
- **Tune:**
 - identify the tunable **hyper-parameters**
 - search the best values for the hyper-parameters

Supervised classification

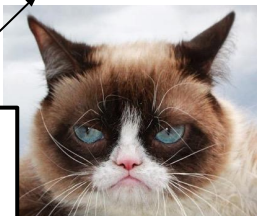
Data preparation

Training

Evaluating

Input Data / Training set
(Labeled examples)

Data points x **gold Labels y**

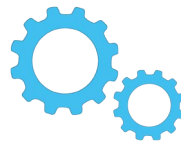


Cat



Sandwich

Represented
by some
features



learning some
function f with
parameters

Test set



output

Cat

20/20

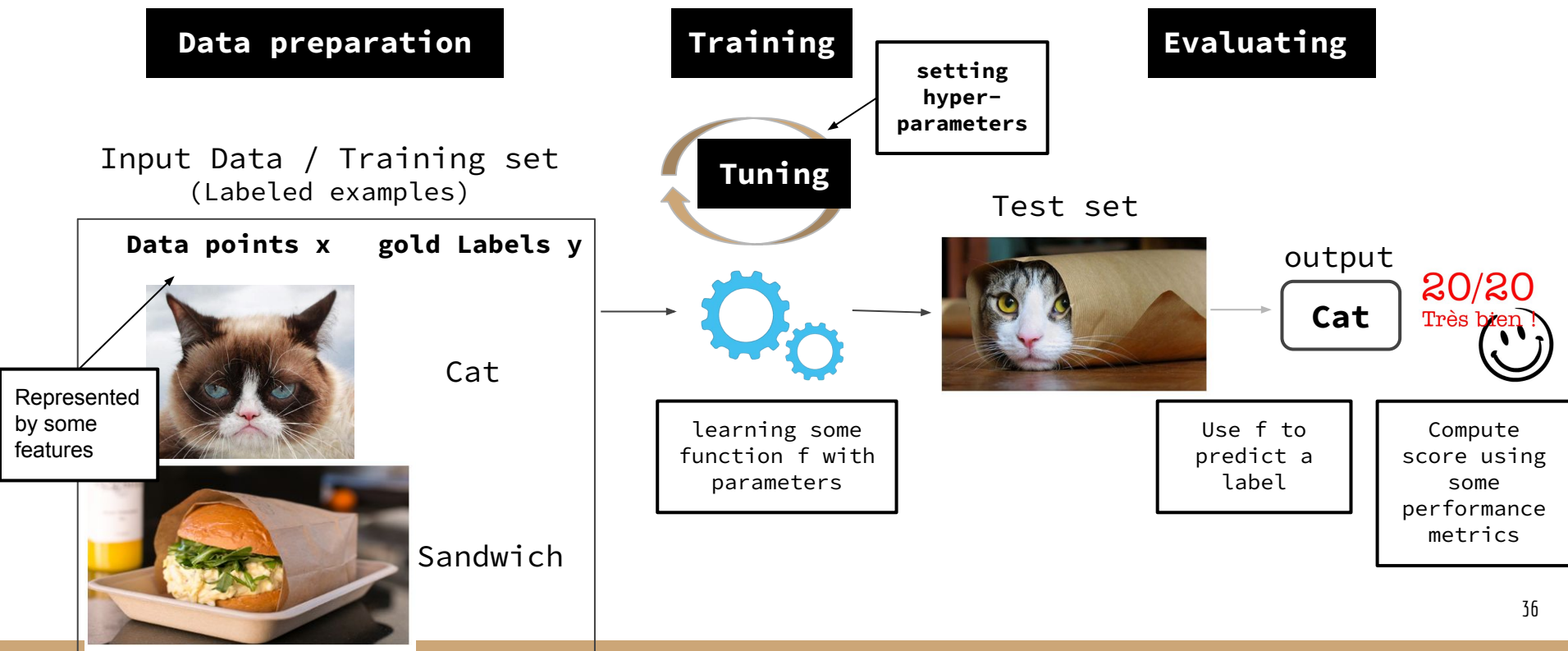
Très bien !



Use f to
predict a
label

Compute
score using
some
performance
metrics

Supervised classification



Tuning / Optimization

- We learn the parameters of the model (or weights, \mathbf{w} or $\boldsymbol{\theta}$)
- We set values for the hyper-parameters associated to the learner

→ Tuning = searching for the best values for hyper-parameters e.g. smoothing, regularization strength etc

Tuning process:

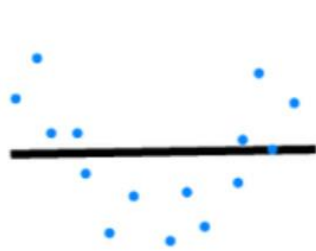
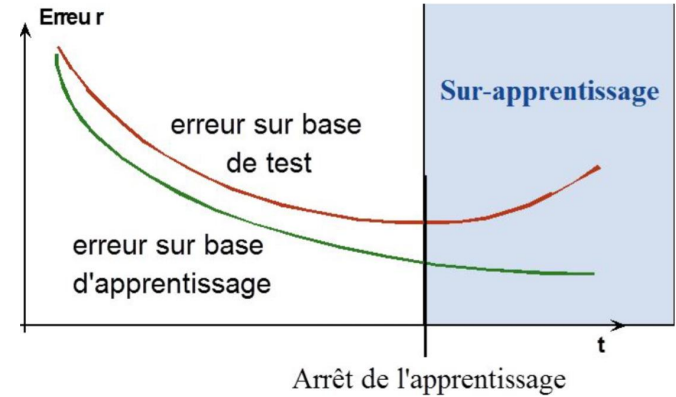
1. identify the hyper-parameters of your model (SciKit: `estimator.get_params()`)
2. choose the *right* performance metrics to optimize (accuracy, F1, rouge ...)
3. choose the right procedure → **always set apart a test set**:
 - a. use a **validation / development set**:
 - i. define a set of possible values for each hyper-parameter
 - ii. train a model for each subset of values and evaluate on the dev set
 - iii. compare the results: keep the model giving the best score on dev
 - iv. evaluate (only) this model on the test set
 - b. **n-fold cross-validation**: esp. when small amount of data
 - i. very easy with scikit: **grid-search cross-validation**

Learning is not memorizing

Consistent model:

- **no error on train set**
- but **poor performance on test**,

i.e. memorize the data, unable to generalize



Underfitting



Desired

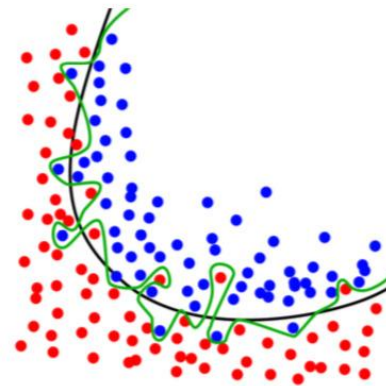


Overfitting

Overfitting

- Very complex decision surface: no generalization to unseen data
- Less complex: might generalize better in spite of some errors

"The **green line represents an overfitted model** and the **black line represents a regularized model**. While the green line best follows the data, it is too dependent on the training data" (Mohri)



→ Solution = **regularization**: constraining a model to make it simpler

= add a regularization term to minimize the complexity of the model, i.e. an **hyper-parameter** corr. to the strength of the regularization

Learning a model

Remember that we want to avoid errors, find the best hypothesis:

- **Loss function**: measures the difference, or loss, between a predicted label and a true label.
 - $L: Y \times \hat{Y} \rightarrow \mathbb{R}$ ~ 'count the number of times' $y \neq \hat{y}$ (also called **cost**)
 - zero-one loss, squared loss, hinge loss...

The learning algorithm tries to get the smaller possible loss on the examples it is given:

- if the predicted label is wrong
 - modifies the values of the weights \mathbf{w} so that next time we compute the label, we get the right one i.e. $\hat{y} = \mathbf{w} \cdot \mathbf{x}$
- if the predicted label is right, don't modify anything (except if we want some margin)

The parameters \mathbf{W} and \mathbf{b} are set to minimize L (usually, the sum of the losses over the training examples)

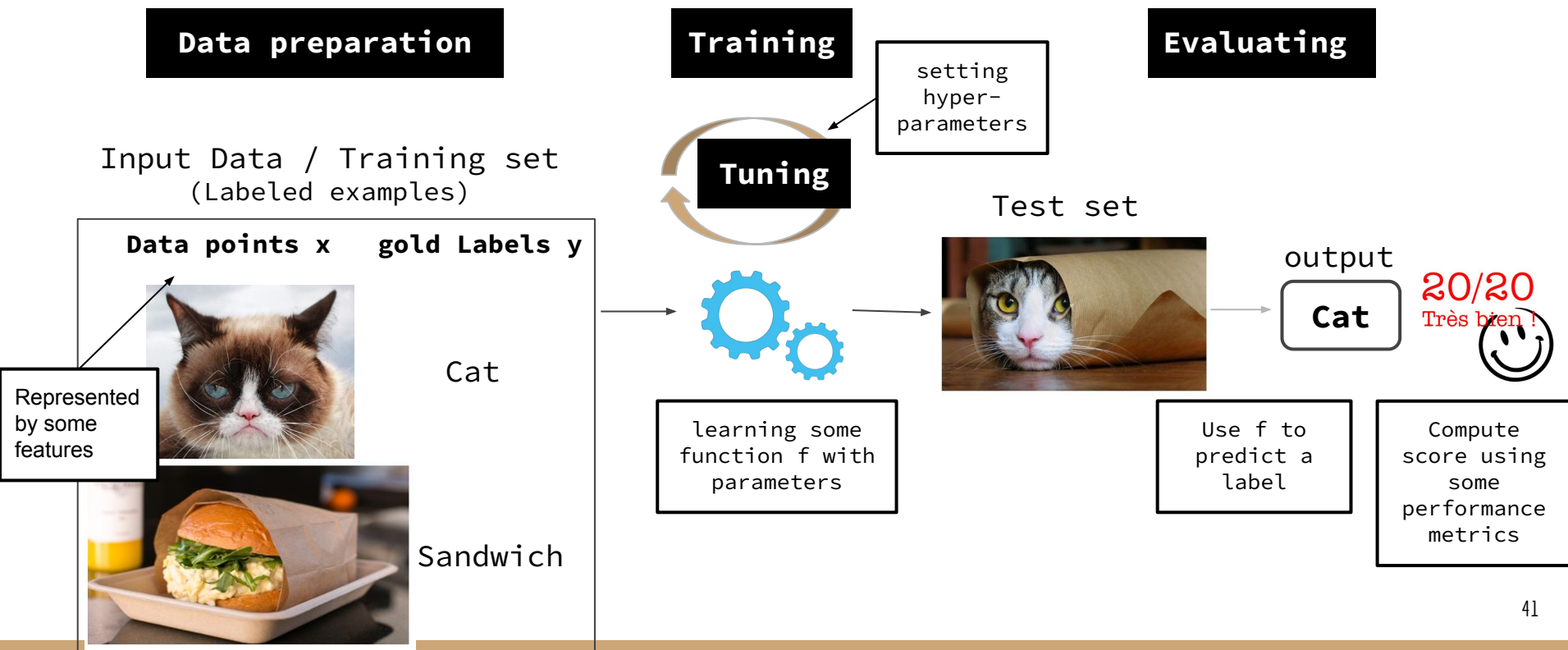
$$L(\Theta) = 1/n \sum_{i=1..n} L(f(\mathbf{x}_i; \Theta), y_i) \rightarrow \text{The loss should decrease with learning (less and less errors)}$$

Thus training correspond to finding this minimum (= **optimization problem**):

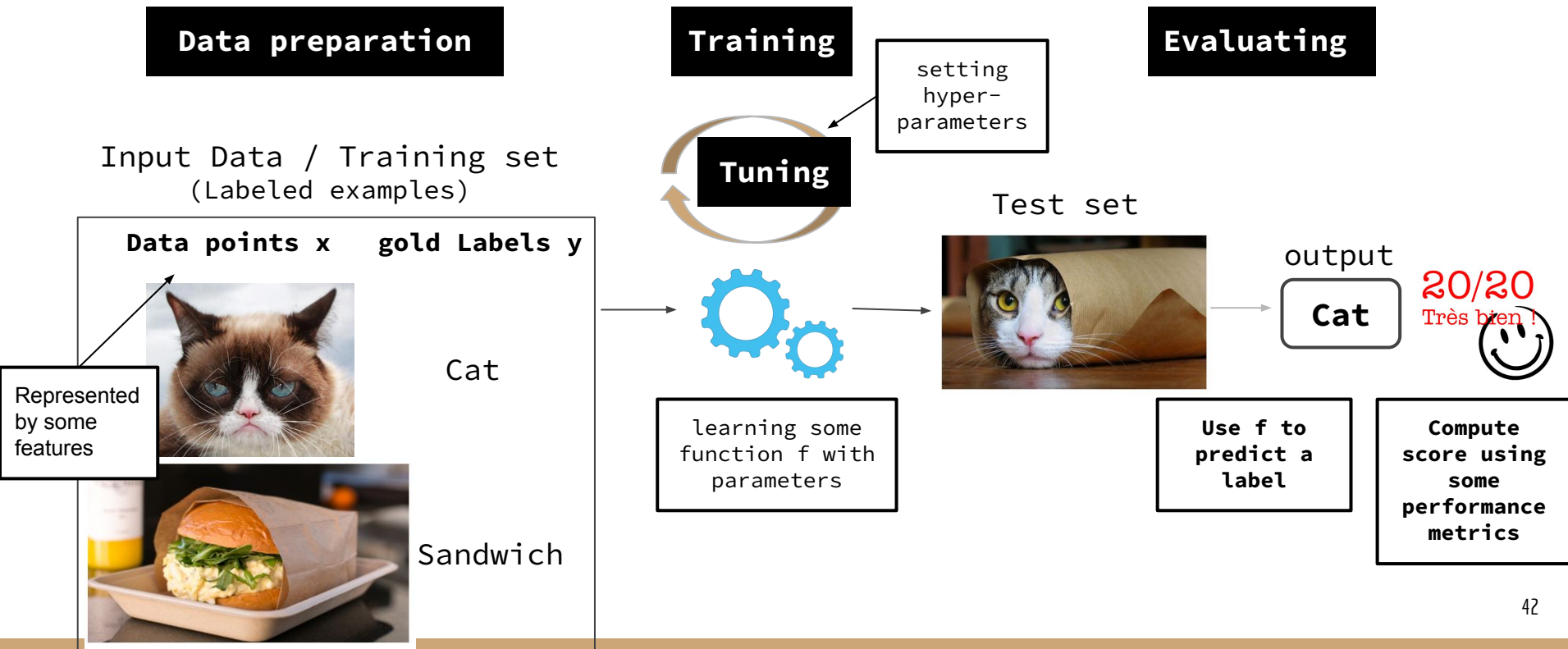
$$\hat{\Theta} = \operatorname{argmin}_{\Theta} L(\Theta) = \operatorname{argmin}_{\Theta} 1/n \sum_{i=1..n} L(f(\mathbf{x}_i; \Theta), y_i) + \lambda R(\Theta)$$

Regularization

Supervised classification



Supervised classification



(3) Prediction + Evaluation

Using the **final values** for parameters and hyper-parameters, **evaluate on test**

- Use your model to **make predictions** on the **unseen** test data (y_{pred})
- **Compute a score** by comparing y_{true} and y_{pred} for each example in the test
- **Compare** to other systems: baselines (on dev first), state-of-the-art...

It is important to:

- Keep track of the values used (final and tested) for the hyper-parameters for reproducibility!
- Choose a / several relevant **evaluation metrics**
- Propose relevant baselines

Classification metrics

For classification, we mostly use:

- Global scores: accuracy, averaged F1
- Per class scores: precision, recall, F1
- + confusion matrix: better understand the system behaviour

Accuracy is the most common metrics:

- fraction of correctly predicted samples
 - e.g. 90 well predicted over 100 examples: accuracy = 90%
- issue esp. with imbalanced data,
 - e.g. Cancer detection: 90 non cancer, 10 cancer is 90% a good score for predicting cancer?
 - we want to predict well the positive class

Evaluating a model

Report one or **several metrics**:

- depend on the setting (binary or multi-class) and task
- classification: Accuracy, Macro/weighted F1, prec/rec/F1 per class

Compare to other systems:

- baselines: simplest feature/algo, dummy classifier (most frequent class)
- state-of-the-art: systems from the literature, reported or reproduced, compare different algorithms
- compare different feature sets
- compare different datasets: prove the robustness of your method over different genres, languages...

Try to **understand your model**:

- Scikit: classifiers have a *coef_* parameters that allows to inspect the weights associated to each feature
 - **eli5**: a library to debug ML models, compatible with Scikit, see the doc
- try to relate observed behaviour to a priori knowledge, esp. linguistic

Input: examples, features and labels

Examples or samples / instances / data points = items of data used for learning or evaluating (m examples)

Features = set of attributes associated to an example (n features)

A **set of examples**: a **matrix X** of size $m \times n$

- 1 example = 1 row, i.e. a vector \mathbf{x} of size n

Labels = values assigned to examples; for classification:

- General label set: Y of p classes
- Labels for all examples = a list of size m , e.g. y_{true}
- 1 label for 1 example = 1 value y in the list
- **labeled example** = a pair (\mathbf{x}, y)

Dataset for ML = X and \mathbf{y}_{true}

Model, Parameters and hyper-parameters

Model: what we learn is a model of our data

- We sometimes call model the weights learned, i.e. the importance that the model associates with each feature
 - e.g. Sentiment analysis: 'love':+10, 'hate':-42, 'green':0...
- Weights are saved in a vector \mathbf{w} (or θ) of size $n(+1)$
- Each of these weights is a **parameter**/coefficient of the model

Hyper-parameters (or free parameters, part of the model):

- there could be parameters dependent on the learning algorithm used
- setting the values for these hyper-parameters is called **tuning** the model

Train / Dev / Test sets

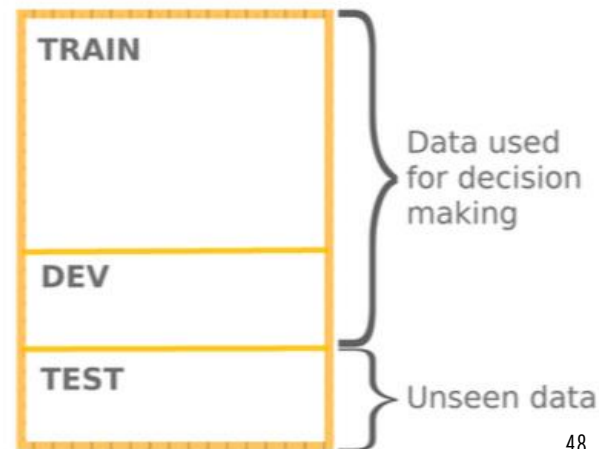
Training sample/set: examples used to train a learning algorithm, to **learn/fit** a model

Development/validation set: examples used to **tune** the hyper-parameters of a learning algorithm

Test/evaluation set: examples used to **evaluate** the performance of a learning algorithm

- The test set is separate from the train/dev sets
- The test set is not made available in the learning stage

Searching for the best model on the test set =



Linear classification

- Binary classification: linear functions, weight matrix and bias
- Reminder: Logistic Regression = linear scores + logistic function
- Loss function

ML → DL:

- Change here = power of non linearity
- → LR performed by each neuron

Why Linear classifiers?

Remember that: ML is about finding a function h that best approximate the target function f

- Searching over the set of all possible functions is very hard
- We thus restrict ourselves over specific families of functions, the ***hypothesis class*** e.g. the space of all linear functions with d_{in} inputs and d_{out} outputs
 - inject the learner with inductive bias: a set of assumptions about the desired solution
 - facilitate procedures for searching solutions
- The hypothesis class also determines what can and cannot be represented by the learner!

Linear Classifiers

Hypothesis class = high-dimensional linear functions, of the form:

$$f(\mathbf{x}) = \mathbf{W} \cdot \mathbf{x} + \mathbf{b} \quad \text{with } \mathbf{x} \in \mathbb{R}^{d_{\text{in}}}, \mathbf{W} \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}, \mathbf{b} \in \mathbb{R}^{d_{\text{out}}}$$

- Searching over the space of functions = searching over the space of parameters, i.e. finding the best $\Theta = \mathbf{W}, \mathbf{b}$.
- Sometimes, to make the parameterization explicit, we write: $f(\mathbf{x}; \mathbf{W}, \mathbf{b})$
- In binary classification, \mathbf{w} is a vector

Recall on linear algebra: $\mathbf{W} \cdot \mathbf{x} = \sum_j w_j x_j = w_0 \cdot x_0 + w_1 \cdot x_1 + \dots + w_n \cdot x_n (+b)$

With n features, we have:

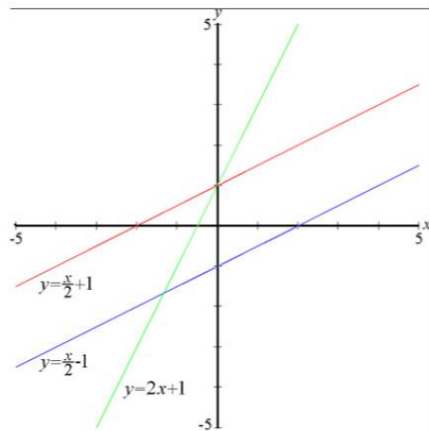
- a data point: $\mathbf{x} = \langle x_0, x_1, \dots, x_n \rangle$
- the weights: $\mathbf{w} = \langle w_0, w_1, \dots, w_n \rangle$

Linear Classifiers

The decision boundary is a **linear function** of the input: in the binary case, it's a line (2 dimensions / features), a plane (3 d) or an hyperplane (n d) separating the two classes

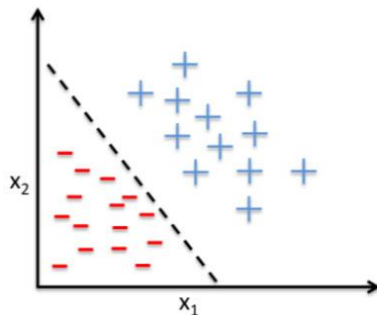
Decision boundary:

$y = w_0 \cdot x_0 + b$: a line

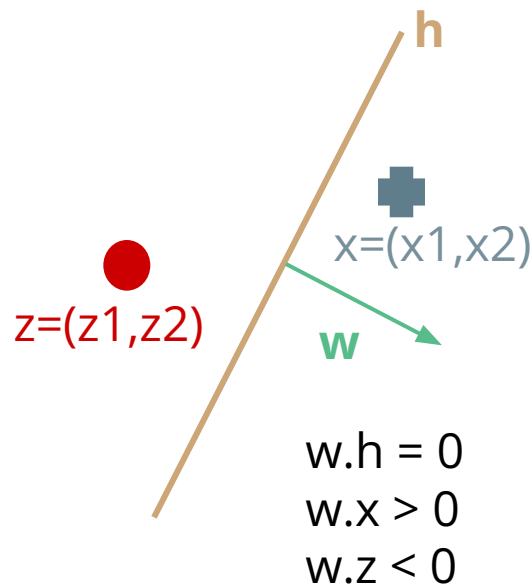


Prediction:

if $\hat{y} = \mathbf{w} \cdot \mathbf{x} > 0$, predict positive class.



Example of a linear decision boundary for binary classification.

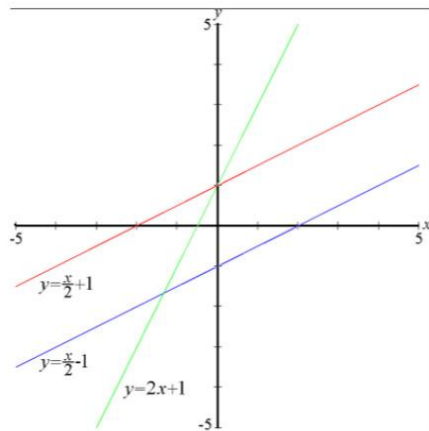


Linear Classifiers

The decision boundary is a **linear function** of the input: in the binary case, it's a line (2 dimensions / features), a plane (3 d) or an hyperplane (n d) separating the two classes

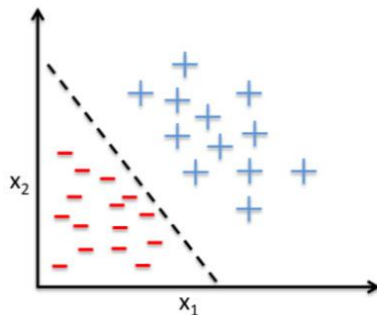
Decision boundary:

$y = w_0 \cdot x_0 + b$: a line

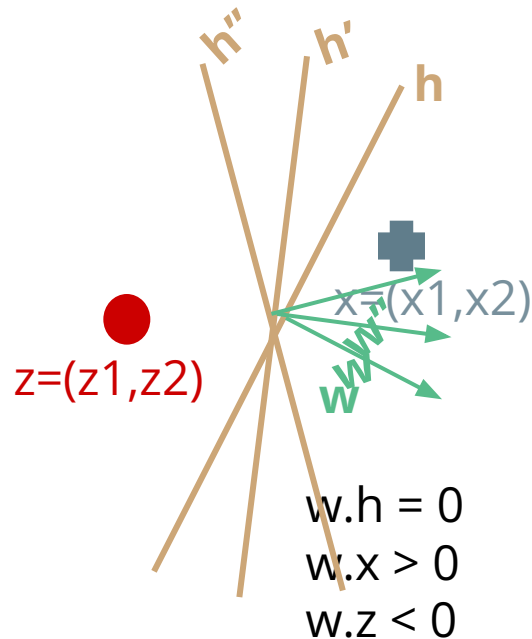


Prediction:

if $\hat{y} = \mathbf{w} \cdot \mathbf{x} > 0$, predict positive class.



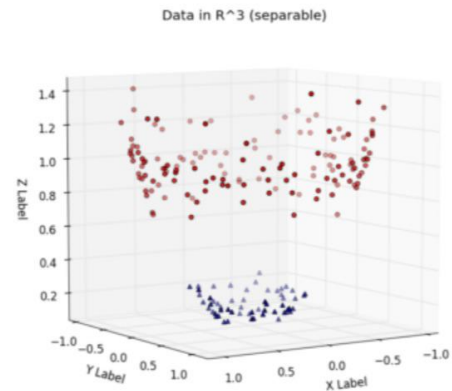
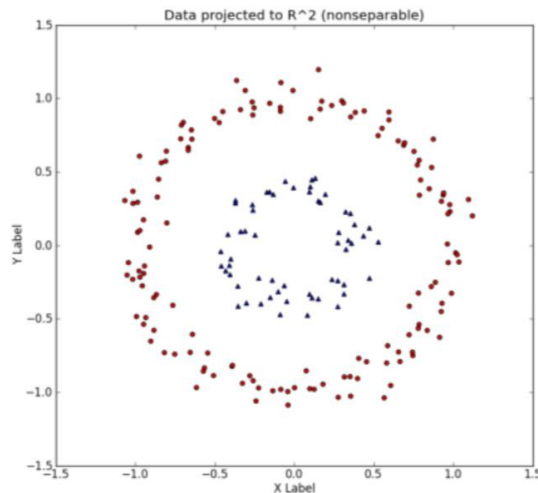
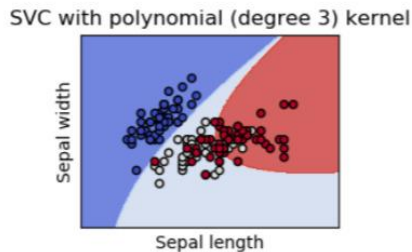
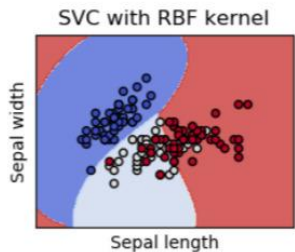
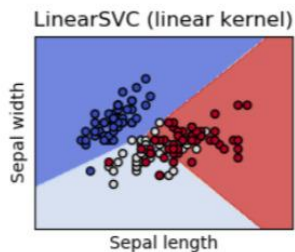
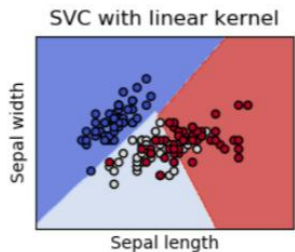
Example of a linear decision boundary for binary classification.



Introducing non-linearity

SVM with non-linear kernel

- mapping of the original input feature space to a **higher-dimensional** feature space,
- with the hope that data may be **linearly separable** in this new space

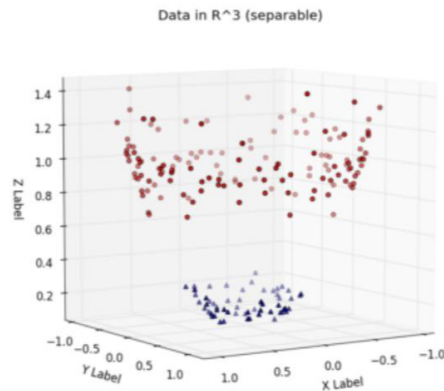
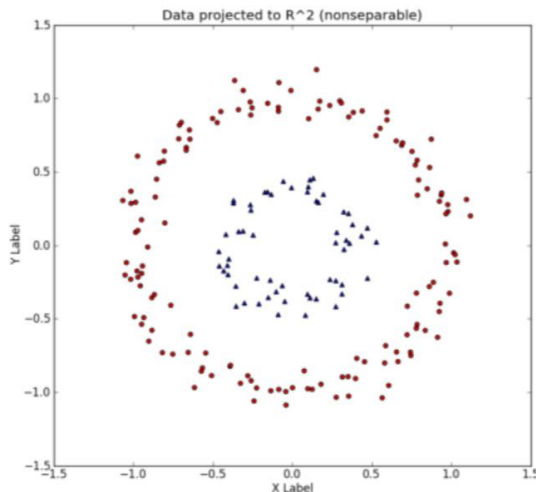
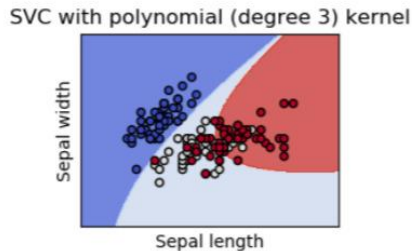
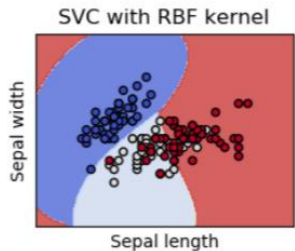
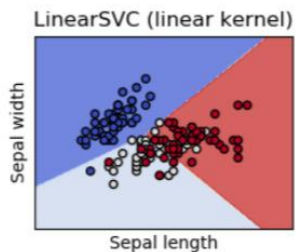
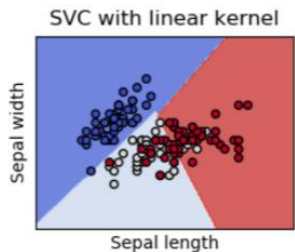


Introducing non-linearity

Neural Network: keep non-linearity and transformation of the input space.

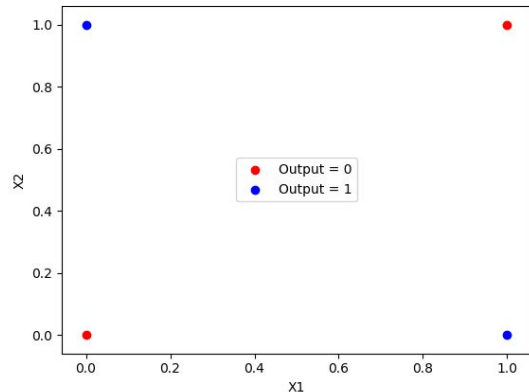
SVM with non-linear kernel

- mapping of the original input feature space to a **higher-dimensional** feature space,
- with the hope that data may be **linearly separable** in this new space



Summary

- Linear functions: a great class of hypothesis for ML, worked for decades
- Non-linearity: seems useful, since many problems are non linear, e.g. XOR problem
- Learning is about solving an optimization problem, i.e. minimizing a function called the loss (while keeping the complexity of the model 'reasonable').



Representation function

- “Feature engineering”:
 - choose features, e.g. words, POS, NE, gaze, meta-data ...
 - represent information (vectorizing, normalizing): bow, n-grams ; TF-IDF, ...

ML → DL: change here = NN seen as representation learners

ML → DL: **sparse vs dense inputs**

Feature representation

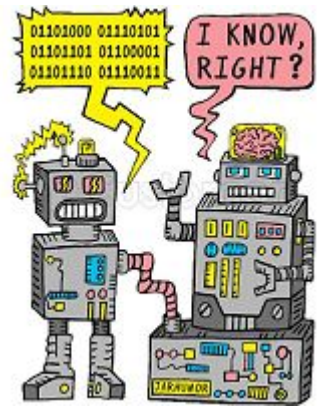
Main issue:

- how to represent text?

e.g. how to transform a sentence into a vector of numerical values?

Bag-of-Words (BOW):

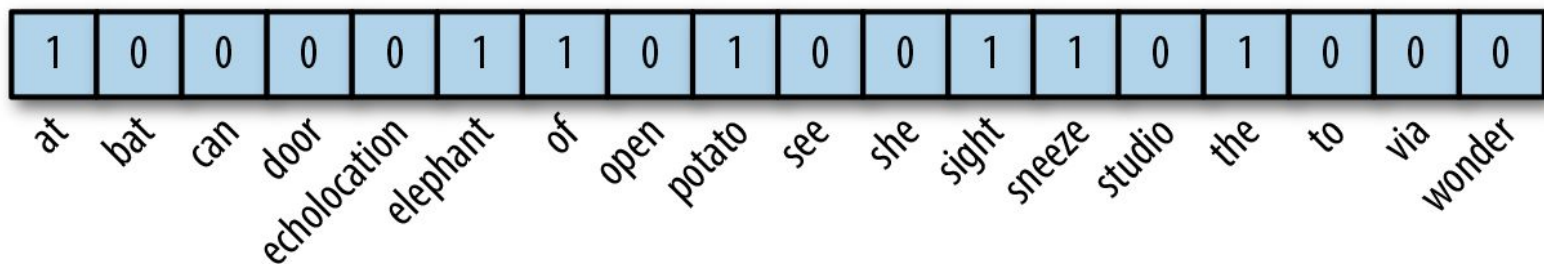
- one vector where **each dimension is a word in our vocabulary**
- if the word / feature is present in the document, associate a specific value



BOW: One-hot encoding

The elephant sneezed
at the sight of potatoes.

- First, build a vocabulary: identify all the word in your data
- If the word is present in the sentence / document, value = 1



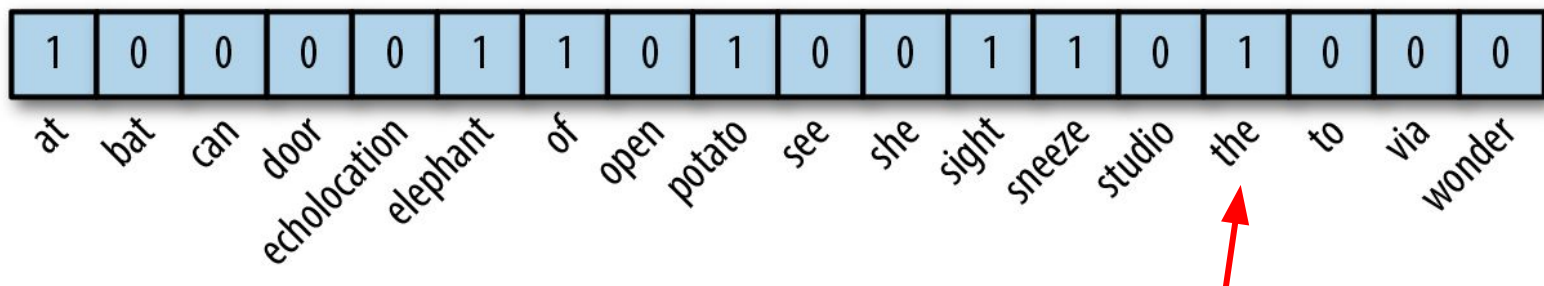
1	0	0	0	0	1	1	0	1	0	0	1	1	0	1	0	0	0
at	bat	can	door	echolocation	elephant	of	open	potato	see	she	sight	sneeze	studio	the	to	via	wonder

18 words / dimensions

BOW: One-hot encoding

The elephant sneezed
at the sight of potatoes.

- First, build a vocabulary: identify all the word in your data
- If the word is present in the sentence / document, value = 1

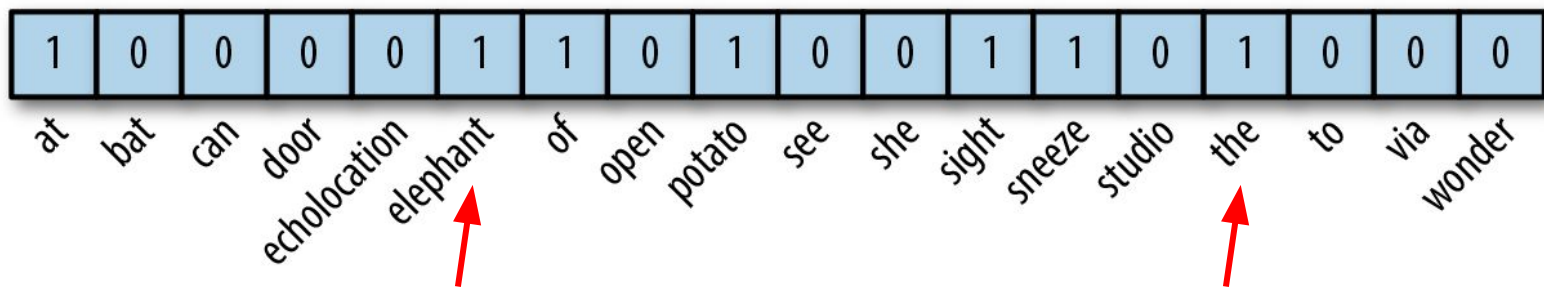


1	0	0	0	0	1	1	0	1	0	0	1	1	0	1	0	0	0
at	bat	can	door	echolocation	elephant	of	open	potato	see	she	sight	sneeze	studio	the	to	via	wonder

BOW: One-hot encoding

The elephant sneezed
at the sight of potatoes.

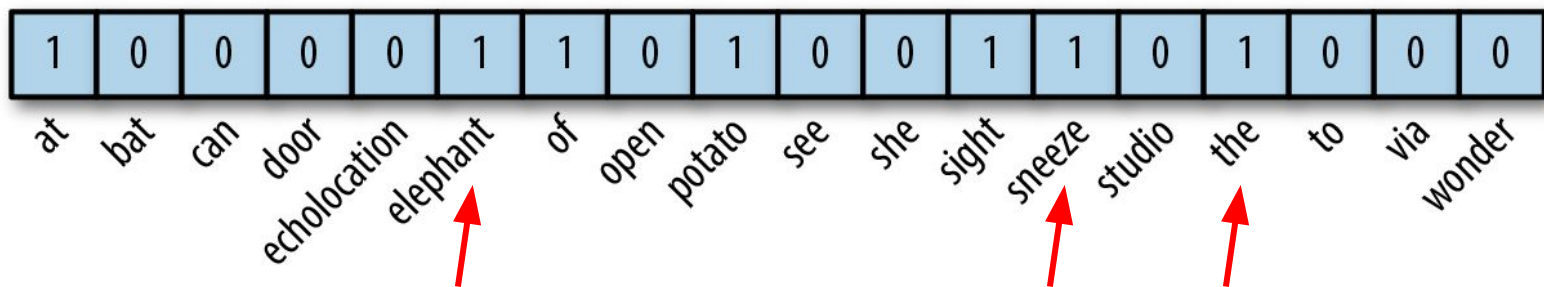
- First, build a vocabulary: identify all the word in your data
- If the word is present in the sentence / document, value = 1



BOW: One-hot encoding

The elephant sneezed
at the sight of potatoes.

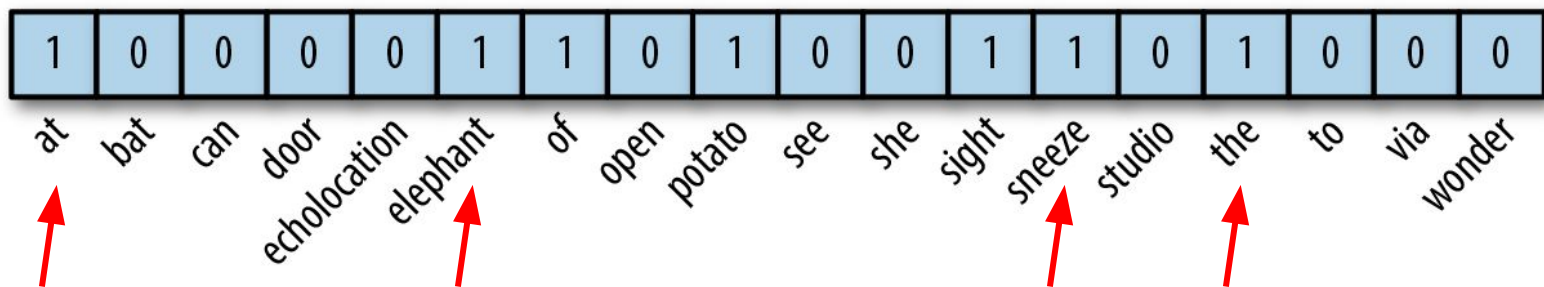
- First, build a vocabulary: identify all the word in your data
- If the word is present in the sentence / document, value = 1



BOW: One-hot encoding

The elephant sneezed
at the sight of potatoes.

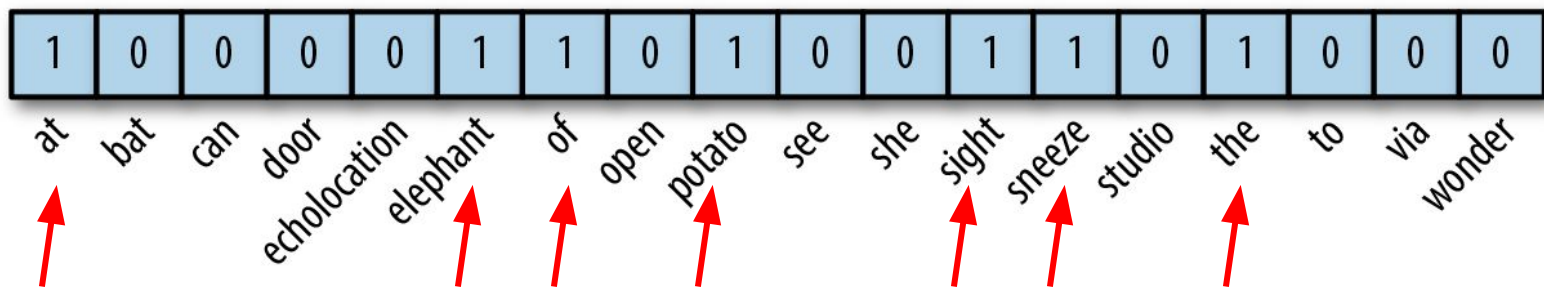
- First, build a vocabulary: identify all the word in your data
- If the word is present in the sentence / document, value = 1



BOW: One-hot encoding

The elephant sneezed
at the sight of potatoes.

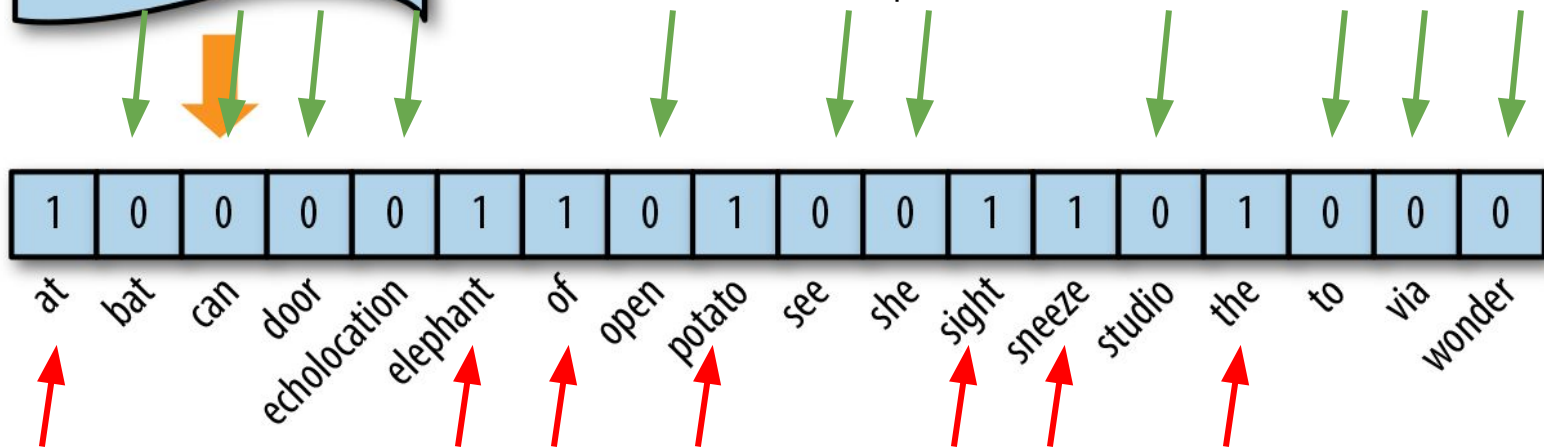
- First, build a vocabulary: identify all the word in your data
- If the word is present in the sentence / document, value = 1



BOW: One-hot encoding

The elephant sneezed
at the sight of potatoes.

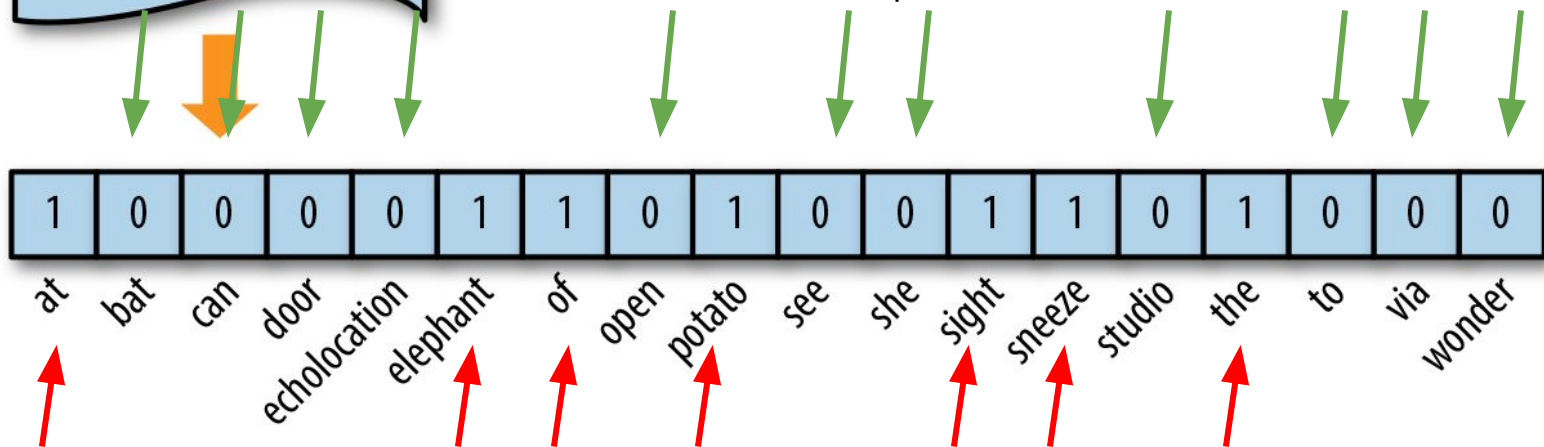
- First, build a vocabulary: identify all the word in your data
- If the word is present in the sentence / document, value = 1
- The other words: value = 0 → present in the training data, but not in this specific sentence / document



BOW: One-hot encoding

The elephant sneezed
at the sight of potatoes.

- First, build a vocabulary: identify all the word in your data
- If the word is present in the sentence / document, value = 1
- The other words: value = 0 → **present in the training data**, but not in this specific sentence / document



Bag-of-Words

Easy to use: now the computer can “read” your sentence

The elephant sneezed at the sight of potatoes.



<1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0>

Varied flavors:

- **Binary**
- Raw frequencies: some words are repeated = more important
- Normalizing with TF-IDF: take into account the distribution of the words in the entire corpus
 - “the”: very frequent but not very crucial
 - “magnificent”: rare, but crucial

Bag-of-Words

Easy to use: now the computer can “read” your sentence

The elephant sneezed at **the** sight of potatoes.



<1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 2, 0, 0, 0>

Varied flavors:

- Binary
- **Raw frequencies**: some words are repeated = more important (?)
- Normalizing with TF-IDF: take into account the distribution of the words in the entire corpus
 - “the”: very frequent but not very crucial
 - “magnificent”: rare, but crucial



Bag-of-Words

$$w_{x,y} = \text{tf}_{x,y} \times \log \left(\frac{N}{\text{df}_x} \right)$$

Easy to use: now the computer can “read” your sentence

TF-IDF

Term x within document y

$\text{tf}_{x,y}$ = frequency of x in y

df_x = number of documents containing x

N = total number of documents

The elephant sneezed at the sight of potatoes.



$\langle 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 2, 0, 0, 0 \rangle$

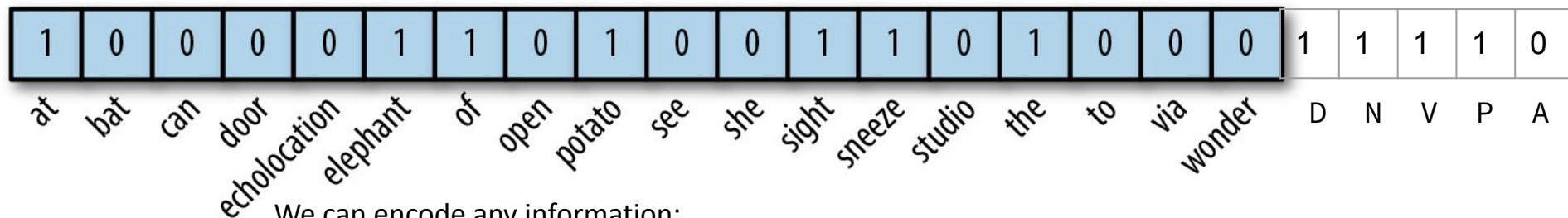
Varied flavors:

- Binary
- Raw frequencies: some words are repeated = more important
- **Normalizing with TF-IDF**: take into account the distribution of the words in the entire corpus
 - “the”: very frequent but not very crucial
 - “magnificent”: rare, but crucial

Bag of any features: one-hot encoding

Can be used to take into account any information, e.g. POS tags:

The/D elephant/N sneezed/V at/P the/D sight/N of/P potatoes/N



1	0	0	0	0	1	1	0	1	0	0	1	1	0	1	0	0	0	1	1	1	1	0
at	bat	can	door	echolocation	elephant	of	open	potato	see	she	sight	sneeze	studio	the	to	via	wonder	D	N	V	P	A

We can encode any information:

- presence of a syntactic relation
- presence of a Named Entity / numbers / dates / amounts
- word associated to a sense if disambiguated
- words in the next sentence
- semantic classes...

Also extra-linguistic features : gender of the writer, number of likes ...

One-hot representation

- Defining features has to be done **manually**: require expertise and tests
- A word is represented with a **one-hot vector**: easy to implement

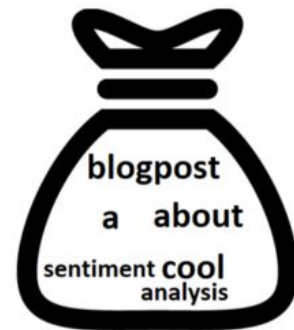
[o o o o o o o o o o o o o o o o **1** o o o o]



[o o o o o o **1** o o o o o o o o o o o o o o]



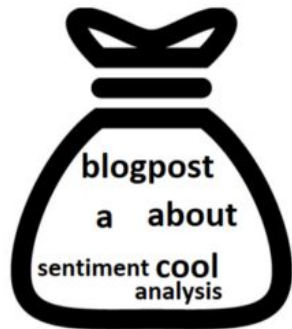
Problems and extensions



1. Very high dimensional:

- 18 dimensions for the previous sentence but could be 100k dimensions!
 - **Curse of dimensionality** (nb of parameters proportional to nb of features) and **sparsity** (many many zeros): makes learning hard, prone to overfitting
- Solutions:
 - ignoring specific words, e.g. stop words
 - keeping only the most frequent / highest TF-IDF
 - grouping words: semantic categories, clusters (Brown)

Problems and extensions



1. Very high dimensional:

- 18 dimensions for the previous sentence but could be 100k dimensions!
 - Curse of dimensionality (nb of parameters proportional to nb of features) and sparsity (many many zeros): makes learning hard, prone to overfitting
- Solutions:
 - ignoring specific words, e.g. stop words
 - keeping only the most frequent / highest TF-IDF
 - grouping words: semantic categories, clusters (Brown)

2. Bag-of-Words representation **ignores word ordering and context**

- crucial: “I don’t know why *I like this movie.*” vs “*I don’t like this movie* and I know why.”
- solutions: n-grams, i.e. use combination of multiple words e.g. trigrams “do not like”...
 - but even more dimensions!

⇒ Representation learning = power of Neural Networks

Basics of OOP (POO in French)

Object Oriented Programming:

- a programming paradigm (vs functional e.g. Camel / Haskell, logic e.g. Prolog, descriptive e.g. LaTeX / HTML...)
- many languages: Ada, C++, **Python**, Java

Python: multi paradigme =

- object but also functional, and structured (hierarchical organization of the code, small pieces of code, extensive use of control and repetition, block structures...)

→ Python libraries are based on the object paradigm: description of **classes** that can be used to perform some operation

Objects

An object is:

- a concept, an idea, an entity in the world
- that as different properties / features / an internal structure
- and also has a specific behavior / a way to interact with other entities
- analogy with real world:
 - we have different types of entities e.g. *cars*, *computers*, *cats*... and also abstract ones such as *time* or a *client*
 - with different 'behaviours': cars start, run, stop, turn ... computers start, bug... cats meow..
- we want to represent them in the computer through their properties and behaviors

Object = a data structure that can answer to specific messages

Objects and classes

Class-based OOP: objects are instances of classes = types

- we have different types of entities in the world, such as cars or cats
- we have different instances of the same type: my cat is different from my neighbors' cat

Class = object type:

- extend the notion of type such as **int**, **char** etc
- used to define the properties of the corresponding entities and how they interact, e.g. **Car**

Object = class instance:

- a specific entity pertaining to a class, e.g. **my_car**

Objects and classes

Class-based OOP: objects are instances of classes = types

- we have different types of entities in the world, such as cars or cats
- we have different instances of the same type: my cat is different from my neighbors' cat



Class = object type:

- extend the notion of type such as int, char etc
- used to define the properties of the corresponding entities and how they interact, e.g.
Car

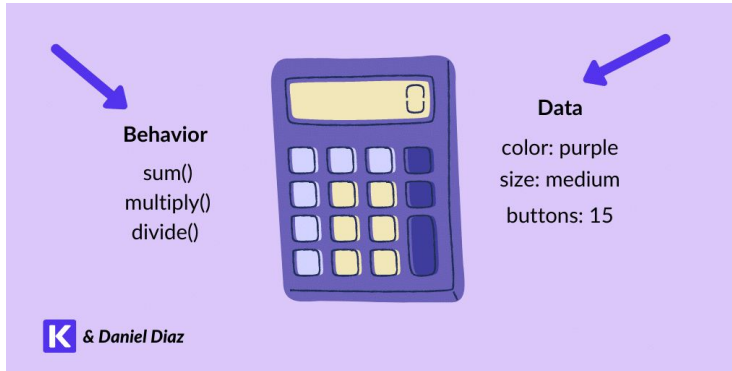
Object = class instance:

- a specific entity pertaining to a class, e.g. **my_car** (of type **Car**)

Classes

→ The modelization step is crucial: how well we define these classes will make for a good, reusable, easy to modify code (or not)

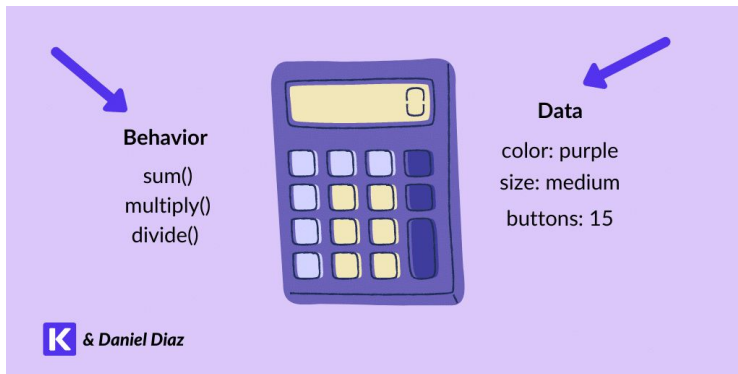
- To define a specific type of object we need to give:
 - a specific collection of data = fields, attributes, properties
 - a specific collection of behaviors = methods, procedures



Classes

→ The modelization step is crucial: how well we define these classes will make for a good, reusable, easy to modify code (or not)

- To define a specific type of object we need to give:
 - a specific collection of data = fields, attributes, properties
 - a specific collection of behaviors = methods, procedures



Note that we have:

- names for attributes
 - verbs for methods
- a very useful convention

Classes in Python

Notebook: <https://colab.research.google.com/drive/1x1cDb0bu87RGFHRavwC2YRhGvNDj1MhK?usp=sharing>

class Car:

Here we define the properties and behavior

def __init__(arguments)

- the **constructor** = the method explaining how to create a new object of this type
- details all the properties / attributes / fields

```
[1] class Car:
    def __init__( self, name, color, year):
        self.name = name
        self.color = color
        self.year = year
```


Classes in Python

Notebook: <https://colab.research.google.com/drive/1oPha9EekRpg5Uvm227xBlumZ0Z3f8UNF?usp=sharing>

class Car:

Here we define the properties and behavior

def __init__(arguments)

- the **constructor** = the method explaining how to create a new object of this type
- details all the properties / attributes / fields

when calling it:

- just use the name of the class = calls the constructor
- the arguments can be used to specify the value for the newly created object

```
[1] class Car:
    def __init__( self, name, color, year):
        self.name = name
        self.color = color
        self.year = year
```



```
my_car = Car( 'Flash McQueen', 'red', 2006 )
```

Classes in Python

Notebook: <https://colab.research.google.com/drive/1oPha9EekRpg5Uvm227xBlumZ0Z3f8UNF?usp=sharing>

class Car:

Here we define the properties and behavior

def __init__(arguments)

- the **constructor** = the method explaining how to create a new object of this type
- details all the properties / attributes / fields

when calling it:

- just use the name of the class = calls the constructor
- the arguments can be used to specify the value for the newly created object

```
[1] class Car:
    def __init__( self, name, color, year):
        self.name = name
        self.color = color
        self.year = year
```



```
my_car = Car( 'Flash McQueen', 'red', 2006 )
```



When I create / instantiate a specific object / instance: I specify the unique shape of my object, its personal attributes

Methods

`def present(self,
arguments):`

I can define a method that describes a possible behavior of my object

```
[1] class Car:
    def __init__( self, name, color, year):
        self.name = name
        self.color = color
        self.year = year

    def present( self ):
        return 'Hey! I am '+self.name+', I am '+self.color+' and I appeared in '+str(self.year)

my_car = Car( 'Flash McQueen', 'red', 2006 )
my_car.present()

'Hey! I amFlash McQueen, I am red and I appeared in 2006'
```

- the method is then 'applied' to an object / an instance of the class
- in its definition, the method can use the fields of the object (here `self.name` or `self.color`)

So, what is **self**?

self is used within a class definition to refer to the current instance, to 'myself'

→ `self.name` ⇒ the name of the instance I'm currently defining

```
[1] class Car:
    def __init__( self, name, color, year):
        self.name = name
        self.color = color
        self.year = year

    def present( self ):
        return 'Hey! I am '+self.name+', I am '+self.color+' and I appeared in '+str(self.year)
```

```
[2] my_car = Car( 'Flash McQueen', 'red', 2006 )
    my_car.present()

'Hey! I am Flash McQueen, I am red and I appeared in 2006'
```

So, what is **self**?

self is used within a class definition to refer to the current instance, to 'myself'

→ `self.name` ⇒ the name of the instance I'm currently defining

Why it is important?

→ we can modify the current object's data fields e.g. `self.color`

```
[1] class Car:
    def __init__( self, name, color, year):
        self.name = name
        self.color = color
        self.year = year

    def present( self ):
        return 'Hey! I am '+self.name+', I am '+self.color+' and I appeared in '+str(self.year)

    def be_painted( self, new_color ):
        self.color = new_color
```

```
[2] my_car = Car( 'Flash McQueen', 'red', 2006 )
    my_car.present()

    'Hey! I am Flash McQueen, I am red and I appeared in 2006'

[3] my_car.be_painted( 'purple' )
    my_car.present()

    'Hey! I am Flash McQueen, I am purple and I appeared in 2006'
```

So, what is **self**?

self is used within a class definition to refers to the current instance, to 'myself'

→ `self.name` ⇒ the name of the instance I'm currently defining

Why it is important?

→ we can modify the current object's data fields

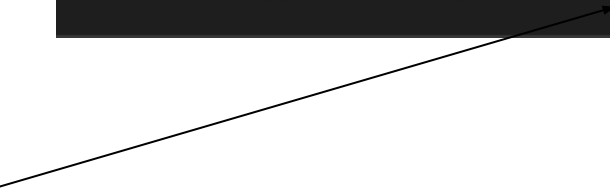
→ we can make use of another instance of the same type within a class definition

```
[1] class Car:
    def __init__( self, name, color, year):
        self.name = name
        self.color = color
        self.year = year

    def present( self ):
        return 'Hey! I am '+self.name+', I am '+self.color+' and I appeared in '+str(self.year)

    def be_painted( self, new_color ):
        self.color = new_color

    def love( self, other_car ):
        return self.name+' is in love with '+other_car.name
```



So, what is **self**?

self is used within a class definition to refers to the current instance, to 'myself'

→ `self.name` ⇒ the name of the instance I'm currently defining

Why it is important?

→ we can modify the current object's data fields

→ we can make use of another instance of the same type within a class definition

```
[1] class Car:
    def __init__( self, name, color, year):
        self.name = name
        self.color = color
        self.year = year

    def present( self ):
        return 'Hey! I am '+self.name+', I am '+self.color+' and I appeared in '+str(self.year)

    def be_painted( self, new_color ):
        self.color = new_color

    def love( self, other_car ):
        return self.name+' is in love with '+other_car.name
```

Exercise...

So, what is **self**?

self is used within a class definition to refers to the current instance, to 'myself'

→ `self.name` ⇒ the name of the instance I'm currently defining

Why it is important?

→ we can modify the current object's data fields

→ we can make use of another instance of the same type within a class definition

```
[1] class Car:
    def __init__( self, name, color, year):
        self.name = name
        self.color = color
        self.year = year

    def present( self ):
        return 'Hey! I am '+self.name+', I am '+self.color+' and I appeared in '+str(self.year)

    def be_painted( self, new_color ):
        self.color = new_color

    def love( self, other_car ):
        return self.name+' is in love with '+other_car.name
```



```
sally = Car( 'Sally Carrera', 'blue', 2006 )
my_car.love( sally )
```

```
'Flash McQueen is in live with Sally Carrera'
```


Summary

- **class**: defines a type of objects, by specifying:
 - its properties = fields ie.
 `self.my_first_property = [initialize with specific value or using argument of the constructor]`
 - its behavior ie.
 `def i_can_do_that(self, [arguments])`
- an **object** (a variable) is an instance of a class that has:
 - specific values for the properties
 - on which we can call all the methods defined in the class

```
class Person(object):  
  
    # __init__ is known as the constructor  
    def __init__(self, name, idnumber):  
        self.name = name  
        self.idnumber = idnumber  
  
    def display(self):  
        print(self.name)  
        print(self.idnumber)  
  
    def details(self):  
        print("My name is {}".format(self.name))  
        print("IdNumber: {}".format(self.idnumber))
```

Summary

- **class**: defines a type of objects, by specifying:
 - its properties = fields ie.
`self.my_first_property = [initialize with specific value or using argument of the constructor]`
 - its behavior ie.
`def i_can_do_that(self, [arguments])`
- an **object** (a variable) is an instance of a class that has:
 - specific values for the properties
 - on which we can call all the methods defined in the class

```
class Person(object):  
  
    # __init__ is known as the constructor  
    def __init__(self, name, idnumber):  
        self.name = name  
        self.idnumber = idnumber  
  
    def display(self):  
        print(self.name)  
        print(self.idnumber)  
  
    def details(self):  
        print("My name is {}".format(self.name))  
        print("IdNumber: {}".format(self.idnumber))
```

Side note: [Python](#) recommends UpperCamelCase for class names, CAPITALIZED_WITH_UNDERSCORES for constants, and snake_case for other names.

Inheritance

- used to define types and subtypes
- parent class: the most abstract / prototypical
- child class(es): implement distinct features

```
class Person(object):  
  
    # __init__ is known as the constructor  
    def __init__(self, name, idnumber):  
        self.name = name  
        self.idnumber = idnumber  
  
    def display(self):  
        print(self.name)  
        print(self.idnumber)  
  
    def details(self):  
        print("My name is {}".format(self.name))  
        print("IdNumber: {}".format(self.idnumber))
```

Parent class:
A person has:

- a name
- an idnumber

```
class Employee(Person):  
    def __init__(self, name, idnumber, salary, post):  
        super().__init__(name, idnumber)  
        self.salary = salary  
        self.post = post  
  
    def details(self):  
        print("My name is {}".format(self.name))  
        print("IdNumber: {}".format(self.idnumber))  
        print("Post: {}".format(self.post))
```

Child class:
An employee also has:

- a name
- an idnumber

but in addition it has:

- a salary
- a post

Inheritance

- used to define types and subtypes
- parent class: the most abstract / prototypical
- child class(es): implement distinct features

```
class Person(object):  
  
    # __init__ is known as the constructor  
    def __init__(self, name, idnumber):  
        self.name = name  
        self.idnumber = idnumber  
  
    def display(self):  
        print(self.name)  
        print(self.idnumber)  
  
    def details(self):  
        print("My name is {}".format(self.name))  
        print("IdNumber: {}".format(self.idnumber))
```

Parent class:
A person has:

- a name
- an idnumber

here is where you say
that it's a child class of
Person

```
class Employee(Person):  
    def __init__(self, name, idnumber, salary, post):  
        super().__init__(name, idnumber)  
        self.salary = salary  
        self.post = post  
  
    def details(self):  
        print("My name is {}".format(self.name))  
        print("IdNumber: {}".format(self.idnumber))  
        print("Post: {}".format(self.post))
```

Child class:
An employee also has:

- a name
- an idnumber

but in addition it has:

- a salary
- a post

Inheritance

- used to define types and subtypes
- parent class: the most abstract / prototypical
- child class(es): implement distinct features
- + you can call the constructor of the parent to fill the corresponding fields using super

```
class Person(object):  
  
    # __init__ is known as the constructor  
    def __init__(self, name, idnumber):  
        self.name = name  
        self.idnumber = idnumber  
  
    def display(self):  
        print(self.name)  
        print(self.idnumber)  
  
    def details(self):  
        print("My name is {}".format(self.name))  
        print("IdNumber: {}".format(self.idnumber))
```

Parent class:
A person has:

- a name
- an idnumber

here is where you say
that it's a child class of
Person

```
class Employee(Person):  
    def __init__(self, name, idnumber, salary, post):  
        super().__init__(name, idnumber)  
        self.salary = salary  
        self.post = post  
  
    def details(self):  
        print("My name is {}".format(self.name))  
        print("IdNumber: {}".format(self.idnumber))  
        print("Post: {}".format(self.post))
```

Child class:
An employee also has:

- a name
- an idnumber

but in addition it has:

- a salary
- a post

Inheritance

```
class Person(object):  
  
    # __init__ is known as the constructor  
    def __init__(self, name, idnumber):  
        self.name = name  
        self.idnumber = idnumber  
  
    def display(self):  
        print(self.name)  
        print(self.idnumber)  
  
    def details(self):  
        print("My name is {}".format(self.name))  
        print("IdNumber: {}".format(self.idnumber))
```

Parent
class:
A person has:

- a name
- an idnumber

Exercise...

```
class Employee(Person):  
    def __init__(self, name, idnumber, salary, post):  
        super().__init__(name, idnumber)  
        self.salary = salary  
        self.post = post  
  
    def details(self):  
        print("My name is {}".format(self.name))  
        print("IdNumber: {}".format(self.idnumber))  
        print("Post: {}".format(self.post))
```

Child class:
An employee also has:

- a name
- an idnumber

but in addition it has:

- a salary
- a post

Inheritance

```
class Person(object):  
  
    # __init__ is known as the constructor  
    def __init__(self, name, idnumber):  
        self.name = name  
        self.idnumber = idnumber  
  
    def display(self):  
        print(self.name)  
        print(self.idnumber)  
  
    def details(self):  
        print("My name is {}".format(self.name))  
        print("IdNumber: {}".format(self.idnumber))
```

Parent
class:
A person has:

- a name
- an idnumber

```
class Employee(Person):  
    def __init__(self, name, idnumber, salary, post):  
        super().__init__(name, idnumber)  
        self.salary = salary  
        self.post = post  
  
    def details(self):  
        print("My name is {}".format(self.name))  
        print("IdNumber: {}".format(self.idnumber))  
        print("Post: {}".format(self.post))
```

Child class:
An employee also has:

- a name
- an idnumber

but in addition it has:

- a salary
- a post

```
# creation of an object variable or an instance  
a = Employee('Rahul', 886012, 200000, "Intern")  
  
# calling a function of the class Person using  
# its instance  
a.display()  
a.details()
```

Note that:

- We can also call the method from the class Person on an employee object

Inheritance

```
class Person(object):  
  
    # __init__ is known as the constructor  
    def __init__(self, name, idnumber):  
        self.name = name  
        self.idnumber = idnumber  
  
    def display(self):  
        print(self.name)  
        print(self.idnumber)  
  
    def details(self):  
        print("My name is {}".format(self.name))  
        print("IdNumber: {}".format(self.idnumber))
```

Parent
class:
A person has:

- a name
- an idnumber

```
# creation of an object variable or an instance  
a = Employee('Rahul', 886012, 200000, "Intern")  
  
# calling a function of the class Person using  
# its instance  
a.display()  
a.details()
```

Note that:

- We can also call the method from the class Person on an employee object
- We can **redefine** a method: here details() is defined in both classes, but here it's the version in Employee class that will be used

```
class Employee(Person):  
    def __init__(self, name, idnumber, salary, post):  
        super().__init__(name, idnumber)  
        self.salary = salary  
        self.post = post  
  
    def details(self):  
        print("My name is {}".format(self.name))  
        print("IdNumber: {}".format(self.idnumber))  
        print("Post: {}".format(self.post))
```

Child class:
An employee also has:

- a name
- an idnumber

but in addition it has:

- a salary
- a post

Summary

- **class**: defines a type of objects, by specifying:
 - its properties = fields ie.
 `self.my_first_property = [initialize with specific value or using argument of the constructor]`
 - its behavior ie.
 `def i_can_do_that(self, [arguments])`
- an **object** (a variable) is an instance of a class that has:
 - specific values for the properties
 - on which we can call all the methods defined in the class

+ an object has the properties and the methods of his parents

```
class Person(object):  
  
    # __init__ is known as the constructor  
    def __init__(self, name, idnumber):  
        self.name = name  
        self.idnumber = idnumber  
  
    def display(self):  
        print(self.name)  
        print(self.idnumber)  
  
    def details(self):  
        print("My name is {}".format(self.name))  
        print("IdNumber: {}".format(self.idnumber))
```

Ok, so why does that matter?...

They are all
classes

`sklearn.linear_model`: Linear Models

The `sklearn.linear_model` module implements a variety of linear models.

User guide: See the [Linear Models](#) section for further details.

The following subsections are only rough guidelines: the same estimator can fall into multiple categories, depending on its parameters.

Linear classifiers

<code>linear_model.LogisticRegression([penalty, ...])</code>	Logistic Regression (aka logit, MaxEnt) classifier.
<code>linear_model.LogisticRegressionCV(*[, Cs, ...])</code>	Logistic Regression CV (aka logit, MaxEnt) classifier.
<code>linear_model.PassiveAggressiveClassifier(*)</code>	Passive Aggressive Classifier.
<code>linear_model.Perceptron(*[, penalty, alpha, ...])</code>	Linear perceptron classifier.
<code>linear_model.RidgeClassifier([alpha, ...])</code>	Classifier using Ridge regression.
<code>linear_model.RidgeClassifierCV([alphas, ...])</code>	Ridge classifier with built-in cross-validation.
<code>linear_model.SGDClassifier([loss, penalty, ...])</code>	Linear classifiers (SVM, logistic regression, etc.) with SGD training.
<code>linear_model.SGDOneClassSVM([nu, ...])</code>	Solves linear One-Class SVM using Stochastic Gradient Descent.

`sklearn.linear_model.LogisticRegression`

```
class sklearn.linear_model.LogisticRegression(penalty='l2', *, dual=False, tol=0.0001, C=1.0, fit_intercept=True,
intercept_scaling=1, class_weight=None, random_state=None, solver='lbfgs', max_iter=100, multi_class='auto', verbose=0,
warm_start=False, n_jobs=None, l1_ratio=None)
```

[\[source\]](#)

Same in PyTorch

```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

```
class Net(nn.Module):
```

```
    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120) # 5*5 from image dimension
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square, you can specify with a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = torch.flatten(x, 1) # flatten all dimensions except the batch dimension
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

```
net = Net()
print(net)
```

- Here we define a class that defines a specific type of network + it inherits from the class nn.Module

Same in PyTorch

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120) # 5*5 from image dimension
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square, you can specify with a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = torch.flatten(x, 1) # flatten all dimensions except the batch dimension
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

net = Net()
print(net)
```

- Here we define a class that defines a specific type of network + it inherits from the class `nn.Module`
- The constructor tells us that this network has
 - different properties: `conv1` and `conv2`, `fc1`, `fc2` and `fc3`
 - a method called `forward(..)`

Same in PyTorch

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120) # 5*5 from image dimension
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square, you can specify with a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = torch.flatten(x, 1) # flatten all dimensions except the batch dimension
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

net = Net()
print(net)
```

- Here we define a class that defines a specific type of network + it inherits from the class nn.Module
- The constructor tells us that this network has different properties: conv1 and conv2, fc1, fc2 and fc3 ; and that it has a method called forward(..)
- Here we instantiate an object of this class, thus a concrete network of this type

TP1: Sentiment analysis with Scikit

In the practical session, we will implement a system for sentiment classification of movie reviews.

- pre-process data (BoW, n-grams)
- train and evaluate a model
- compare different algorithms
- investigate model decisions

<https://colab.research.google.com/drive/1icJsbnjykYRpvNiJYJDDzip9RVWVhL-O?usp=sharing>

Sources

- *Foundations of Machine Learning*, Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar, MIT Press
- Comparing SVM and NN:
 - **Short answer:** On small data sets, SVM might be preferred.
<https://stats.stackexchange.com/questions/510052/are-neural-networks-better-than-svms>
 - <https://www.baeldung.com/cs/svm-vs-neural-network>
- https://dair.ai/notebooks/nlp/2020/03/19/nlp_basics_tokenization_segmentation.html
- <https://www.infoq.com/presentations/nlp-practitioners/>
- <https://github.com/sebastianruder/NLP-progress>
- *Aleatoric and epistemic uncertainty in machine learning: an introduction to concepts and methods*.
Hüllermeier, E., Waegeman, W. *Mach Learn* **110**, 457–506 (2021).
<https://doi.org/10.1007/s10994-021-05946-3> (Picture on hypothesis space)
-