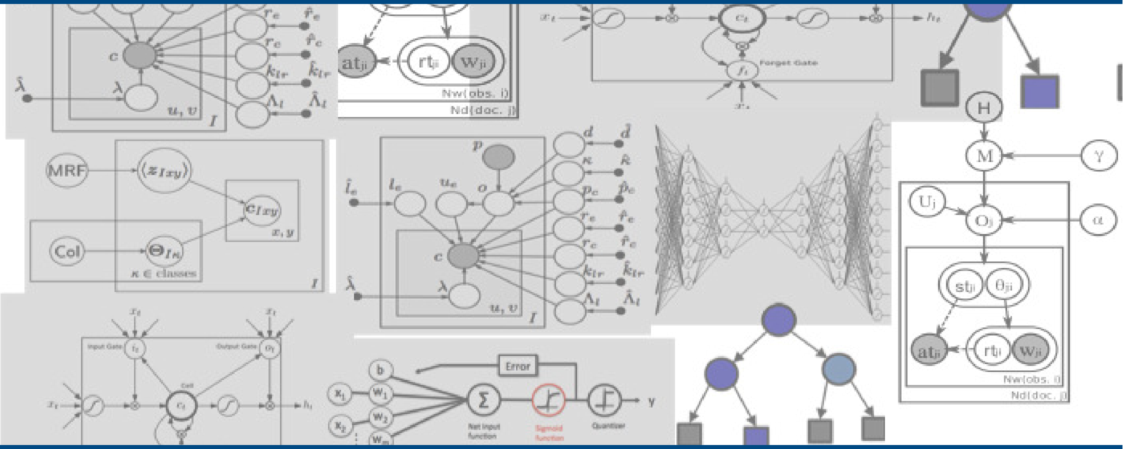
EE613 Machine Learning for Engineers

Decision Trees

Dr. Jean-Marc Odobez

Fall 2019



Outline

- Decision Trees
- Bagging and Random Forests
- Applications

Outline

Decision Trees

Bagging and Random Forests

Applications

Game of twenty questions

Decision trees in machine learnuing are related to the game of twenty questions, where one tries to guess an object by asking less than 20 questions.

Q: Is it a human?

A: Yes

Q: Is she/he alive?

A: Yes

Q: Is it a man?

A: No

Q: Is she american?

A: ...

Game of twenty questions

Important points:

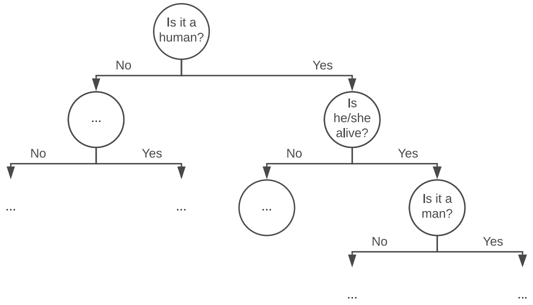
1: The full strategy for asking question can be defined before playing, from the statistics we have about the most likely choices of objects and people

Game of twenty questions

Important points:

1: The full strategy for asking question can be defined before playing, from the statistics we have about the most likely choices of objects and people

2: The strategy can be represented by a tree:



Decision trees, in short

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

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

✎ L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone. *Classification and regression trees.* Wadsworth, 1984

Decision trees, an example

Dataset

Let’s assume that we have a two class classification problem: classify a given fruit as an Apple (class c1) or a Mango (class c2).

To learn the classifier, we are given some training data:

Decision trees, an example

Objective

The objective is to learn a mapping F from an input X (an image of a fruit for instance) to the set of class $\mathcal{C} = \{c1, c2\}$.

$$F(X) = Y \in \mathcal{C}$$

This can be seen as finding a decision boundary on the representation space of data.

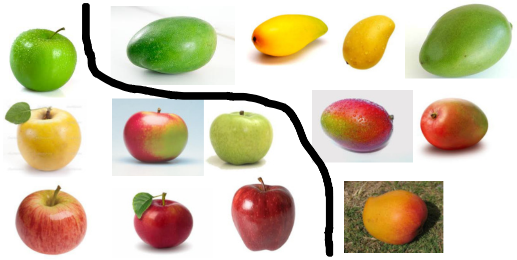
Decision trees, an example

Objective

The objective is to learn a mapping F from an input X (an image of a fruit for instance) to the set of class $\mathcal{C} = \{c1, c2\}$.

$$F(X) = Y \in \mathcal{C}$$

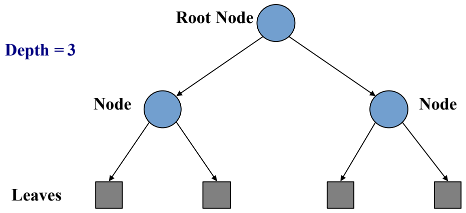
This can be seen as finding a decision boundary on the representation space of data.

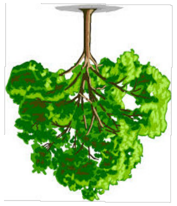


Decision trees, an example

Tree structure

Reminder: tree terminology

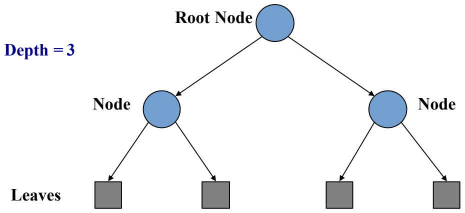


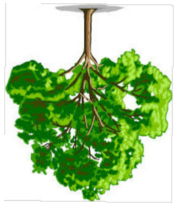


Decision trees, an example

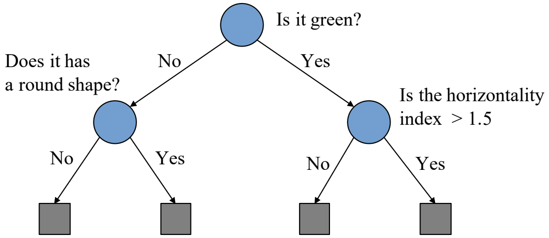
Tree structure

Reminder: tree terminology





Decision tree: includes a binary decision test at each node.

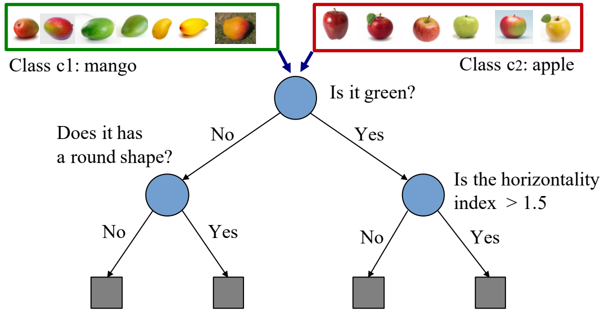


Decision trees, an example

Learning

How to learn the decision tree, given the training data? for the example, assume the decisions at each node are given. We will discuss how to learn this later.

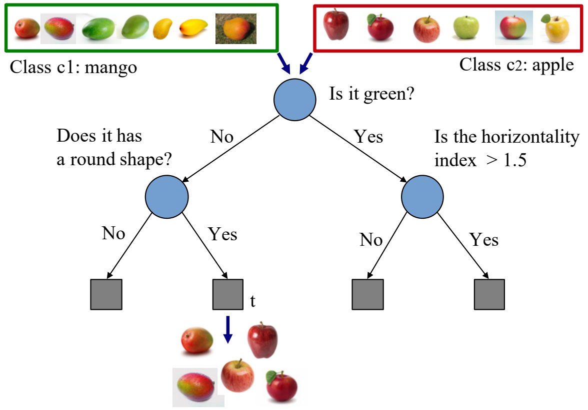
The training data goes through the tree



Decision trees, an example

Learning

When traversing the tree, a set of samples reach the same leave in the tree



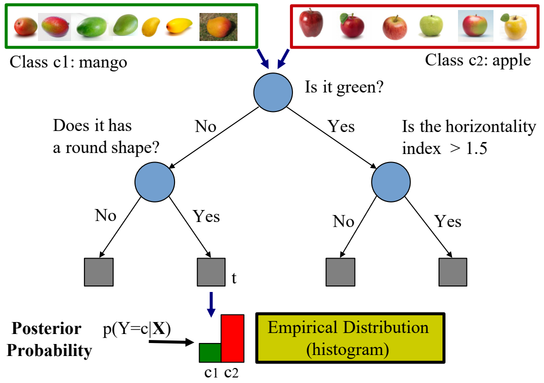
Decision trees, an example

Learning

From the set of samples in a given leave t, we can build its posterior probability $p_t(Y = c|X)$ from the empirical distribution (histograms):

$$p_t(Y = c|X) = \frac{N_c^t}{N^t}$$

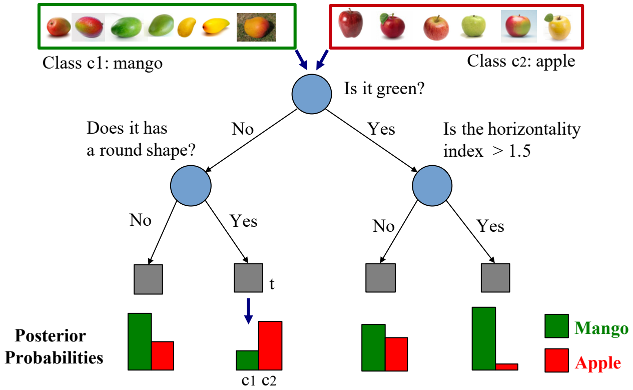
where N_c^t is the number of training sample of class c which fall in the leave t, and N^t is the number of training samples falling in leave t.



Decision trees, an example

Learning

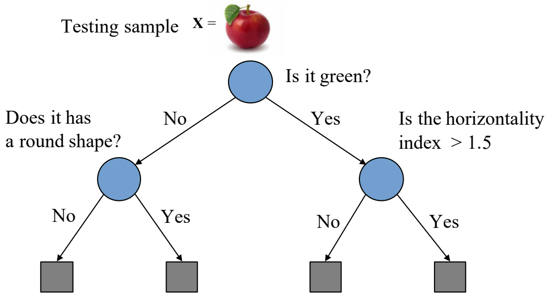
Such posterior probabilities can be computed for each leave.



Decision trees, an example

Testing (tree evaluation)

At test time, to compute the prediction $F(X)$ of a sample, we traverse the tree using the binary decisions made according to the sample.



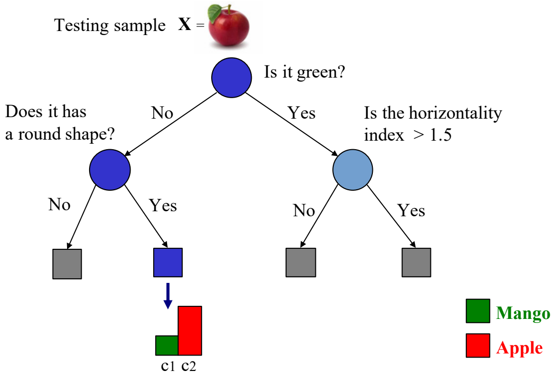
Decision trees, an example

Testing (or tree evaluation)

The sample reaches a leave, where the prediction can be made according to the maximum à posteriori (MAP):

$$\hat{Y} = \underset{c}{\operatorname{argmax}}\, p_t(Y = c|X)$$

In the current case the prediction is $c2$, so an apple.



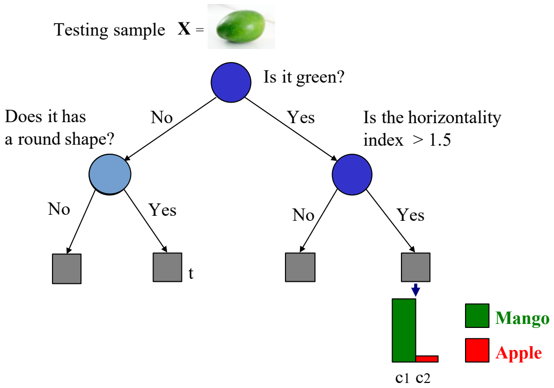
Decision trees, an example

Testing (or tree evaluation)

The sample reaches a leave, where the prediction can be made according to the maximum à posteriori (MAP):

$$\hat{Y} = \underset{c}{\operatorname{argmax}}\, p_t(Y = c|X)$$

In this other exemple, the prediction is $c1$, so a mango.



Decision trees

Benefits

Despite being an old approach, (random forests of) decision trees work really well. They can handle both relatively small and large amounts of data.

✎ Manuel Fernández-Delgado, Eva Cernadas, Senén Barro, and Dinani Amorim. Do we need hundreds of classifiers to solve real world classification problems? *Journal of Machine Learning Research*, 15:3133–3181, 2014

"We evaluate 179 classifiers arising from 17 families (discriminant analysis, Bayesian, neural networks, support vector machines, decision trees, rule-base classifiers, boosting,...)

The random forest is clearly the best family of classifiers (3 out of 5 best classifiers are RF),..."

Decision trees

Notations

Decision trees can be applied to:

- classification, using continous, discrete, or categorical data
- regression
- density estimation

For the rest of the course, we will focus on classification from continuous features. The extension to other cases is relatively natural.

Decision trees

Notations

Decision trees can be applied to:

- classification, using continous, discrete, or categorical data
- regression
- density estimation

For the rest of the course, we will focus on classification from continuous features. The extension to other cases is relatively natural.

Let $X = (X^1, \dots, X^{N_D})$ be a random variable in $\mathbf{R}^{N_D}$ representing the vector of measurements, and Y a random variable in $\mathcal{C} = (1, \dots, N_C)$ the unknown class to predict.

Decision trees

Notations

Decision trees can be applied to:

- classification, using continous, discrete, or categorical data
- regression
- density estimation

For the rest of the course, we will focus on classification from continuous features. The extension to other cases is relatively natural.

Let $X = (X^1, \dots, X^{N_D})$ be a random variable in R^{N_D} representing the vector of measurements, and Y a random variable in $\mathcal{C} = (1, \dots, N_C)$ the unknown class to predict.

Let $(X_i, Y_i)_{i=1 \dots N_s}$ be the training set.

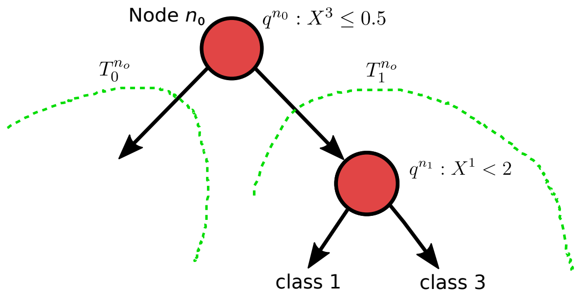
The purpose of the lecture is to predict Y from X with decision trees built from the training set.

Decision tree

Definition

We can define a decision tree comprising a set of nodes $\{n\}$ in a recursive fashion.

1. if the node n is a leaf, it carries a label from $\mathcal{C}$
2. if the node n is internal, it carries a Boolean test $q^n : \mathsf{R}^{N_D} \longrightarrow \{0, 1\}$ and it has two subtrees T_0^n and T_1^n of finite depth



Decision tree

prediction (evaluation)

The prediction $F(X)$ of a tree T given a vector of measurements X is given by:

1. if T is a leaf node, the prediction is the label it holds.
2. if T is a tree with a root node n, its prediction is the prediction of its subtree $T^n_{q(X)}$.

Decision trees - Learning

Given the training set, learning an efficient tree minimizing some classification performance measure is NP hard.

Decision trees - Learning

Given the training set, learning an efficient tree minimizing some classification performance measure is NP hard.

In practice, the tree is built in a greedy and recursive fashion, starting from the root.

At each node n, the training of the subtree with root node n relies on a training set D^n of (X_i, Y_i) samples.

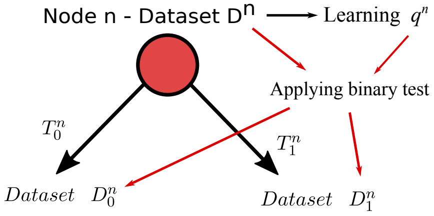
Decision trees - Learning

Denoting by $\mathcal{B}$ the set of binary tests we can make at each node, the decision tree is built recursively from node n as follows:

Decision trees - Learning

Denoting by $\mathcal{B}$ the set of binary tests we can make at each node, the decision tree is built recursively from node n as follows:

1. if all Y_i from the set D^n are the same, then n is a leaf, and the common value of Y_i is its label;
2. otherwise: select the binary test q^n optimizing some criterion based on D^n, apply the selected binary test on D^n, which result on two subsets D_0^n and D_1^n which can be used to build the decision subtrees T_0^n and T_1^n.

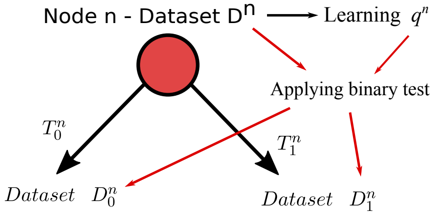


Decision trees - Learning

22/72

Denoting by $\mathcal{B}$ the set of binary tests we can make at each node, the decision tree is built recursively from node n as follows:

1. if all Y_i from the set D^n are the same, then n is a leaf, and the common value of Y_i is its label;
2. otherwise: select the binary test q^n optimizing some criterion based on D^n, apply the selected binary test on D^n, which result on two subsets D_0^n and D_1^n which can be used to build the decision subtrees T_0^n and T_1^n.



The two (related) remaining issues for learning the tree are:

1. the choice of the set $\mathcal{B}$ of binary tests to consider (at each node)
2. how to select (or learn) an optimal (or at least good) q given some training data.

Decision trees - Types of binary tests

Examples of $\mathcal{B}$:

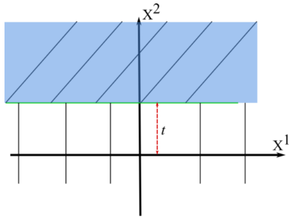
1. Stumps: parameterized by $p = (d, t)$, with $d \in \{1, \ldots, N_D\}$ is the data component to take into account, and a threshold $t \in \mathbb{R}$. It is defined as

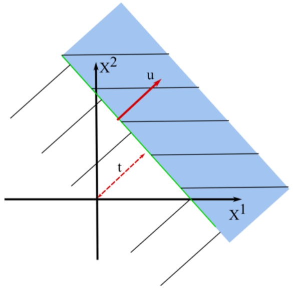
$$q(X) = \mathbb{1}_{\{X^d > t\}}$$

2. Linear classifiers: parameterized by $p = (u, t)$, where $u \in \mathbb{R}^{N_D}$ and threshold t:

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

Such a test is a generalization of the previous case.



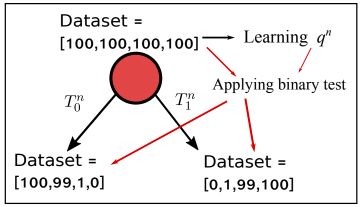


How to select a good binary test q?

A natural choice would be to use the classification error rate, since it is what matters eventually.

Let us look at the following example. We set $N_C = 4$, and denote by $[n_1, n_2, n_3, n_4]$ a set with the corresponding number of samples in each class. Consider now the two following cases, where the learning of a binary test from a set with 100 samples in each class results in the corresponding splits

CASE A:



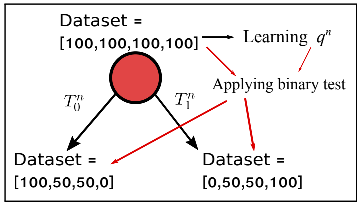
Dataset = [100,100,100,100] → Learning q^n

Applying binary test

T_0^n Dataset = [100,99,1,0]

T_1^n Dataset = [0,1,99,100]

CASE B:



Dataset = [100,100,100,100] → Learning q^n

Applying binary test

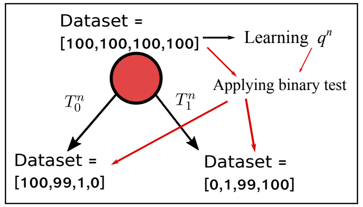
T_0^n Dataset = [100,50,50,0]

T_1^n Dataset = [0,50,50,100]

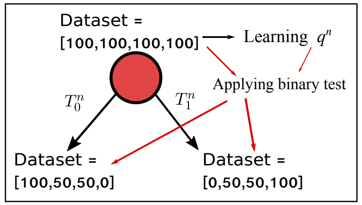
Is there a split which sounds more preferable?

How to select a good binary test q?

CASE A:



CASE B:



Is there a split which sounds more preferable?

We consider that at a given node, the best prediction is the one that minimizes the error rate when making a constant prediction.

- accordingly, what is the prediction at each node?
- what are the amount of errors before the split? what is the error rate?
- similarly, for each of the cases, what is the error rate after the split?
- do you think that the error rate is a good criterion for selecting a split? Why?

How to select a good binary test q?

Considerations:

Our goal is to build an efficient tree: we want to build it 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 (here, less questions) generalizes better.

A very good estimate of the efficiency of a "question" (binary test) is based on (a gain in) Shannon's entropy of the distribution of labels in the sample set.

Shannon entropy - reminder

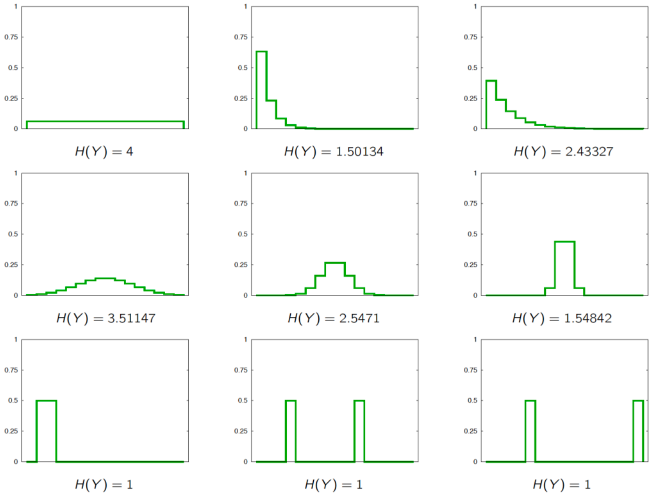
Given a random variable Y on $\{1, \dots, N_C\}$, its entropy is

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

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

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

Shannon entropy of various distributions (on 16 classes)



$H(Y) = 4$ $H(Y) = 1.50134$ $H(Y) = 2.43327$

$H(Y) = 3.51147$ $H(Y) = 2.5471$ $H(Y) = 1.54842$

$H(Y) = 1$ $H(Y) = 1$ $H(Y) = 1$

High entropy: flat histogram, sampled values are less predictable
Low entropy: varied distribution, with peaks and valleys (histogram has many high and lows), values sampled from it are more predictable

Conditional entropy

We want to compare the entropy of the class distribution before and after the split. To do this, we need to introduce the conditional entropy.

Given two random variables X and Y, we define the conditional entropy of Y given X as:

$$\begin{aligned} H(Y|X) &= \sum_x H(Y|X=x)P(X=x) \\ &= -\sum_x \left\{ \sum_y P(Y=y|X=x)\log_2 P(Y=y|X=x) \right\} P(X=x) \end{aligned}$$

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, $H(Y|X=x)$ is always 0.
3. if Y and X are independant, then $\forall x, H(Y|X=x) = H(Y)$, and $H(Y|X) = H(Y)$.

Entropy - chain rule

We have

$$H(Y,X) = H(Y|X) + H(X)$$

which is consistent with Shannon's theorem: to describe Y and X, first send enough bits for X, and then the bits needed to describe Y given X is known.

And with a convexity argument, we can show that

$$H(Y|X) \leq H(Y)$$

which makes sense:

- either X helps you to know Y, in which case you need less bits to describe Y when X is known;
- or it is useless, and you need as many.

Binary test selection

Given the properties of the entropy, it is legitimate to select at every node the binary test q maximizing the uncertainty drop quantified by the empirical Shannon entropy reduction.

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

This measures the entropy of the labels before the split (first term), minus the entropy of the label after the split.

Decision trees - learning algorithm refinement

The learning algorithm to build the decision tree recursively from node n can be defined as:

1. Define the set $\mathcal{B}^n$ of binary tests to consider at node n.
2. Then:
   - 2.1 if all Y_i from the set D^n are the same, then n is a leaf, and the common value of Y_i is its label;
   - 2.2 otherwise (internal node):
     - 2.2.1 select the $q^n \in \mathcal{B}^n$ minimizing $\hat{H}(Y|q(X))$ on D^n;
     - 2.2.2 split the training set into two subsets $D_{\mathbf{0}}^n$ and $D_{\mathbf{1}}^n$;
     - 2.2.3 build the left subtree $T_{\mathbf{0}}^n$ from $D_{\mathbf{0}}^n$ and $T_{\mathbf{1}}^n$ from $D_{\mathbf{1}}^n$.

Decision trees - learning algorithm refinement

The learning algorithm to build the decision tree recursively from node n can be defined as:

1. Define the set $\mathcal{B}^n$ of binary tests to consider at node n.
2. Then:
   - 2.1 if all Y_i from the set D^n are the same, then n is a leaf, and the common value of Y_i is its label;
   - 2.2 otherwise (internal node):
     - 2.2.1 select the $q^n \in \mathcal{B}^n$ minimizing $\hat{H}(Y|q(X))$ on D^n;
     - 2.2.2 split the training set into two subsets $D^n_{\mathbf{0}}$ and $D^n_{\mathbf{1}}$;
     - 2.2.3 build the left subtree $T^n_{\mathbf{0}}$ from $D^n_{\mathbf{0}}$ and $T^n_{\mathbf{1}}$ from $D^n_{\mathbf{1}}$.

Note: modifications of the algorithm are needed to limit over-fitting by controlling the depth and/or the minimum of samples per node.

Binary test comparison

If we come back to the example taken earlier (with $D = [100, 100, 100, 100]$), we have:

D_0	D_1	Classification error	$\hat{H}(Y\|q(X))$
[100,99,1,0]	[0,1,99,100]	0.5	1.0707
[100,50,50,0]	[0,50,50,100]	0.5	1.5000

Hence, using the reduction in entropy leads to better results.

Gini's impurity as a binary test criterion

Another classical criterion is based on Gini's impurity. It measures the probability that an element taken randomly would be misclassified if its label would be selected according to the distribution of labels in the set D.

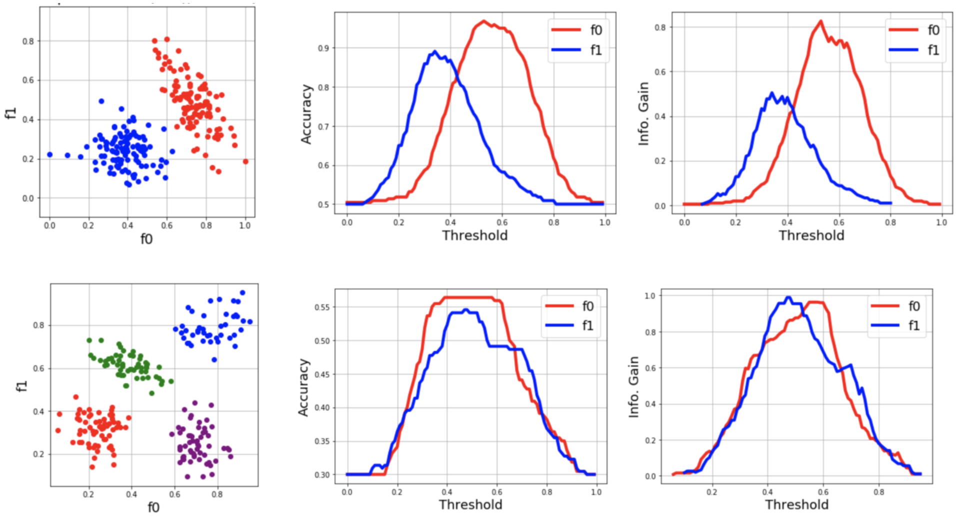
$$I_g(D) = \sum_y \hat{P}(Y=y)\hat{P}(Y \neq y) = \sum_y \hat{P}(Y=y)\left(\sum_{k\neq y}\hat{P}(Y=k)\right)$$
$$= \sum_y \hat{P}(Y=y)(1-\hat{P}(Y=y))$$

The minimum (0) is achieved when all samples belon to a single category.

The score for a binary test can be measured accordingly

$$\Delta I_g = I_g(D) - \frac{I_g(D_0)\times\|D_0\| + I_g(D_1)\times\|D_1\|}{\|D_0\|+\|D_1\|}$$

Decision tree - splitting examples



For the two class example, accuracy and information gain (entropy) are rather aligned in terms of features to select and thresholds. This is not the case for 4 class example, see for example the 'plateau' between 0.4 and 0.6 for feature f_0 in the accuracy case compared to the f_0 curve in the entropy case.

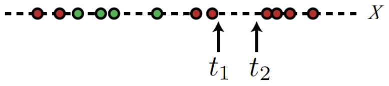
Decision trees - Note: set of possible thresholds

- Binary tree, split on attribute X^k - Stump classifier
  - $X^k < t$, one branch
  - $X^k \geq t$, other branch
- Search best value of t
  - with entropy: no closed form solution
  - test all possible values of t: seems hard!

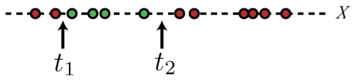
Decision trees - Note: set of possible thresholds

- Binary tree, split on attribute X^k - Stump classifier
  - $X^k < t$, one branch
  - $X^k \geq t$, other branch
- Search best value of t
  - with entropy: no closed form solution
  - test all possible values of t: seems hard!

- But only a finite number of $t's$ are important



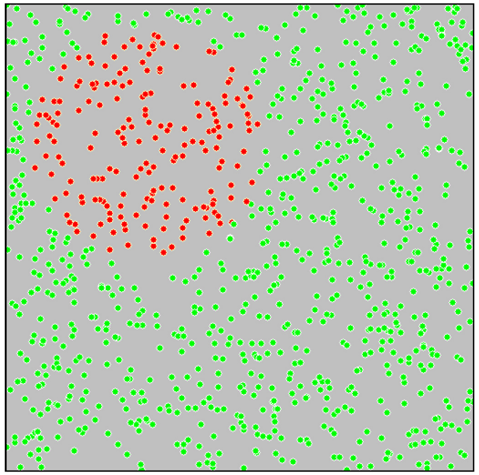
  - sort data according to X^k into $\{X_1^k, \ldots, X_{N_s}^k\}$
  - consider split point of the form $X_i^k + (X_{i+1}^k - X_i^k)/2$
  - moreover, only splits between examples of different classes matter



This pre-sorting can be done on small datasets (cf Scikit-Learn), but is computationally heavy. An alternative is to select a (fixed) random set of thresholds (cf later).

Learning on a toy example

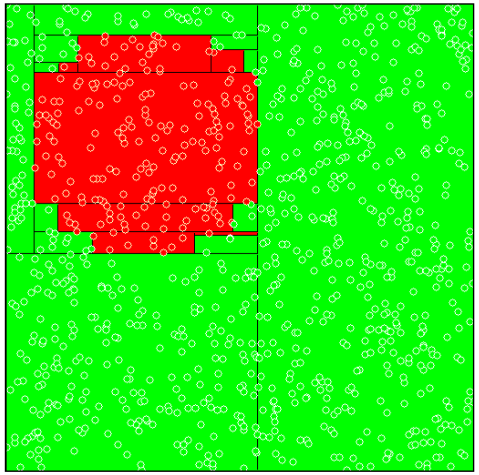
data



We consider the training set above, with $N_D = 2, N_s = 1000$.

Learning on a toy example

decision with stumps

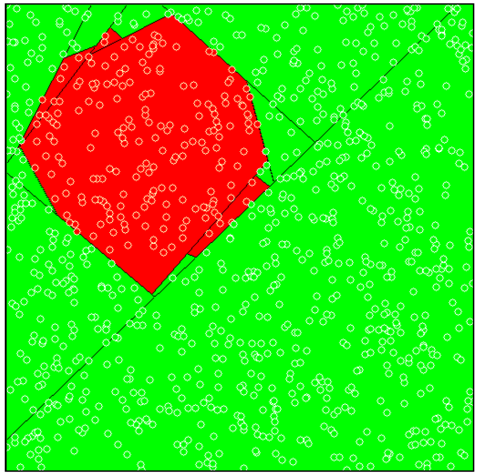


Tree depth = 10

Using the stump binary classifier (orthogonal splits, more or less what is used in the ID3 algorithm, which also handles label features).

Learning on a toy example

decision with arbitrary splits (randomized)

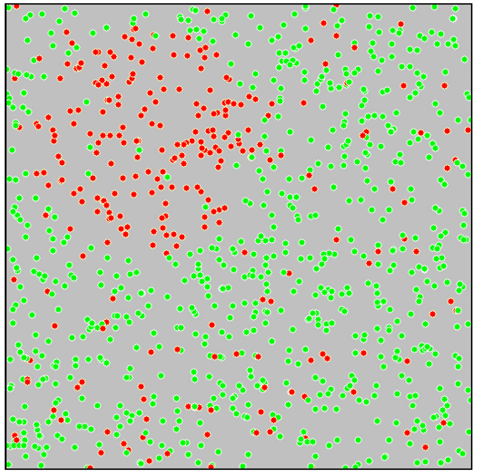


Tree depth = 11

Using the linear splits

Learning on a toy example, adding noise

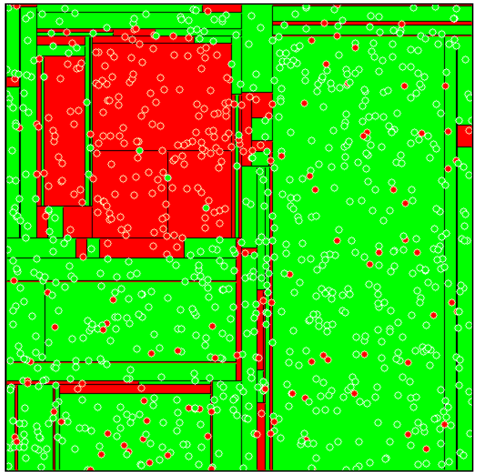
data



We consider the same situation, but now with noisy training labels.

Learning on a toy example, adding noise

decision with stumps

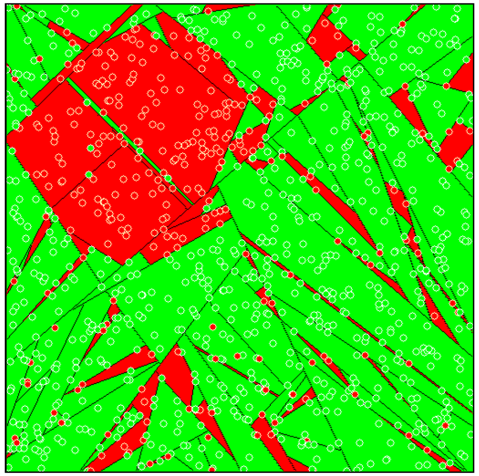


Tree depth = 10

Using the stump binary classifier (orthogonal splits, more or less what is used in the ID3 algorithm, which also handles label features).

Learning on a toy example, adding noise

decision with arbitrary splits (randomized)



Tree depth = 11

Using the linear splits.

Controlling over-fitting

The behavior observed in the data with noise is due to over-fitting: without constraints, during training, the tree can be grown until all data points within a leave are from a single category (i.e. it can always reach a training set error of 0!), since nodes can be split until there are leaves containing single data points.

Controlling over-fitting

The behavior observed in the data with noise is due to over-fitting: without constraints, during training, the tree can be grown until all data points within a leave are from a single category (i.e. it can always reach a training set error of 0!), since nodes can be split until there are leaves containing single data points.

We thus need a way to control this over-fitting, and introduce a bias towards simpler trees.

Controlling over-fitting

The behavior observed in the data with noise is due to over-fitting: without constraints, during training, the tree can be grown until all data points within a leave are from a single category (i.e. it can always reach a training set error of 0!), since nodes can be split until there are leaves containing single data points.

We thus need a way to control this over-fitting, and introduce a bias towards simpler trees.

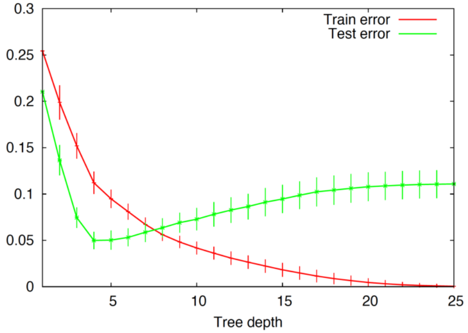
Several strategies exists, like

- limiting the depth of the tree.
- setting a lower bound on the number of samples required to split a node during training. The idea is that when the number of samples is low, it is difficult to compute useful statistics, and also that the growth of the tree will adapt to the amount of data.
- ensemble methods (using sets of classifiers, cf Random Forests)
- more complex rules exist and can be optimized by pruning the trees (cf C4.5 algorithm).

Controlling over-fitting

Limiting the depth

A very simple strategy consists of limiting the depth of the tree, adding a penalty term which is 0 when the depth of the tree is below a threshold, and ∞ otherwise. The threshold to be used can be set by relying on a validation set and monitoring the error on this validation set.



Example of accuracy vs depth training curve on the training and test (or validation) sets.

Outline

Decision Trees

Bagging and Random Forests

Applications

Controlling over-fitting : ensemble methods

Main idea: variance reduction

Another very powerful strategy consists of relying on ensemble methods, which is a class of methods which make predictions by combining the outputs of a set of individual models.

As a reminder, averaging reduces variance. If we have a set of N_B estimators F_b, then

$$F(X) = \frac{1}{N_B}\sum_{b=1}^{N_B} F_b(X), \text{ then } Var\left(F(X)\right) = \frac{Var(F_b(X))}{N_B}$$

assuming that the predictions are independent and are build in the same way (hence having the same statistical variances).

How to do so with decision trees? How to build different predictors?

Controlling over-fitting : ensemble methods

Randomization

The goal is to build several trees with different random sequences, and using a voting scheme to combine their decisions.

Two different sources of randomization can be used:

- randomize the data (Bootstrap Aggregating aka bagging).

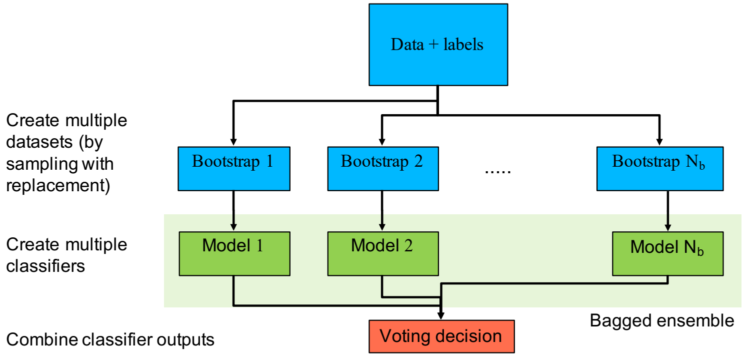
✎ Leo Breiman. Bagging predictors.
Machine Learning, 24(2):123–140, Aug 1996

- randomize the classifiers.

✎ Yali Amit, Donald Geman, and Kenneth Wilder. Joint induction of shape features and tree classifiers.
Pattern Analysis and Machine Intelligence, IEEE Transactions on, 19:1300 – 1305, 12 1997

Bagging

The idea of bagging is general and can be applied to any predictor (not only trees).



It consists of sampling from the original training dataset as if this was the true distribution, to produce as many 'artificial' training sets as needed (this is known as bootstrapping).

From these multiple training sets, one can build multiple classifiers and compute aggregated results out of them, including uncertainty measures about the predictions (like the variance).

Bagging: creating multiple datasets

Sampling with replacement

To produce a set of new training sets, we sample the original sample set. That is, given the initial training set:

$$D = \{(X_i, Y_i)_{i=1\dots N_s}\}$$

for $b \in \{1, \dots, N_B\}$, we can sample N_s' random numbers i_j i.i.d. uniform on $\{1, \dots, N_s\}$ and build the b^{th} dataset:

$$D_b = \{(X_{i_j}, Y_{i_j})_{j=1\dots N_s'}\}$$

from which we can train the classifier F_b.

Bagging: creating multiple datasets

Sampling with replacement

To produce a set of new training sets, we sample the original sample set. That is, given the initial training set:

$$D = \{(X_i, Y_i)_{i=1\dots N_s}\}$$

for $b \in \{1, \dots, N_B\}$, we can sample N_s' random numbers i_j i.i.d. uniform on $\{1, \dots, N_s\}$ and build the b^{th} dataset:

$$D_b = \{(X_{i_j}, Y_{i_j})_{j=1\dots N_s'}\}$$

from which we can train the classifier F_b.

This corresponds to sampling with replacement, i.e. the same datapoints of D can appear multiple times in D_b.

Bagging: creating multiple datasets

Sampling with replacement

To produce a set of new training sets, we sample the original sample set. That is, given the initial training set:

$$D = \{(X_i, Y_i)_{i=1\ldots N_s}\}$$

for $b \in \{1, \ldots, N_B\}$, we can sample N_s' random numbers i_j i.i.d. uniform on $\{1, \ldots, N_s\}$ and build the b^{th} dataset:

$$D_b = \{(X_{i_j}, Y_{i_j})_{j=1\ldots N_s'}\}$$

from which we can train the classifier F_b.

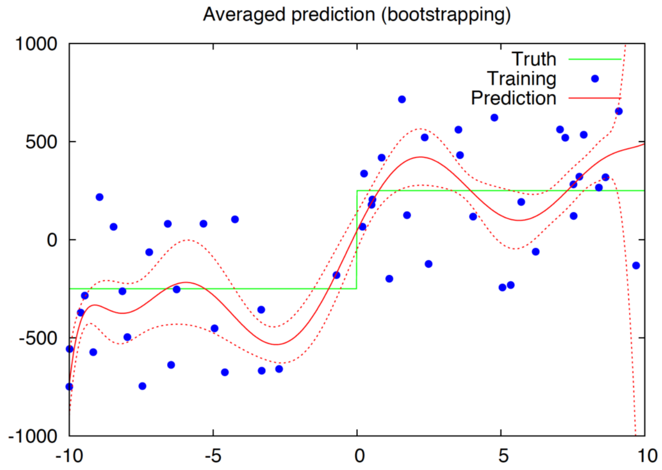
This corresponds to sampling with replacement, i.e. the same datapoints of D can appear multiple times in D_b.

When $N_s' = N_s$, the resampled datasets contain on average around 63% of the samples of the original dataset (the other 37% are duplicates).

Bagging

Example with polynomial regression

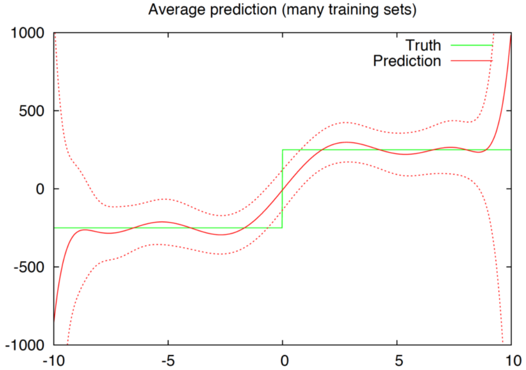
Expected prediction and variance estimated by applying bagging, i.e. when using multiple training sets bootstrapped from a single noisy training dataset (blue dots).



Bagging

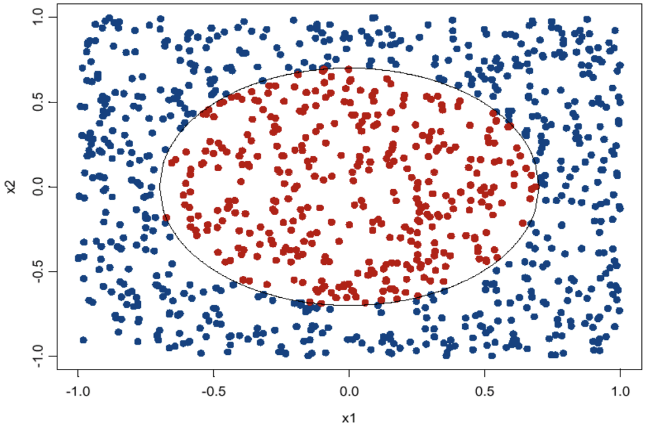
Example with polynomial regression

Expected prediction and variance estimated from the prediction obtained using multiple (independently drawn) datasets (cf blue datapoints in previous plot)



Bagging

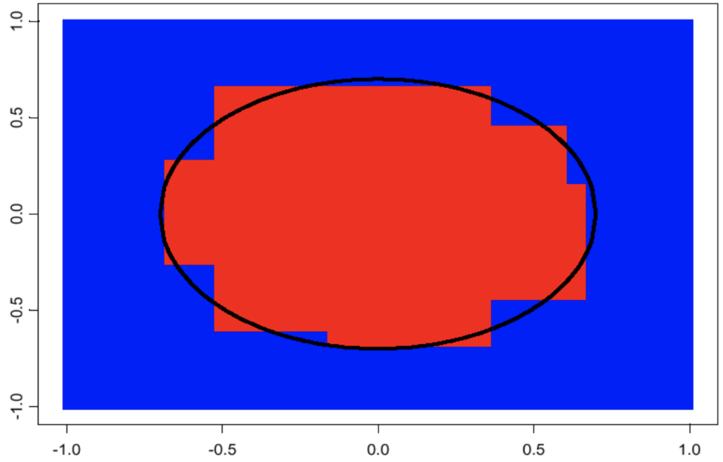
Example for classification - data



Bagging

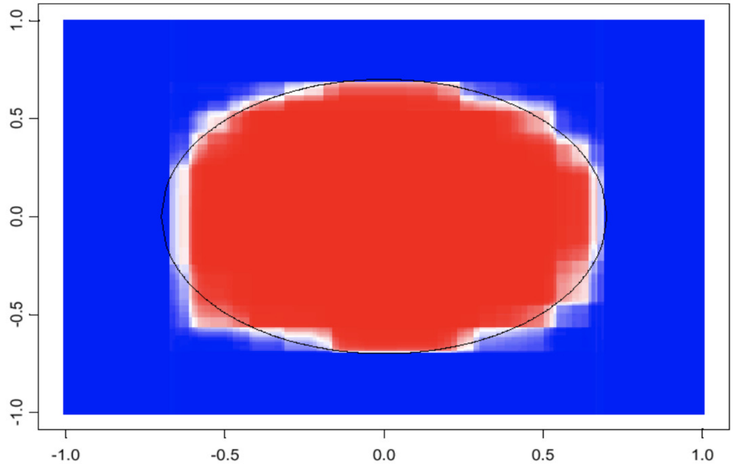
Example for classification - data

Using the CART algorithm (decision tree learning algorithm).



Bagging

Example for classification - 100 bagged trees



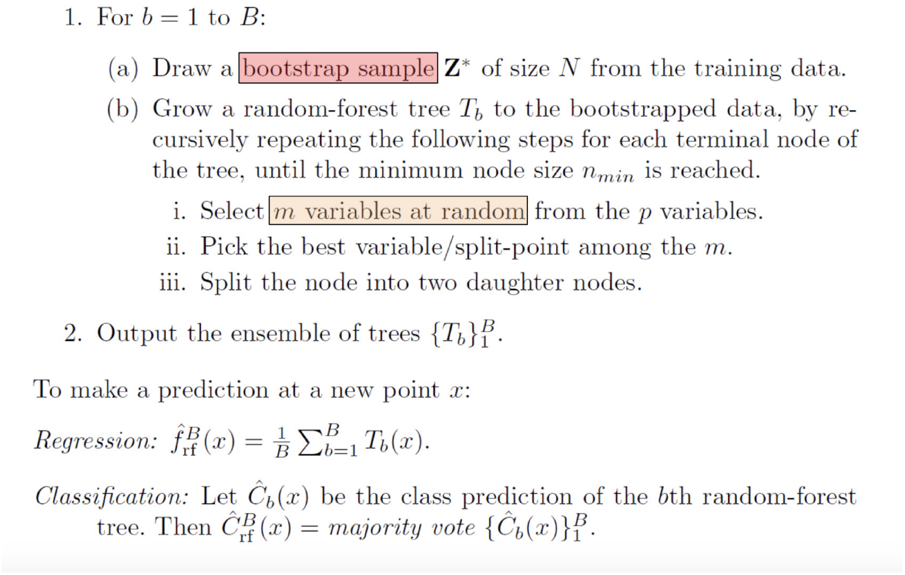
Random Forests

Ensemble methods specifically designed for decision tree classifiers, introducing the two sources of randomness:

- Bagging (cf above)
- Randomness in the classifiers. The latter is obtained by searching at each node the best split amongst a random sample of N_m features (attributes) instead of all attributes.

Random Forests

An algorithm



1. For $b = 1$ to B:
   (a) Draw a bootstrap sample $\mathbf{Z}^*$ of size N from the training data.
   (b) Grow a random-forest tree T_b to the bootstrapped data, by recursively repeating the following steps for each terminal node of the tree, until the minimum node size n_{min} is reached.
      i. Select m variables at random from the p variables.
      ii. Pick the best variable/split-point among the m.
      iii. Split the node into two daughter nodes.
2. Output the ensemble of trees $\{T_b\}_1^B$.

To make a prediction at a new point x:

Regression: $\hat{f}_{\text{rf}}^B(x) = \frac{1}{B}\sum_{b=1}^B T_b(x)$.

Classification: Let $\hat{C}_b(x)$ be the class prediction of the bth random-forest tree. Then $\hat{C}_{\text{rf}}^B(x) = \textit{majority vote}\ \{\hat{C}_b(x)\}_1^B$.

Note: we can use the uncertainty information contained in the set of predictions to do rejection. It consists of classifying as unknown a sample for which less than a certain proportion of the trees agree.

Random Forests

Extremely randomized trees

As an example *Extremely randomized trees* relies on a highly randomized selection of the Binary tests to do at each node. At each node:

- randomly select N_m attributes $a_k \in \{1, \dots, N_D\}, k \in 1 \dots N_D$
- for each attribute, pick a threshold $s_k \sim \mathcal{U}([\min_t X^{a_k}, \max_t X^{a_k}])$
- select the pair (a_k, s_k) maximizing the selected objective score (entropy, Gini index; variance)

Random Forests

Extremely randomized trees

As an example *Extremely randomized trees* relies on a highly randomized selection of the Binary tests to do at each node. At each node:

- randomly select N_m attributes $a_k \in \{1, \dots, N_D\}, k \in 1 \dots N_D$
- for each attribute, pick a threshold $s_k \sim \mathcal{U}([\min_t X^{a_k}, \max_t X^{a_k}])$
- select the pair (a_k, s_k) maximizing the selected objective score (entropy, Gini index; variance)

In general, increasing N_m reduces the bias (by allowing the filtering of potentially irrelevant variables) but increases the variance (by reducing randomization, by fitting more the data).

Also, given the highly random nature of the classifier learning, all the data of the original dataset are usually used, rather than bootstrapped version of it, which should reduce the bias.

✎ Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine Learning*, 63(1):3–42, Apr 2006

Random Forests

Why do we like them?

- they are very simple conceptually and algorithmically
- they allow to deal with any type of signals (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 (in Kinect!)
- they can be analyzed after learning (to some extent, looking at first nodes, in some domains)
- select the pair (a_k, s_k) maximizing the selected objective score (entropy, Gini index; variance)

Outline

Decision Trees

Bagging and Random Forests

Applications

Application to images

Stump classifiers are nice in general, but for images, they correspond to measuring the color (or intensity) value at a predefined position in an image.

Application to images

60/72

Stump classifiers are nice in general, but for images, they correspond to measuring the color (or intensity) value at a predefined position in an image.

This makes such measurements not robust to changes of illumination (contrast, color, etc) or geometry (translation of the object in the image, scaling or simple image resolution e.g. obtained through image resampling, etc).

Application to images

Stump classifiers are nice in general, but for images, they correspond to measuring the color (or intensity) value at a predefined position in an image.

This makes such measurements not robust to changes of illumination (contrast, color, etc) or geometry (translation of the object in the image, scaling or simple image resolution e.g. obtained through image resampling, etc).

To handle this, many computer vision algorithms based on random forests have been working by extracting (local) patches from the image, and classifying these patches into different categories, and then aggregating the results.

Handwritten Digit Recognition

✎ Yali Amit, Donald Geman, and Kenneth Wilder. Joint induction of shape features and tree classifiers. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 19:1300 – 1305, 12 1997

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

Handwritten Digit Recognition

Main algorithmic idea

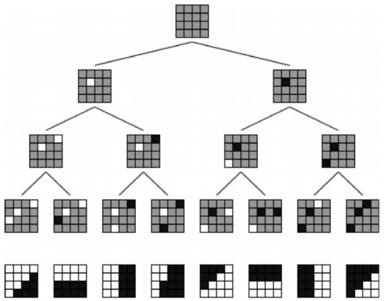
- STEP 1: 'Tag' each pixel p depending on 4x4 patch configurations.
- STEP 2: Build decision trees which look for geometric 'arrangements' of these tags.
- STEP 3: Arrangements are built recursively, by either adding a new tag in relation to an existing one, and selecting the best ones using the Shannon entropy

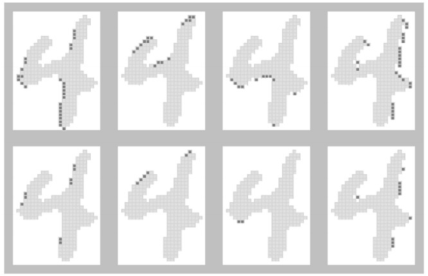
Illustration ...

Handwritten Digit Recognition

Tags

STEP 1: 'Tag' each pixel p depending on whether a 4x4 patch (with the pixel p on the upper left) corresponds to a local patch configuration (the tag category). The configuration is defined by a decision tree.

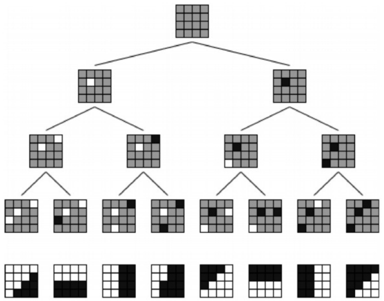


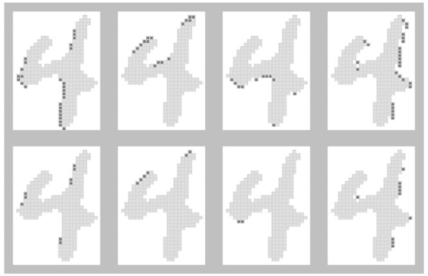


Handwritten Digit Recognition

Tags

STEP 1: 'Tag' each pixel *p* depending on whether a 4x4 patch (with the pixel *p* on the upper left) corresponds to a local patch configuration (the tag category). The configuration is defined by a decision tree.



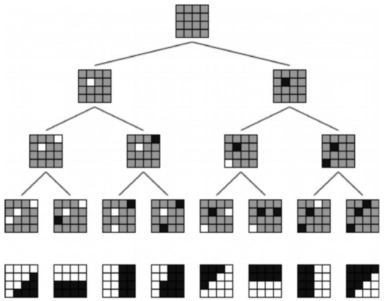


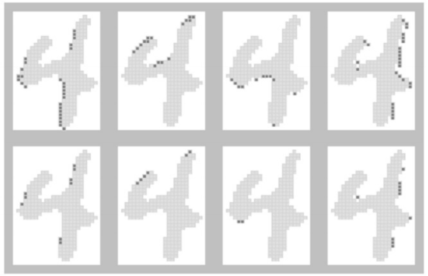
LEFT image: tags (depth 3, provides 8 tags). Most common 4x4 patch configurations falling within a tag leaf found in the data are displayed below the tag.

Handwritten Digit Recognition

Tags

STEP 1: 'Tag' each pixel *p* depending on whether a 4x4 patch (with the pixel *p* on the upper left) corresponds to a local patch configuration (the tag category). The configuration is defined by a decision tree.



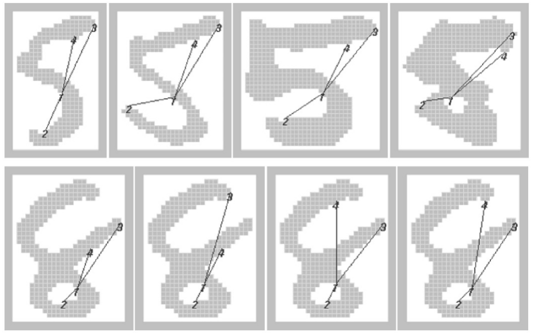


LEFT image: tags (depth 3, provides 8 tags). Most common 4x4 patch configurations falling within a tag leaf found in the data are displayed below the tag.

RIGHT image: First row: all instances in the given image of 4 depth-3 tags. Second row: all instances of 4 depth-5 tags (each of them being an extension in the tree of each of the depth-3 tags of the first row)

Handwritten Digit Recognition

STEP 2: build decision trees which look for geometric 'arrangements' of these tags. The relation between two tags is simply defined by their relative locations (Est, North-Est, North, ...).



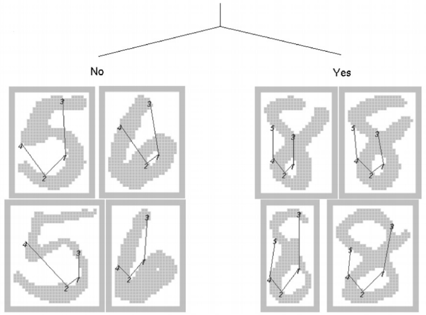
First row: a given arrangement can be seen in quite different images of the same class

Second row: several instances of a geometric arrangement in one 8

Handwritten Digit Recognition

STEP 3: arrangements are built recursively, by either adding a new tag in relation to an existing one (tags from the pending arrangement along the tree path leading to a node), or adding a new relation between two existing tags.

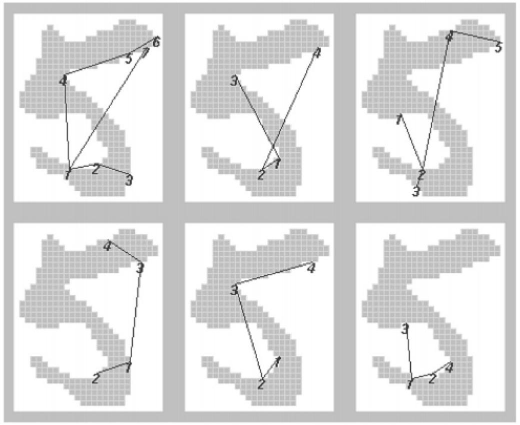
At each node, the best tag/relation/extension (and hence binary decision) is selected amongst a small subset of tags and relation using the Shannon entropy



Example of a node splitting in a typical digital tree.

The binary test involves adding a fith tag (vertex) to the pending arrangement. More specifically, the proposed arrangement adds a fith vertex and a fourth relation to the existing graph which has four vertices and three relations.

Handwritten Digit Recognition



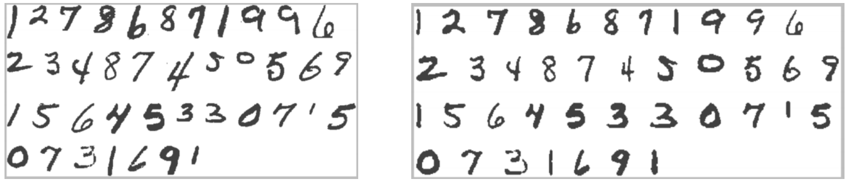
Arrangements found in an image at terminal nodes of 6 different trees.

Handwritten Digit Recognition

Results

Training with portions of the NIST Special Database 3, which contains around 233,000 binary images written by more than 2,000 writers.

Only some basic preprocessing is used (orientation normalization, resolution 32x32 for all images)

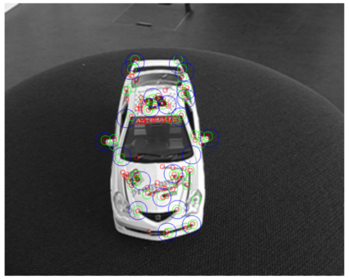


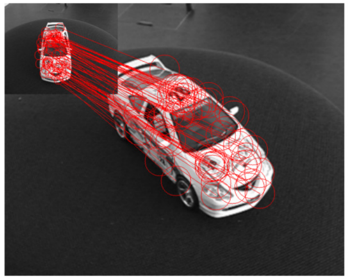
The average depth is 8.8, the maximum 20. The average number of leaf is 600.

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

Point of interest recognition

In many vision applications, one needs to recognize small patches of an image (located around interest points for instance) to match points specified on a reference image into newly acquired image.





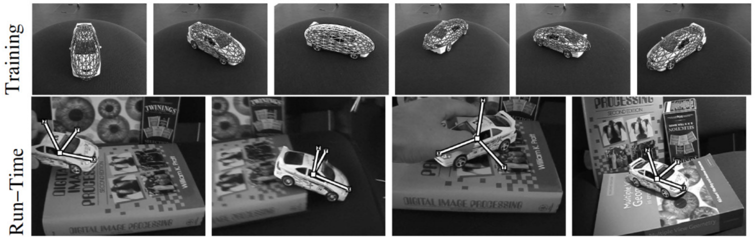
✎ M. Oezuysal, V. Lepetit, F. Fleuret, and P. Fua. Feature harvesting for tracking-by-detection. In *Proceedings of the European Conference on Computer Vision (ECCV)*, volume 3953, pages 592–605, 2006

Point of interest recognition

Principle

Approach principle: show the object, detect interest points, learn a decision tree to recognize them.

After learning decision trees for recognizing the keypoints in the reference model, the detected keypoints in a test image can be identified and used to estimate the 3D pose of the object in the image

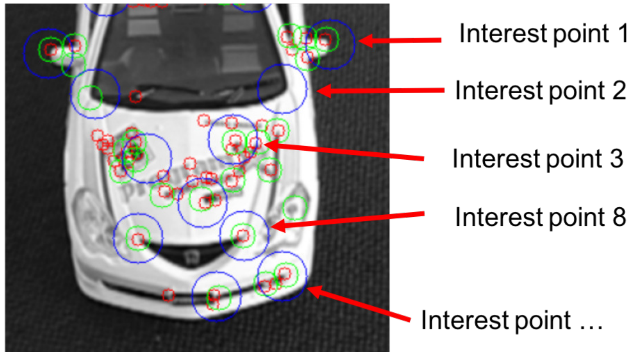


Testing keypoints - paper demo

Point of interest recognition

Tree building

The tree recognizes to which characteristic feature on the object image keypoint belongs to.



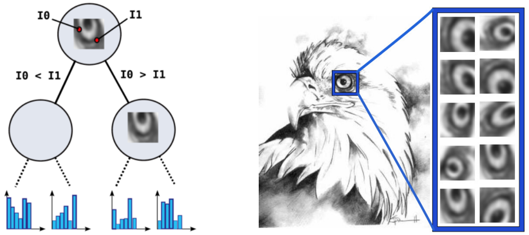
Point of interest recognition

Tree building

The tree recognizes to which characteristic feature an image keypoint belongs to.
Simple binary tests are tested at each node.

Training sets are builts

- from an initial video sequence (tracking points);
- by applying affine transforms to each patch obtained from the sequence (i.e. doing data augmentation by generating virtual samples as if the patch would be seen from different viewpoints).



(Lepetit & Fua, 2006)

Bibliography

Yali Amit, Donald Geman, and Kenneth Wilder.
Joint induction of shape features and tree classifiers.
Pattern Analysis and Machine Intelligence, IEEE Transactions on, 19:1300 – 1305, 12 1997.

L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone.
Classification and regression trees.
Wadsworth, 1984.

Leo Breiman.
Bagging predictors.
Machine Learning, 24(2):123–140, Aug 1996.

Manuel Fernández-Delgado, Eva Cernadas, Senén Barro, and Dinani Amorim.
Do we need hundreds of classifiers to solve real world classification problems?
Journal of Machine Learning Research, 15:3133–3181, 2014.

Pierre Geurts, Damien Ernst, and Louis Wehenkel.
Extremely randomized trees.
Machine Learning, 63(1):3–42, Apr 2006.

M. Oezuysal, V. Lepetit, F. Fleuret, and P. Fua.
Feature harvesting for tracking-by-detection.
In *Proceedings of the European Conference on Computer Vision (ECCV)*, volume 3953, pages 592–605, 2006.