Decision trees in machine learning are related to the game of twenty questions, where one tries to guess an object by asking less than twenty questions.

Q: Is it a human?
A: Yes

Q: Is (s)he alive?
A: No

Q: Is (s)he a man?
A: Yes

Q: Was he European?
...
1. The full strategy can be defined before playing, from the statistics we have about the most likely choices of people.

2. It can be unfolded as a tree:

```
Is it a human?
  No
    ...
  Yes
    Is (s)he alive?
      No
        ...
      Yes
        Is (s)he a man?
          No
            ...
          Yes
            ...
```

Decision trees
In a nutshell

For prediction, decision trees can be seen as a simple form of adaptive testing: The next property to check depends on which you have already tested and the answers you got.

For training, as for twenty questions, one tries to pick the most efficient questions to be able to predict the unknown state as quickly as possible.

Decision trees
Why do we like them

Despite their old age, and their simplicity, (random forests of) decision trees work really well.


*We evaluate 179 classifiers arising from 17 families (discriminant analysis, Bayesian, neural networks, support vector machines, decision trees, rule-based classifiers, boosting, /…/)

*The random forest is clearly the best family of classifiers (3 out of 5 bests classifiers are RF), /…/*

Decision trees
Notation

For the rest of the course, we will mainly focus on classification from continuous features.

Let $X = (X^1, \ldots, X^D)$ be a random variable on $\mathbb{R}^D$ standing for the vector of measurements, and $Y$ a random variable on $\{1, \ldots, C\}$ the unknown class to predict.

Let $X_1, \ldots, X_T$ and $Y_1, \ldots, Y_T$ be the training set.

The purpose of that lecture is to predict $Y$ from $X$ with decision trees built from the training set.
Decision trees
Definition

We define a decision tree recursively as being either

1. A leaf carrying a label from \( \{1, \ldots, C \} \), or
2. An internal node carrying a Boolean test \( q : \mathbb{R}^D \rightarrow \{0, 1\} \) and two decision trees \( T_0 \) and \( T_1 \) of finite depth.

\[
\begin{align*}
X^3 & \geq 0.5 \\
0 & \quad \quad \quad \quad \quad \quad \quad X^1 & \geq -1.2 \quad \quad \quad \quad \quad \quad \quad 0 \quad \quad \quad \quad \quad \quad \quad 1
\end{align*}
\]

Decision trees
Evaluation

The prediction of a tree given a vector of measurement \( X \), is

1. For a leaf: the label it holds,
2. For an internal node: the prediction of \( T_{q(X)} \).
To fully define the trees we are manipulating, we have to chose a set of binary tests $\mathcal{B}$. Then, given the training set

$$(X_1, Y_1), \ldots, (X_T, Y_T),$$

building the decision tree is also a recursive procedure:

1. If all $Y_t$ are equal, the tree is a leaf with the common value of the $Y_t$ as label,
2. else, it is an internal node: pick a $q \in \mathcal{B}$, split the training set in two sub-sets and build $T_0$ and $T_1$ from these subsets.

The two main issues remaining are the choice of $\mathcal{B}$ and the procedure to pick $q$.

**Decision trees**

**Set of binary tests**

We will consider binary tests parametrized by a vector $u \in \mathbb{R}^D$ and a threshold $t \in \mathbb{R}$ and defined by

$$q(X) = \mathbb{1}_{\langle X, u \rangle \geq t}$$

Such a test is a simple linear criterion.
Decision trees
Error rate as a criterion

A natural criterion would be the error rate itself, since it is what matters eventually.

We set $C = 4$ and let $[n_1, n_2, n_3, n_4]$ denote a set with the corresponding number of samples in each class.

Consider the two following splits of a set with 100 samples of each classes. The error rate is the best we can obtain with a constant prediction on each subset.

<table>
<thead>
<tr>
<th>$T_{q=0}$</th>
<th>$T_{q=1}$</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>[100, 99, 1, 0]</td>
<td>[0, 1, 99, 100]</td>
<td>0.5</td>
</tr>
<tr>
<td>[100, 50, 50, 0]</td>
<td>[0, 50, 50, 100]</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Hence, the error rate is a bad criterion.

Decision trees
Motivation

Our goal is to built an efficient tree: we want to build a tree so that the average number of questions asked is as low as possible.

Beside computational issues, this is motivated by the Occam’s razor principle: A simpler predicting rule generalizes better.

A very good estimate of the efficiency of a “question” is Shannon’s entropy.
Given a random variable $Y$ on $\{1, \ldots, C\}$, its entropy is

$$H(Y) = - \sum_{y=1}^{C} P(Y = y) \log_2 P(Y = y)$$

1. The base of the log is arbitrary. Base 2 relates entropy clearly to binary encoding,
2. by convention $0 \log 0 = 0$.

The entropy of a deterministic variable is 0, and the entropy of a variable uniform on $C$ values is $\log_2(C)$. 

Decision trees
Entropy, various distributions
Given two random variables $X$ and $Y$ we define the conditional entropy as

$$H(Y \mid X) = \sum_x H(Y \mid X = x)P(X = x)$$

$$= -\sum_x \left\{ \sum_y P(Y = y \mid X = x) \log_2 P(Y = y \mid X = x) \right\} P(X = x)$$

1. It is the minimum number of bits to send on average to describe $Y$ when $X$ is already known.
2. If $Y$ is a deterministic function of $X$, $H(Y \mid X = x)$ is always 0.
3. If $X$ and $Y$ are independant, then $\forall x$, $H(Y \mid X = x) = H(Y)$, and $H(Y \mid X) = H(Y)$.

Decision trees

Chain rule

We have

$$H(X, Y) = H(X \mid Y) + H(Y)$$

which is consistent with Shanon’s theorem: to describe $X$ and $Y$, first send enough bits for $Y$, and then bits needed to describe $X$ given $Y$ is known.

And with a convexity argument, we can show that

$$H(X \mid Y) \leq H(X).$$

which makes sense: Either $Y$ helps you to know $X$, in which case you need less bits to describe $X$ when $Y$ is known, or it is useless and you need as many.
Considering the properties of the entropy, it is legitimate to select at every node the test $q$ maximizing the uncertainty drop quantified by the empirical Shannon entropy reduction:

$$S(q) = \hat{H}(Y) - \hat{H}(Y \mid q(X))$$

Finally, given $\mathcal{B}$, the learning algorithm can be summarized as:

1. If all $Y_t$ are equal, the tree is a leaf with the common value of the $Y_t$ as label,
2. else, the tree is an internal node:
   2.1 pick the $q$ in $\mathcal{B}$ minimizing $\hat{H}(Y \mid q(X))$,
   2.2 split the training set in two sub-set $\mathcal{T}_{q=0}$ and $\mathcal{T}_{q=1}$,
   2.3 build $\mathcal{T}_0$ from $\mathcal{T}_{q=0}$ and $\mathcal{T}_1$ from $\mathcal{T}_{q=1}$.

The depth or minimum number of samples can be controlled to limit over-fitting.
If we come back to the example with the error rate as a split criterion, we have:

<table>
<thead>
<tr>
<th>$T_{q=0}$</th>
<th>$T_{q=1}$</th>
<th>Error</th>
<th>$\hat{H}(Y \mid q(X))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>[100, 99, 1, 0]</td>
<td>[0, 1, 99, 100]</td>
<td>0.5</td>
<td>1.0707</td>
</tr>
<tr>
<td>[100, 50, 50, 0]</td>
<td>[0, 50, 50, 100]</td>
<td>0.5</td>
<td>1.5</td>
</tr>
</tbody>
</table>

Hence, using the entropy leads to better results.

**Gini’s Impurity as a criterion**

Another classical criterion is based on Gini’s impurity:

$$l_g(T) = \sum_{i,j} \hat{P}_T(Y = i) \hat{P}_T(Y = j)$$

The score for a binary test is

$$\Delta l_g = l_g(T) - \frac{l_g(T_{q=0})\|T_{q=0}\| + l_g(T_{q=1})\|T_{q=1}\|}{\|T\|}$$

Results are similar to those obtained with entropy. What matters is convexity.
We consider the case $D = 2$ and $T = 1,000$, with the following training set.
Decision trees
Arbitrary splits (randomized)

Depth = 11

Decision trees
Toy problem

We consider the same situation with noisy training labels.
Decision trees
Orthogonal splits ($\approx$ ID3), noisy labels

Depth = 10

Decision trees
Arbitrary splits, noisy labels (randomized)

Depth = 11
We need a way to control the overfitting.

A very simple strategy consists of limiting the depth of the tree, hence adding a penalty term which is 0 when the depth of the tree is below a threshold and $+\infty$ otherwise.

Another criterion is to put a lower bound on the number of samples required to split a node during training.

More complex rules exist and can be optimized by pruning the trees (cf. C4.5).
Another very powerful strategy consists of building several trees with different random sequences, and using a voting scheme.

Two different sources of randomization can be used

- Randomize the data (Bootstrap Aggregating aka bagging)
  

- Randomize the classifiers
  

The idea of bagging is general and may be applied to any predictor.

It consists of sampling from the original sample set as if it was the true distribution, to produce as many “artificial” training sets as needed.

From these multiple training sets, one can compute estimates of quantities such as the variance of the prediction.
More precisely, given a training set

$$(X_t, Y_t), \ t = 1, \ldots, T$$

for any $r = 1, \ldots, R$, we can generate

$$N^1_1 \ \ldots \ \ N^1_T,$$

$$\ldots$$

$$N^R_1 \ \ldots \ \ N^R_T.$$

i.i.d uniform on $\{1, \ldots, T\}$, and from each

$$(X_{N^r_t}, Y_{N^r_t}), \ t = 1, \ldots, T$$

we can train a classifier $f^r$.

Bagging

Example with polynomial regression

Expected prediction and variance estimated with bootstrapping.
Bagging
Example with polynomial regression

Expected prediction and variance estimated with multiple training sets.

Decision trees
Voting

Randomizing the trees is done by using different random seeds when picking the best splits. It is often better since all the training set is used.

Smooth voting

Rejection consists of classifying as unknown a sample for which less than a certain proportion of the trees agree (here 90%, shown in blue).

Rejection

(this slide contains videos)
Decision trees
Voting, noisy labels

Same thing with noisy labels.

Smooth voting, rejection
(this slide contains videos)

Decision trees
Controlling the depth + vote

Voting + controlled depth.

Smooth voting, rejection
(this slide contains videos)
The exact same tree-building strategy can be used for regression, simply by using the variance in place of Shannon’s Entropy.

The resulting algorithm aims at creating nodes with constant target values over the training set.

As for classification, randomization improves performance, in that case by reducing the variance without increasing the bias much.

**Decision trees**

**Extremely randomized trees**

A variant of decision trees called *Extremely randomized trees* relies on a highly randomized selection of the test to put at each node.

In practice, at every node, the selection of the Boolean test is done as follow, given a parameter $K$:

- Select $K$ attributes $a_k \in \{1, \ldots, D\}$, $k = 1, \ldots, K$
- For each, pick a threshold $s_k \sim U ([\min_t X_{t}^{a_k}, \max_t X_{t}^{a_k}])$
- Select the pair $(a_k, s_k)$ maximizing your objective score (entropy or variance).

Increasing $K$ reduces the bias but increases the variance.

So, why do we like decision trees?

- They are very simple conceptually and algorithmically.
- They allow to deal with any type of signal (categorical or continuous) as long as we can test Boolean properties.
- They are computationally extremely efficient.
- Combined with ensemble methods they have excellent performance.
- They are used: Google, Yandex, Microsoft (Kinect !)
- They can be analyzed after learning (to some extent).
A very simple and powerful algorithm for classification extracts many sub-windows of different sizes from the image, and classifies each of them individually.

Sub-windows are sampled, re-sized to $16 \times 16$ and converted to HSV, which is more robust to changes in illumination.


During training, each patch is labelled with its category, and trees are trained with the extremely randomized procedure.
During test, each patch votes for its category.

Performance on the Zubud data-set (201 buildings, 5 images of each) is 4.35% error, i.e. 5 misclassified images.
APPLICATION

POINT OF INTEREST RECOGNITION

Application
Keypoint Recognition

In many vision applications, one needs to recognize small patches of an image to match points specified on a reference image into a newly acquired image.

After learning the keypoint appearance on a reference model, the 3D pose can be estimated in real time.

Testing keypoints
(Özuysal & al. 2006)

(this slide contains videos)
Application

Handwritten digit recognition

The MNIST database is a free database of tens of thousands of handwritten characters.

Application
Handwritten digit recognition (cont.)

(Geman, Amit & Wilder, 1996)

(Geman, Amit & Wilder, 1996)
Application
Handwritten digit recognition (cont.)

(Geman, Amit & Wilder, 1996)
Training with portions of the NIST Special Database 3, which consists of \( \approx 223,000 \) binary images written by more than 2,000 writers.

Average depth is 8.8, maximum 20. Average number of leaf 600.

- With one tree, the classification rate is 93%
- With 25 trees, the classification rate is 99.2%
- With 25 trees and 1% rejection, classification rate is 99.5%
- With 25 trees and 3% rejection, classification rate is 99.8%
APPLICATION

HOUGH FORESTS FOR OBJECT DETECTION

Application
Object detection

Forest of decision trees have been used for object detection in natural scenes. The problem consists of automatically estimating bounding boxes around instances of the class of interest.

The Hough transform is a general technique to aggregate multiple partial evidences for the presence of a larger structure.

Each local evidence votes for the geometrical parameters consistent with it. For instance, to detect a circle of fixed radius, each black pixel votes for the “possible” centers in the image plan.

This scheme can be justified with a (simple) Bayesian formulation.

The same generalizes to multiple objects
And to more complex structures

The algorithm proposed by Gall & Lempitsky uses decision trees to predict, given a patch taken in an image, if it is in the bounding box of a pedestrian, and if it is the case, what is the distribution of the center’s location.

Training aims at minimizing at every node, either Shanon’s entropy for the variable “to be in the bounding box”, or the variance of the center’s location in the image plan.

(Gall & Lempitsky, 2009)
APPLICATION

REAL-TIME POSE ESTIMATION WITH CASCADED REGRESSION FORESTS

Application
Fine face pose estimation

The idea of cascaded pose regression proposed by Dollar


was used for precise face localization by Kazemi

The core idea is to have a sequence of (forests of) regression trees $r_1, \ldots, r_T$ and to feed each one with the current estimate

$$\hat{S}^{t+1} = \hat{S}^t + r_t(l, \hat{S}^t)$$

The image properties tested in the trees are *relative to the current pose estimate* (pose-indexed features). This allows the cascade to refine the accuracy of the estimation.

Real-time face pose estimation
The objective of the programming session is to play a bit with simple decision trees for classification with the MNIST dataset.


wget http://www.idiap.ch/~fleuret/files/EE613/EE613-pw10.tgz
  tar zxvf EE613-pw10.tgz
  cd EE613/pw10
wget http://www.idiap.ch/~fleuret/files/EE613/mnist.tgz
  tar zxvf mnist.tgz
  ./do.sh -v demo graph

After two minutes, you should see printed on the screen:

Depth 18 ... done (0.948333% / 14.49%).
Depth 19 ... done (0.561667% / 14.61%).

and the graph
class VignetteSet {
public:
    int nb_vignettes();
    int nb_classes();
    int width();
    int height();

    unsigned char pixel(int p, int x, int y);
    unsigned char label(int p);

    void load_mnist_format(char *picture_file_name,
                            char *label_file_name);
    void bootstrap(int nb, VignetteSet *vs);
};

class QuestionSet {
public:
    virtual int nb_questions() = 0;
    virtual bool response(int n_question,
                           VignetteSet *vs, int n_vignette) = 0;
};

class PixelQuestionSet : public QuestionSet {
public:
    PixelQuestionSet(int w, int h);
    virtual int nb_questions();
    virtual bool response(int n_question,
                           VignetteSet *vs, int n_vignette);
};
class Classifier {
public:
    virtual void train(VignetteSet *train_set) = 0;
    virtual int predict(VignetteSet *vs, int n_vignette) = 0;
    int nb_errors(VignetteSet *vs);
};

class DecisionTree : public Classifier {
public:
    DecisionTree(QuestionSet *question_set,
                 int depth_max,
                 int nb_min_samples_for_split,
                 int nb_questions_for_optimization);
    virtual ~DecisionTree();
    virtual void train(VignetteSet *train_set);
    virtual int predict(VignetteSet *vs, int n_vignette);
};

VignetteSet train_image_set, test_image_set;

train_image_set.load_mnist_format("train-images-idx3-ubyte",
                                 "train-labels-idx1-ubyte");

test_image_set.load_mnist_format("t10k-images-idx3-ubyte",
                                  "t10k-labels-idx1-ubyte");

PixelQuestionSet qs(train_image_set.width(),
                     train_image_set.height());

DecisionTree dt(&qs, 3, 10, 100);

test_error = scalar_t(dt.nb_errors(test_image_set)) /
              scalar_t(test_image_set->nb_vignettes());
It is suggested to implement each question in the corresponding function

```cpp
void computation_question1(VignetteSet *train_image_set,
                           VignetteSet *test_image_set) {
    ...
}
```

and to test it by calling the `./do.sh question1`, `./do.sh question2`, etc.

Note that if you compile the code with `make -k DEBUG=yes` the `VignetteSet::pixel()` method will check for out-of-bound errors.