Introduction to Machine Learning

k-NN and the geometry of learning

Guillaume Wisniewski
guillaume.wisniewski@limsi.fr
January 2018
Université Paris Sud — LIMSI

Recap: what we saw in the last lab...

The classifier

Goal

- binary classification (positive/negative review)
- decision tree of given depth
- measure precision on train/dev/test sets

Decision tree

- to predict a label: path from root to leaf
- path = conjunction of features (e.g. "bad" $\wedge \neg$ "not")
- the deeper the tree, the more complex the decision
- depth = measure of the capacity of the classifier

The code i

```
import numpy as np

import matplotlib.pyplot as plt
from matplotlib.backends.backend_agg import FigureCanvasAgg as FigureCa
from matplotlib.figure import Figure

from sklearn.tree import DecisionTreeClassifier
from data import loadTextDataBinary


plt.style.use('ggplot')

X_train, Y_train, dictionary = loadTextDataBinary('data/sentiment.tr')
X_test, Y_test, _ = loadTextDataBinary('data/sentiment.te', dictionary)
X_dev, Y_dev, _ = loadTextDataBinary('data/sentiment.de', dictionary)
```

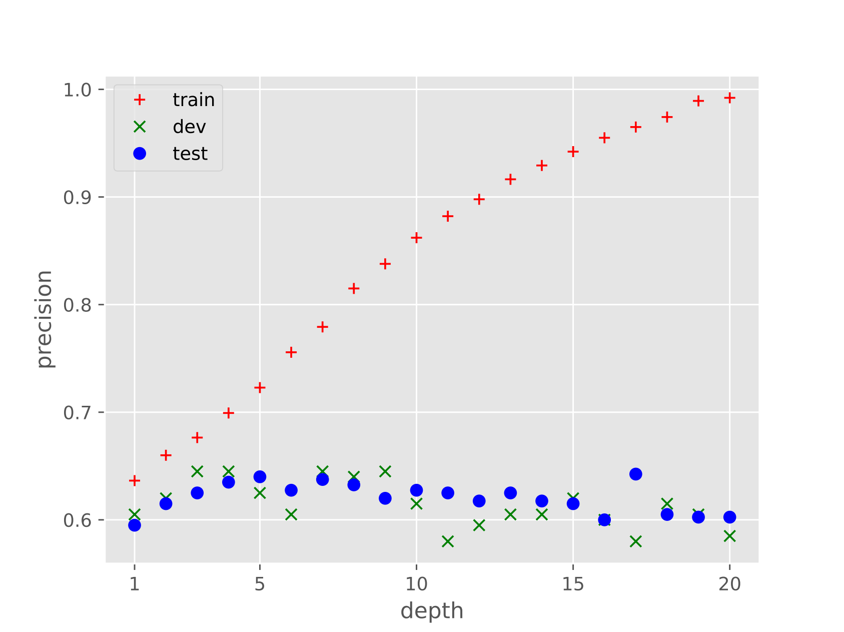
The code ii

```
def evaluate(depth):
    clf = DecisionTreeClassifier(max_depth=depth)
    clf.fit(X_train, Y_train)

    return np.mean(clf.predict(X_train) == Y_train), \
        np.mean(clf.predict(X_dev) == Y_dev), \
        np.mean(clf.predict(X_test) == Y_test)

precisions = [evaluate(d) for d in range(1, 21)]

p_train, p_dev, p_test = zip(*precisions)
x = np.arange(1, len(p_train) + 1)

fig = Figure()
FigureCanvas(fig)
```

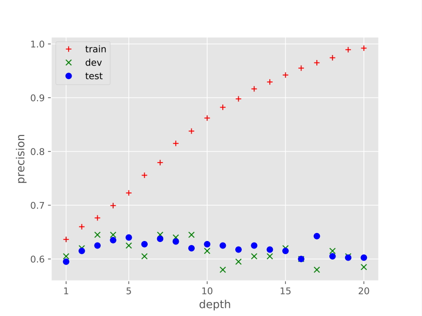

The code iii

```
ax = fig.add_subplot(111)
ax.plot(x, p_train, "r+", label="train")
ax.plot(x, p_dev, "gx", label="dev")
ax.plot(x, p_test, "bo", label="test")
ax.set_xlabel("depth")
ax.set_ylabel("precision")
ax.legend()

fig.savefig("learning_curve.pdf")
```

Result



Interpretation



- increasing the capacity ⇒ increasing the precision on the train set
- it is always possible to achieve 0 error on trainset by increasing the capacity
- but, at some point, precision on test & dev set start decreasing ⇒ overfitting

Why having a ‘dev’ and ‘test’ sets

The context

- main idea: evaluate classifier on ‘unseen’ examples ⇒ generalization error (expected loss)
- train set: choose the parameters of the algorithm
- here: hyper-parameter (depth) ⇒ must be chosen before training
- how?

A primer on learning theory

‘theorem’: if no parameter and no-hyperparameter is chosen by considering information coming from the test set (e.g. error on test set), then the error on the test set is a ‘good approximation’ on the generalization error.

In practice...

1. choose several depths (exhaustive search)
2. train a model for each depth (on the train set)
3. evaluate their loss on the dev set
4. choose the model with the smallest error
5. evaluate the error on the test set ϵ_{test}

ϵ_{test} is ‘independent’ of the choice of the test set

k-nearest neighbors

First example: predicting the state of water

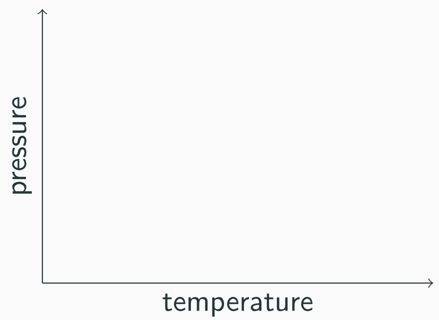
The problem

- water can be in three states: liquid, solid, gas
- depends on temperature and pressure

Your first physic lab

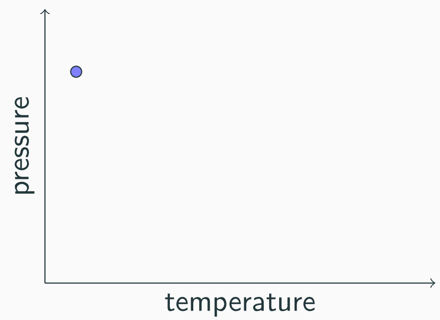
- change temperature and/or pressure
- see state
- plot your data

Plotting



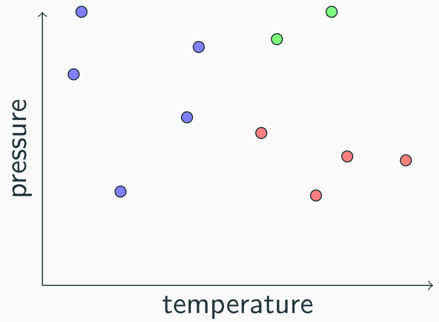
- first measure:
  - temperature = 90
  - pressure = 2
  - state = liquid

Plotting



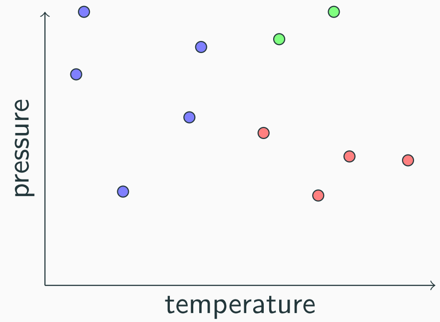
- first measure:
  - temperature = 90
  - pressure = 2
  - state = liquid
- features & feature value

Plotting



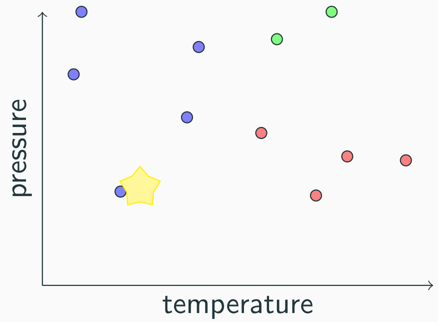
- first measure:
  - temperature = 90
  - pressure = 2
  - state = liquid
- features & feature value
- many other measures: liquid, solid and gas

Plotting



- first measure:
  - temperature = 90
  - pressure = 2
  - state = liquid
- features & feature value
- many other measures: liquid, solid and gas
- state when temperature is 20 and pressure 0.1 ?

Plotting



- first measure:
  - temperature = 90
  - pressure = 2
  - state = liquid
- features & feature value
- many other measures: liquid, solid and gas
- state when temperature is 20 and pressure 0.1 ? $\Rightarrow$ plot the corresponding point and look for the state of the closest point

Bird's-eye view

Examples

- supervised classification: N labeled observations $(\mathbf{x}^i, y^i)_{i=1}^N$
- observation = point/vector of $\mathbb{R}^2 = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$

The geometric view of learning

- Euclidean space $\Rightarrow$ geometric notions (distance, angle, ...)
- distance between 2 points = similarity between 2 observations
- inductive bias: if two observations are 'similar', they have the same label

More formally

Notations

- train set: $\mathcal{D} = (\mathbf{x}^i, y^i)_{i=1}^{N}$
- with $\mathbf{x}^i = \begin{pmatrix} x_1^i \\ x_2^i \end{pmatrix}$ and $y_i \in \{\text{liquid}, \text{gas}, \text{solid}\}$

***k*-nearest neighbors**

- training: do nothing
- test:
  - new example $\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$
  - find closest example of the training set: $\mathbf{x}_c = \arg\min_{\mathbf{x}^i \in \mathcal{D}} d(\mathbf{x}, \mathbf{x}^i)$
  - return label of $\mathbf{x}_c$

Distance...

Usually:

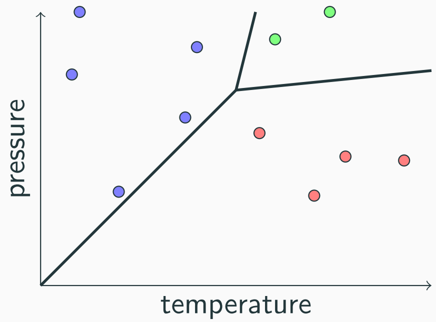
$$d(\mathbf{x}, \mathbf{y}) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2} \tag{1}$$

$\Rightarrow$ Euclidean distance

Many other distance exists:

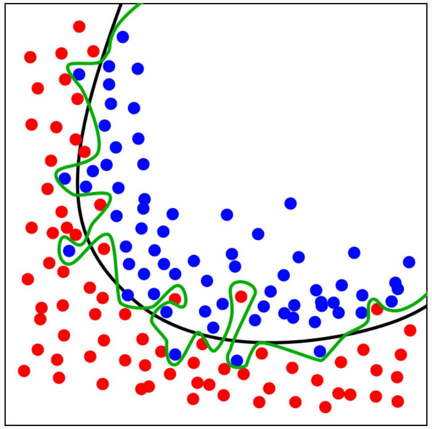
- radial distance
- Manhattan distance
- ...

Decision boundaries

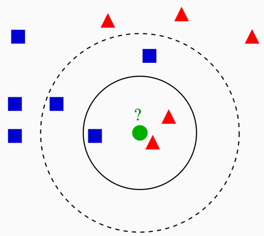


- decision boundaries = points that are equidistant o
- good way to visualize the complexity of a model

Example



Robustness to noise

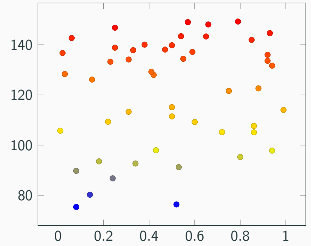


- rather than considering the closest observation, consider the k closest neighbors
  - deal with 'errors' in training data
- majority label between the k closest neighbors

Impact of feature scale: the problem

The context

- 2 features :
  - $x_1 \in [0, 1]$
  - $x_2 \in [50, 150]$

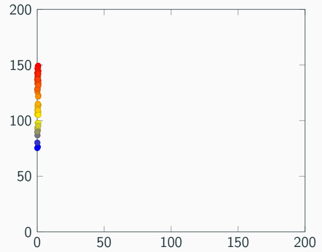


As usual...

Impact of feature scale: the problem

The context

- 2 features :
  - $x_1 \in [0, 1]$
  - $x_2 \in [50, 150]$



Without ‘cheating’

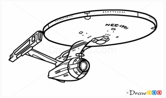
$\Rightarrow$ normalize feature value

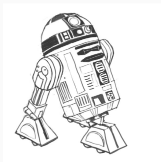
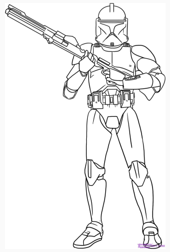
An important detail

- in k-nn all features are considered equal
- even if they are not relevant
- without considering their scale

Limits of k-NN

A new task

Star Wars or Star Trek ?

How to deal with it?

- image = list of pixels (we can assume they are all made of 10,000 pixels)
- image = vector in dimension $I = 10,000 \times 3$
- we can apply the k-nn algorithm: $d(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{I} (x_i - y_i)^2}$

Curse of dimensionality

The problem...

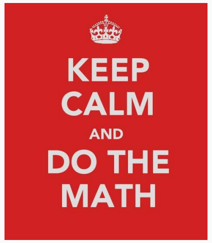
- are two Star Wars images really close? e.g. when considering the number of pixel they have in common
- given a maximal distance d (e.g. you can change n pixels) and an image i, if you can build i' so that $|i - i'| = d$, what will i' look like?

$\Rightarrow$ how many images do you need to 'cover' the whole space?

'Formalizing' these intuitions?

- what is the distribution of distance of random points in a D-dim space?

Let's do the math



The problem

$$\mathbb{E}_{a\sim\mathrm{Uni}(D)}\left[\mathbb{E}_{b\sim\mathrm{Uni}(D)}\left[|a-b|\right]\right] \qquad (2)$$

- this can be solved analytically ($a-b$ follows a Irwin–Hall distribution)

The solution

- the maximal distance in a D dimensional space is $\sqrt{D}$
- the average distance between two points in a D dimensional space is $\frac{\sqrt{D}}{3}$
- its variance is $\frac{1}{\sqrt{18}}$

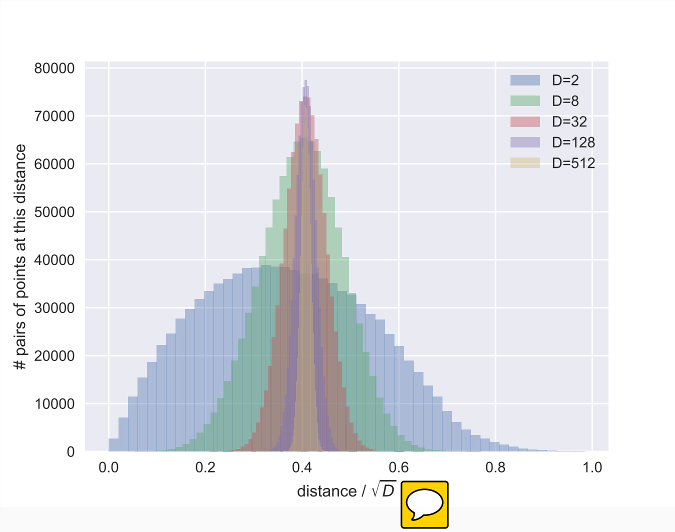
Show me the code i

```
import numpy as np

from itertools import product

from numpy.random import uniform

import seaborn as sns
sns.set(color_codes=True)

n_points = 1000
for dim in [2, 8, 32, 128, 512]:

    # lazzy generation of the points (generator)
    data = (uniform(0, 1, (dim,)) for i in range(n_points))
    # numpy.linalg.norm(a-b) would be faster
```

Show me the code ii

```
    # this version is only for `educationnal' purpose
    distances = [np.sqrt(np.sum((a-b)**2)) / np.sqrt(dim)
                 for a, b in product(data, repeat=2) if a is not b]

    ax = sns.distplot(distances, kde=False, label="D={}".format(dim))

ax.legend()
ax.set_xlabel("distance / $\sqrt{D}$")
ax.set_ylabel("# pairs of points at this distance")

fig = ax.get_figure()

fig.savefig("dist_distrib.pdf")
```


Result



Conclusion

k-nn classifier

- as D increases, all points start to be at the same distance
- but this is the only information a k-nn classifier has!

More generally...

- large space are strange
- many of our intuitions in 2d or 3d are no longer true

A solution...

A new task

Fire truck or 'regular' truck?

How to train a classifier?

- consider the pixels? $\Rightarrow$ curse of dimensionality
- consider the number of red pixels? $\Rightarrow$ 'easier' decision

What we have just do?

Feature extraction

- change the representation of the picture
- all pixels *versus* a 'small' number of handcrafted, high-level features

Central idea

- design arbitrary features that are relevant for the task
- allow to reduce the 'dimension' of examples
- why: 'simple' decision

Formally

- $\mathcal{X}$:
  - input space
  - whatever object we consider (text, mail, image, sound, network, ...)
- all object are mapped to a vector of features:

$$\phi : \mathcal{X} \mapsto \mathbb{R}^n \tag{3}$$

  - $\phi(x)$ is a vector of reals
  - $\phi(x)_i$, its i-th coordinate, is a feature value (e.g. number of words in a mail, proportion of red pixels, ...)
- ϕ = hand-coded function = where the 'intelligence'/'knowledge' used to be

On the importance of good features...

- algorithm have only access to feature values and not the semantic of the feature
- permuting all feature vectors will not affect training as long as all examples are permuted in the same way

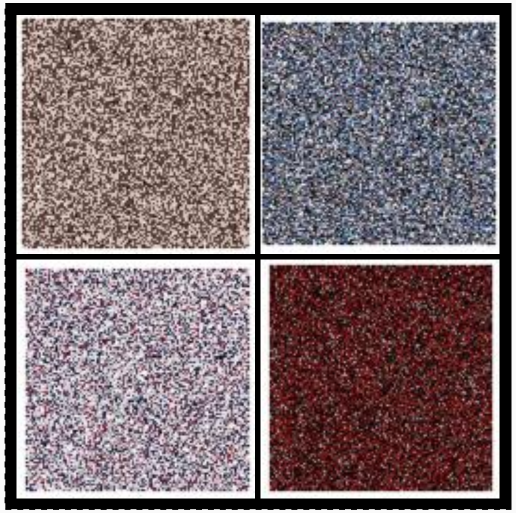
Example

Do you recognize these images?

(Original images)

Example

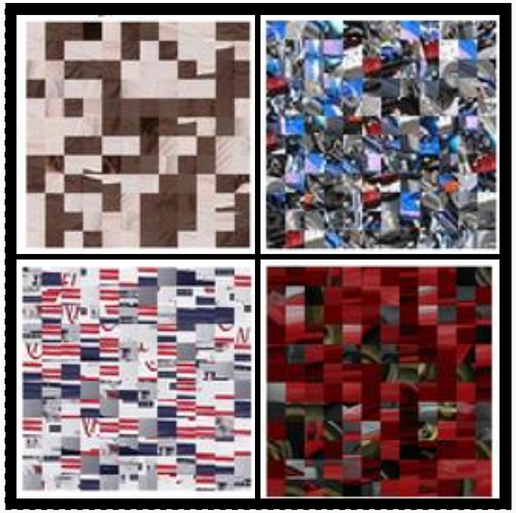
Do you recognize these images?



(random permutation of pixels)

Example

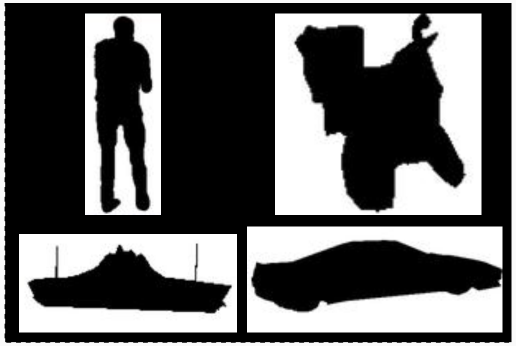
Do you recognize these images?



(random permutation of patches) (shape detection)

Example

Do you recognize these images?

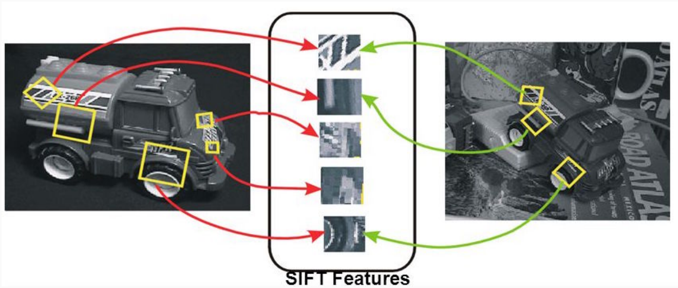


Consequence

- A lot of effort has been put into designing ‘good’ features
- e.g.:
  - image: SIFT features (Scale-invariant feature transform)
  - sound: MFCC (Mel-frequency cepstral coefficients)
  - text: list of ‘relevant’ words

Example: idea of SIFT

Transformation of an image into local features that are invariant to translation, rotation, scale and other imaging parameters



Warning

In practice...

In practice $\mathcal{X} = \mathbb{R}^n$: machine learning starts after all features have been extracted

Note for the future...

- no longer true (at least partially)
- 'deep learning' feature are extracted automatically during training

Conclusions

What do you have to remember?

- examples = point in an Euclidean space
- how to compute a distance + implement k-nn
- curse of dimensionality
- feature extraction