Artificial Intelligence: a concise introduction

Vicenç Torra

School of Informatics
University of Skövde
SE-54128 Skövde, Sweden
e-mail: vtorra@his.se

Chapter 4. Machine Learning

The area of machine learning studies methods whose goal is to improve the performance of a system. This is achieved by means of incorporating knowledge into the system or by means of refining and improve previous knowledge. Because of that, machine learning methods are strongly related to the different types of knowledge representation formalisms. For example, there are methods to learn rules, fuzzy rules, first order logic predicates, Bayesian networks.

In addition to the relationship between machine learning and knowledge representation, there is also relationship between machine learning and search. Machine learning procedures can be seen as search processes. For example, we look for a good set of rules. Then, it is sometimes appropriate to use search algorithms to help in this process. Genetic algorithms are one of the search methods used in machine learning. The next example is about the use of genetic algorithms to learn parameters for fuzzy rules.

Example 1. Let us consider a fuzzy rule based system to represent the relationship between a set of variables. We want to express a given dependent variable in terms of a set of independent ones. Let us consider that we have a set of data for which we know this relationship.

Note that regression methods in statistics focus in the same problem using a (linear) regression model for representing the relationship.

In order to build the system we need to define the rules, and the memberships of the terms involved in the rules.

Then, given the set of rules, and the set of examples to be approximated by the fuzzy rule based system, we can consider the problem of learning the memberships that approximate best the set of examples. Genetic algorithms can be used for this purpose.

At the same time, machine learning is used for improving the performance of search algorithms. Learning can consist on acquiring new knowledge (strategic learning to guide search), learning new operators (macro-operators that are operators defined in terms of simpler and already existing operators), and learning heuristics[1] (learn parameters of a parametric heuristic).

[1] The LEX system[10] was used to learn heuristics for symbolic integration

1 Introduction

The most usual way to classify machine learning methods is according to how we supervise the result (i.e., the degree of supervision). That is, what type of information is available on the outcome expected by the system in a given circumstance. It is usual to consider supervised, unsupervised and reinforcement learning.

Supervised learning. In this case the expected outcome of the system is available for a set of examples, also known as instances (of the problem). The goal is to build a system that approximates as well as possible the outcome for the examples given, and that when the system faces new cases the outcome delivered is also good. Example 1 corresponds to this type of learning. In this type of problem we have data and we consider that there are some variables that depend on other ones. I.e., there is an output (the expected outcome) of the system that should depend functionally on another set of variables. That is, we have data with both dependent and indepedent variables, and the goal of the learning process is that the system approximates the dependent variable. In other words, we want to find a model for the functional dependency.

A supervised learning process can often be formulated as an optimization process, finding a good approximation of a function from a set of pairs (input, output).

The term *learning from examples* is also used in this setting. The set of examples used to build the model is know as the training set.

In order to formalize this type of learning, we consider a training set C with N examples. Each example is the pair (x, y) where x is a vector of dimension M and y is the outcome of applying a function f (which is not known) to vector x. Therefore, we have N examples in a given M dimensional space. That is, $X = \{x_1, \ldots, x_i, \ldots, x_N\}$. We usually say that the examples X_i are described in terms of M variables or attributes $A_1, \ldots, A_M$. With $DOM(A_k)$ we denote the domain or range of variable A_k. Then, we use $A_j(x_i)$ to express the value of variable A_j of example x_i.

Missing values. It is usual in real data that some variables are not specified for some examples. These cases are known as missing values. Using the notation above $A_{j_0}(x_{i_0})$ is not defined for example i_o and variable j_0. Missing values can be due to data of low quality (i.e., difficulties in data collection, errors, and unknown values in questionnaires), but they can also be made on purpose. For example, some methods for data privacy reduce disclosur risk by means of data suppression. Not only the suppression of sensitive data but also the suppression of other data that can help intruders to guess sensitive data (this is known as secondary suppression)

In case that there are measurement errors, we have that y is $f(x)$ plus a given error ϵ. That is, $y = f(x) + \epsilon$. From this information we build a model

that we denote by M_C (we use the subindex C because the model depends on the examples C).
The goal is that the model M_C applied to an element x leads to something similar to $f(x)$. In other words, that $M_C(x)$ is an approximation of $f(x)$ and $M_C(x) \sim f(x)$. When there is no problem of confusion, we use simply M to express the model.
Within supervised machine learning it is convenient to distinguish between different types of problems, because it is usual to apply different algorithms to each type. We describe them below.

- **Regression problems.** Each example includes a variable with the solution (i.e., variable y above), and this variable is numerical. The goal of the learning process is to build a model of the attribute. Statistic regression methods, as linear regression methods, and neural networks are examples of methods for solving this type of problems.
- **Classification problems.** This is a problem similar to regression but the variable to be learnt is categorical or binary. Most methods to learn rules are of this family.

Other types of problems: similarity learning (and metric learning), sequence learning, preference learning.

Reinforcement learning. In this case we have only partial knowledge on the performance of the system. We do not have the output of the system but only a reward or a penalty that give a rough idea of the performance. For example, if we consider planning robot trajectories, the number of colisions and the time used to achieve the goal give an idea of the performance.
Sutton and Barto [16] define reinforcement learning as follows:

> Reinforcement learning is learning what to do – how to map situations to actions – so as to maximize a numerical reward signal. The learner is not told which actions to take, as in most forms of machine learning, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics – trial-and-error search and delayed reward – are the most important distinguishing features of reinforcement learning. (Sutton and Barto, 2012 [16], p 4).

Unsupervised learning. The only information available is on the inputs of the system, but no specific action or output is expected. The algorithm tries to extract useful knowledge from the information available. This corresponds to finding similarities and regularities (i.e., interesting patterns) in the data. Because of that, we have data described in terms of variables, but there is no distinguished variable (i.e., an output variable) as we have in supervised learning.
Clustering agorithms, which build clusters and taxonomy from data, are examples of unsupervised machine learning. Association rule mining algorithms are also unsupervised machine learning tools. Methods to find latent

variables as principal components, SVD and the EM algorithm can also be seen from this perspective.

Another way to classify machine learning algorithms is to distinguish between deductive and inductive learning. We have inductive learning when knowledge is extracted from examples, and we have deductive learning when we only reelaborate knowledge already existing. The knowledge implicit in previous knowledge is also known with the term chunking.

1.1 Knowledge representation and learning

The algorithms discussed in some detail in the next sections do not take into account domain knowledge. They are based only in examples from which knowledge is acquired. Because of that, they can be seen as knowledge modeling algorithms.

There are machine learning methods that use (or even need) previous knowledge in order to build more complex knowledge structures. Inductive logic programming is an area that studies and develops these type of methods. We have to mention here also the algorithms for improving the performance of search algorithms, as they also need knowledge. Cognitive architectures, that integrate knowledge representation, problem solving algorithms and machine learning methods, also relate knowledge representation and learning. Examples of these architectures are Soar and Prodigy.

Inductive logic programming systems build models from examples. The difference is that they use knowledge to overcome some of the limitation of other inductive algorithms. In particular, they try to overcome the following three limitations.

- **Limited representation.** Inductive methods only build models with expressions that correspond to propositional calculus. Therefore, they can not build models representing propositions on first-order calculus.
- **Do not consider previous domain knowledge.** Inductive methods learn descriptions from examples, and all they learn has to be in the examples. Other approaches as macro-operators focus on existing knowledge, reformulating it, but then no new knowledge can be learnt.
- **Vocabulary bias.** Methods can only use those terms that appear in the examples. It is not possible to invent new terms.

Inductive logic programming tries to solve these limitations with methods based on first-order logics. This is used to represent what is already known (previous knowledge) and to represent what is learnt. Examples of inductive programming systems include: MIS, SOIL, CLAUDIEN, and GOLEM.

Given a previous knowledge K, a set of positive examples $E+$, and a set of positive examples $E-$ (examples that can not be proven from K), inductive logic programming systems builds a set of clauses in first-order logic H such that they prove $E+$ but they do not prove $E-$.

We illustrate below an example that consists on finding Prolog predicates to define the quicksort algorithm.

Example 2. [11] Definition of quicksort algorithm using inductive logic programming. To do so we need the previous knowledge K and the set of examples $E+$ and $E-$.

Background knowledge:

```
partition(X,[],[],[]).
partition(X,[Head|Tail], [Head|Sublist1], Sublist2) :-
   lte(Head, X),partition(X, Tail, Sublist1, Sublist2).
partition(X, [Head|Tail], Sublist1, [Head|Sublist2]):-
   gt(Head, X),partition(X, Tail, Sublist1, Sublist2).
append([],List,List).
append([Head|Tail], List,[Head|Rest]):-append(Tail, List, Rest).
lte(0,0). lte(0,1). ...
gt(1,0). gt(2,0). gt(2,1). ...
```

In these definitions we have the predicates `append` and `partition`. The first is satisfied when the third list corresponds to the concatenation of the first two lists. The predicate `partition` has four parameters. First a numerical value `X`, and then three lists. The predicate is true when the second and the third list define a partition of the first list so that in the second list we have the elements less than or equal to `X`, and the third one contains the elements larger than `X`. Therefore, `partition(4, [1,6,2,5,3,4], [1,2,3,4], [6,5])` evaluates in true.

We have also among the definitions predicates for `lte` and `gt`. The former is satisfied when the first numerical value is smaller than or equal to the second. The latter is satisfied when the first numerical value is larger than the second.

In addition, we need some positive and negative examples.

Positive examples:

```
qsort([],[]). qsort([0],[0]). qsort([1,0],[0,1]). ...
```

Negative examples:

```
qsort([1,0],[1,0]). ...
```

With this information, the inductive logic programming system GOLUM generated the following model for `qsort`:

```
qsort([A|B],[C|D]):-
   partition(A, B, E, F), qsort(F, G),
   qsort(E, H), append(H, [A|G], [C|D]).
```

and also the following model for `qsort`:

```
qsort([A|B], [C|D]):-
   qsort(B,E), partition(A,E,F,G), append(F, [A|G],[C|D]).
```

The first solution is the usual implementation. That is, for a list with head `A` and tail `B`, we divide elements of `B` in two sets: elements less than or equal to `A`, and all the others. Then, we order the two sets and append them including `A` between them.

The second solution is also an implementation of a predicate to order lists. Nevertheless, it is an inefficient solution. For a list with head `A` and tail `B`, we first order `B`. Then, we partition `B` using `A`. Therefore, we have `F` and `G` which are ordered, the former with the elements less than and equal to `A` and the latter with the elements larger than `A`. Thus, their concatenation (including `A` in the right position) is an ordered list.

1.2 Preprocessing and basics

We will use the following notation. We have a data set $X = \{x_1, \ldots, x_N\}$ of N objects. Each object is represented in terms of M attributes $A = \{A_1, \ldots, A_M\}$. We use $A_t(x_l)$ to denote the value of attribute A_t of object x_l. $A(x_l)$ corresponds to the vector $(A_1(x_t), \ldots, A_1(x_t))$. The domain or range of attribute A is denoted by $DOM(A)$. When the attribute A_t is categorical, we denote its terms by $DOM(A) = \{a_{t1}, a_{t2}, \ldots, a_{tR_t}\}$.

It is usual to consider the following types of variables.

- Numerical variables.
- Binary or Boolean variables.
- Ordinal variables. Set of ordered terms. We can compare them with $>$ and $\geq$.
- Nominal variables. Set of terms. The only allowed operation is to check for equality.

If needed, we can transform ordinal and nominal variables into binary variables.

Distances and similarities Machine learning algrithms need functions that evaluate how similar are two objects, or at which distance they are.

Similarity coefficients permits us to evaluate and quantify the similarity between pairs of objects. These coefficients can be clasified in four classes: based on distances, on associations, on correlations and probabilistic similarity.

2 Supervised machine learning

2.1 Consistency

It is important for any learning method to be consistent. A method is consistent when the size of the training set increases, the model tends to converge to the correct solution. This property comes from statistics, where we have the definition of consistent estimator. See e.g. [8] (p.130).

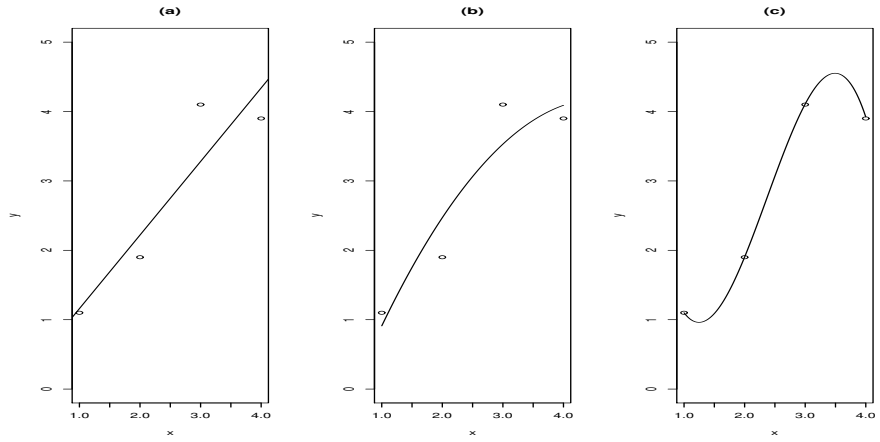

Fig. 1. Three different generlizations for a set of points.

2.2 Bias and variance

When we are building a model from data (for regression or classification) we expect that our model is able to generalize from the examples given in order that we can apply our model to new situations.

For example, if we build a classification rule to classify patients, we expect that the rule can be correctly applied to other patients. Not only the ones in the set of examples. This is so because we do not expect the set of examples (the training set) to be exhaustive, but only a sample of possible situations. So, the method that builds the rule should be able to generalize our data so that the scope of the rule is not too much narrow or too focused to the training set.

The generalization poses several difficulties, because from a single data set there are several possible generalizations. They depend on our assumptions on the model.

Figure 1 illustrates the generalization problem with an example with one input variable and one output variable. The figure displays four points that represent four examples (training set) and three different models. Each model generalizes the four examples in a different way. The models are based on: (i) a regression line, (ii) a polynomial of degree two, (iii) a polynomial of degree three.

The construction of a model depends on the knowledge representation formalism. In the case above, the model is either a regression line, a polynomial of degree two or a polynomial of degree three. Each formalism permits to represent correctly some situations, and others may not be representable. This causes some bias. The bias is some predisposition or inclination of the system to learn some aspects and ignores some others.

In addition to the bias caused by the knowledge representation formalism, we have bias due to the search algorithm (the approach used to determine the model). This is the inductive bias. For example, in case of different possible options, select the simplest (or shortest) one. This is related to Occam's razor.

The use of bias has advantages (e.g., model simplicity and reduced computational costs) and inconveniences (e.g., we may be unable to build the appropriate model). Selection of an appropriate knowledge representation formalism needs to find a good balance between them.

Formalization In regression problems, the bias is computed as the difference between the expected estimated value of our model (if x_0 is the input of our model M, the estimated value is $M(x_0)$) and the value of the function for this input (if f is the function, that is $f(x_0)$). That is,

$$Bias(M(x_0)) = E(M(x_0)) - f(x_0).$$

Note that $E(M(x_0))$ is the expected estimated value of our model for x_0 and that the expectation of a random variable X with density function p is $E(X) = \sum_{x \in X} x \cdot p(x)$. In an experiment, we would consider models built from training sets and we approximate this expectation by the mean of the outcome of the corresponding models. For example, if we have five training sets $C = \{C_1, C_2, C_3, C_4, C_5\}$, then $E(M(x_0))$ is the mean of the outcome of the models built from $C_1, \ldots, C_5$. If we denote them by $M_{C_1}, M_{C_2}, M_{C_3}, M_{C_4}, M_{C_5}$, then,

$$E(M(x_0)) = (M_{C_1}(x_0) + M_{C_2}(x_0) + M_{C_3}(x_0) + M_{C_4}(x_0) + M_{C_5}(x_0))/5.$$

Naturally, we have that a method has no bias when the expected estimated value correponds perfectly to the function f at x_0. On the contrary, we have bias when these values differ. It is important to note that not having bias does not imply that the models approximate correctly $f(x_0)$. We may have $E(M(x_0))$ but $M_{C_i} \neq f(x_0)$ for all M_{C_i}.

Another aspect to be taken into account is the variance, that corresponds to the dispersion of values around the expectation. If our model is such that $E(M(x_0))$ is near to $f(x_0)$ but the variance is large, it means that in most of the cases (or in all cases) we will get a value $M(x_0)$ that is far from $f(x_0)$. Among the learning algorithms with the same bias, we will prefer those with lower variance.

The variance of the learning method is defined as follows:

$$Variance(M(x_0)) = E([M(x_0) - E(M(x_0))]^2).$$

Recall that the variance of a sample $\{x_1, \ldots, x_N\}$ is $s^2 = (1/N)\sum(x_i - E(X))^2$, and when $E(X) = 0$ is of course $s^2 = (1/N)\sum x_i^2$.

Bias and variance and error of a model In this section we consider that we model the outcome of a system, that corresponds to the function f discussed

in the previous section. Nevertheless, there is some error in the outcome, and because of that the real outcome is $f(x) + \epsilon$. We assume that ϵ has zero mean. Then, the error between the system and a model constructed from a training set C (i.e., the model M_C) is

$$e = M_C(x) - y = M_C(x) - (f(x) + \epsilon).$$

It is possible to prove that

$$E(e^2) = E(\epsilon^2) + Bias^2(M(x_0)) + Variance(M(x_0)).$$

Trade-off between bias and variance Let us consider what happens with bias and variance when the complexity of models increase. As a simple example, consider the approximatation of a function in $\mathbb{R}$ using polynomials. Here the complexity is the degree of the polynomial. The larger the degree, the larger the complexity. Note that for a polynomial of degree m the model (i.e., the corresponding polynomial) has $m + 1$ parameters.

It is easy to see that when the complexity of models increase, these models approximate better the outcome (they approximate better the function and the error ϵ). Because of that, when the complexity increases, the bias decreases.

At the same time, when the complexity increases, the variance increases. Note that small changes in the examples, will change largely the model.

Observe that in Figure 1 (a) we have that the bias is large and the variance is small (because changing one point does not affect largely the regression line). In contrast in (c) we have that the bias is small (zero for the four points in the training set) but that the variance is large, because modifying a point will modify largely the curve.

We have stated that the error of the training set tends to decrease when the complexity of the model increases. If we consider a different set for testing the performance of the model (a test set), we have that the error of this set has a different behavior. Initially, the error tends to decrease when the complexity increases, but from a certain point the error tends to increase if the complexity is further increased. The region in which the error starts to increase usually correponds to a good complexity value. Complexities larger than this one usually means that the model is too fitted to the training set. This is known as overfitting. That is, we have that from this point, the model does not generalize correctly but that it learns the examples including the error ϵ.

2.3 Evaluation and metrics

The simplest type of classification problem is when the class to learn is binary. In this case it is usual to consider one of the classes as the positive one, and the other as the negative.

When we apply the model we have built, and we compare what the model returns and what should be returned, we find different situations. In the case of binary classification problems with a positive and a negative class, we have four

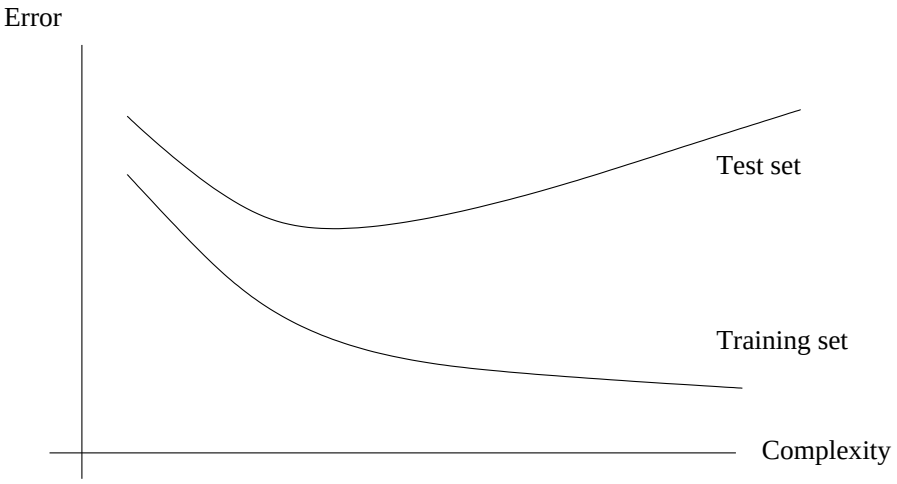

Fig. 2. Error for the training and test sets.

situations. They are as follows. Figure 3 represents graphically these four cases. It is usual to use the term ground truth[2] to refer to what is expected, i.e., the real class of a record.

- **True positives.** The model returns a positive class and it should be a positive class. Therefore, the answer is correct. We denote the number of true positives by TP.
- **True negatives.** The model returns a negative class and it should be a negative class. Therefore, the answer is also correct in this case. We denote the number of true negatives by TN.
- **False positives.** The model returns a positive class, but it should return a negative class. The outcome is incorrect. We denote the number of false positives by FP. This type of errors is known as type I error in statistics.
- **False negatives.** The model returns a negative class, but it should return a positive class. Therefore, the outcome is also incorrect. We denote the number of false negatives by FN. This type of errors is known as type II error in statistics.

The goal of a supervised machine learning method is to have as few as possible false negatives and false positives. However, it is important to note that not all errors may have the same importance. For example, if we have a system to detect severe illnesses, we may prefer to have false positives than false negatives.

[2] In some applications, the ground truth is not know and cannot be found. For example, when we consider the diagnosis of a patient. In this case, it is usual to consider the gold standard. This is the best diagnosis that can be made in current circumstances.

For example, in a system to detect cancers, it may be better to raise an alert and inform that it is possible that someone has cancer when it is not, than to ignore a case of cancer. That is, we prefer to be in the safe side.

There are some measures based on these four situations.

- **Accuracy.** This corresponds to the percentage of cases correctly classified. That is,
$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}.$$
- **Recall (of the positive classes).** This corresponds to the proportion of cases correctly classified as positive among those that are truly positives. Therefore, this is
$$Recall = \frac{TP}{TP + FN}.$$
This measure is also known as the true positive rate (TPR) and sensitivity. In an information retrieval system[3], we have 100% recall when all documents matching are retrieved, and in a diagnosis system, we have 100% recall when all positive cases of an illness are detected. Note that we can have 100% recall when uninteresting documents are also retrieved, or when we classify as positive cases that should not be classified as such. In particular, note that a binary classifier that assigns the positive class to any situation, will have a 100% recall.
We have discussed above the case of a system to detect cancers, and avoiding false negatives. A system that tries to not miss any positive instance, will focus on having a large recall.
- **Precision (of the positive classes).** This corresponds to the proportion of cases correctly classified as positive among those that are classified as positives. That is,
$$Precision = \frac{TP}{TP + FP}.$$
This measure is also known as positive predictive value (PPV).

These measures can be extended to multi-class problems. That is, to problems in which the class is not binary, but that there are several possible outcomes. In this case, we define recall and precision for each possible class. The definitions given below are for a given class c_i. The definition uses F_{ij} to denote the number of cases classified as belonging to class c_i when the correct class is c_j. I.e., the model states c_i and the ground truth is c_j.

$$Recall_i = \frac{F_{ii}}{\sum_k F_{ki}}$$

[3] Information retrieval is about selecting documents that match a given query. Here, matching does not necessarily mean that we find in the document the list of terms of a query, but e.g. semantic matching, or queries in natural language may be possible.

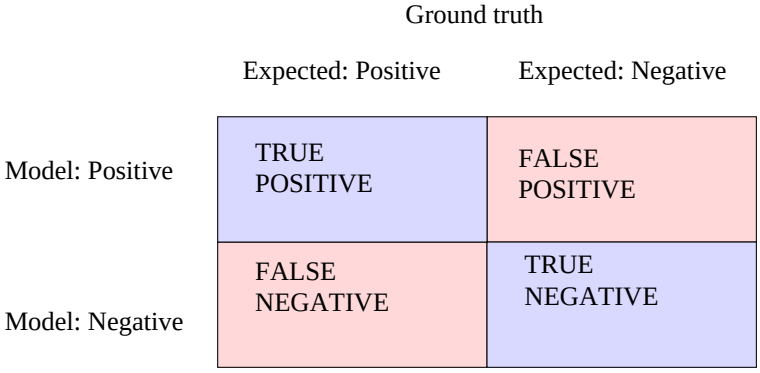

Fig. 3. Possible situations for binary classes (model vs. expected outcome).

$$Precision_i = \frac{F_{ii}}{\sum_k F_{ik}}.$$

In general, both precision and recall are necessary in an application. Because of that there are two indices that average precision and recall. They are the F-measure and the G-measure. The difference between them is on the type of mean[4] they use to average the two terms.

- F-measure. It is defined as the harmonic mean of the precision and the recall.

$$F\text{-}measure = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

  This measure is also known as the F1-score because there is the F_β, a weighted harmonic mean, and for $\beta = 1$ we have the F-measure.
- G-measure. It is defined as the geometric mean of the precision and the recall.

$$F\text{-}measure = \sqrt{precision \cdot recall}$$

Additional related concepts: ROC curve (ROC for reeceiver operating characteristic) and AUC (area under the curve).

[4] A *simpler* way to average precision and recall would be to use the arithmetic mean. That is, use

$$(precision + recall)/2.$$

If people use geometric and harmonic mean instead is because these other means compensate less when one of the terms lead to bad results. Discussion on different ways to aggregate can be found in [17].

Validation When there is no set to validate a model (all the data are for the training set), we can use cross-validation. It uses some of the examples in the training set for learning and others for testing.

- k-Fold cross-validation. The set is divided into k parts with the same size. Then, we use one of these parts for testing and the others for training. We repeat the same with the other parts. In this way we can apply the procedure k times with each of the k parts. We can compute mean values of statistics for the k different executions, and in this way e.g. compare machine learning algorithms, or parameterizations.
- Leave-one-out cross-validation. This uses one record for testing and the other for learning the model. We repeat the same process considering each of the records in the training set. This is equivalent to k-fold cross-validation with k equal the cardinality of the set.

2.4 Decision trees

Decision trees permit us to classify objects by means of a sequence of tests. The tree is defined with questions in the nodes and classes (or probability distributions on classes) in the leaves. Figure 4 represents a decision tree built from the iris dataset [**?**] to classify three species of the iris flowers (iris setosa, iris versicolor, and iris virginica). This tree built using the function `ctree` of package partykit [7] in R has a probability distribution over the set of classes in each leaf.

In order to use the tree for the classification of a given record, we proceed from the root of the tree (the top) and go downwards following the path indicated by those questions that evaluate into true. The leaf indicates the class in which we classify our record. For example, if our record indicates that petal length is 2.3 and that petal width is 1.9, we will follow the path that leads to node 7.

```
Algorithm class (record, tree) is
  if (leaf(tree)) return class(tree)
  else {
    subtree = select-sub-tree (node,record)
    class(record, subtree) }
end algorithm
```

Formally, decision trees are equivalent to logical rules where a set of predicates are used to infer a class. For example, the tree in Figure 4 includes the following rule.

If *petal.length(x)*> 1.9 and *petal.width(x)*>1.7 then iris.setosa

Decision trees are an effective method in learning. We can use them to approximate any function. The goal of machine learning algorithms for decision trees is to build trees with minimal height. That is, that the number of evaluations needed to classify a record is a small as possible. We outline below a greedy method to build the tree. The definition is recursive. At each point we

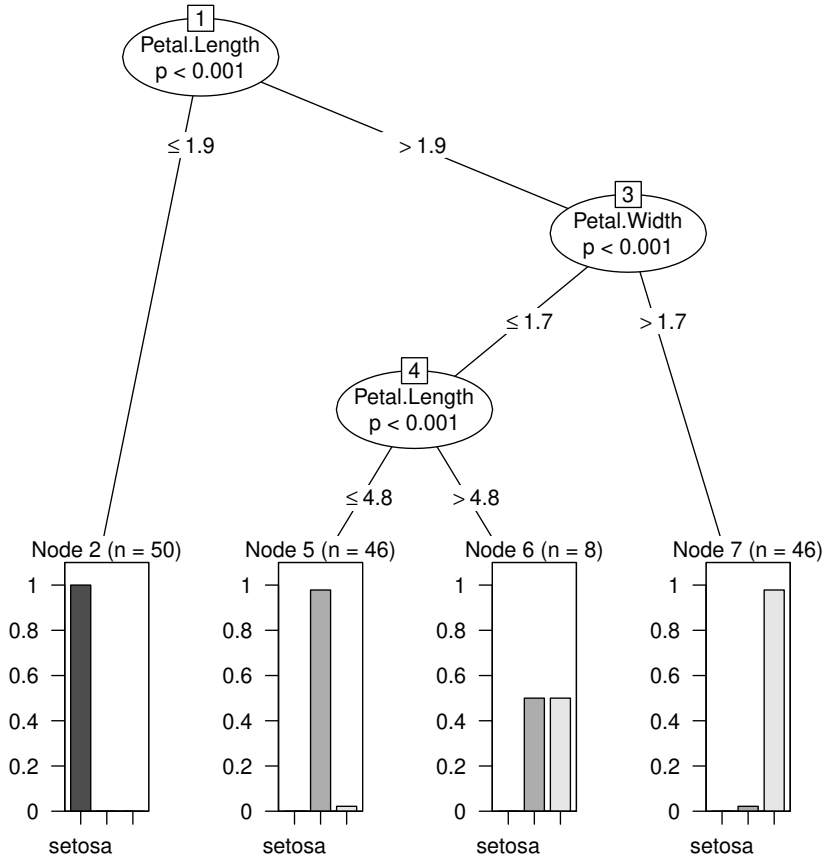

Fig. 4. Decision tree.

select which is the best partition (and best question to be asked). Then, the records in the training set are partitioned in disjoint sets (one set per branch) and the same procedure is applied recursively. The process is stopped when all records in the training set are in the same class or if there are no records left. This greedy approach is the most used one, differences between methods are on how the partition is done at each point, and on whether we need to process all records in the training set. With respect to partition generation some measures as the information gain and Gini gain are used for this purpose. With respect to processing all records, some algorithms (as the `ctree` mentioned above) do not expand the whole tree until all records in the training set have the same class. The expansion is stopped when there are still records of different classes in the

training set. This is done, among other reasons, to avoid overfitting. Decision trees are consistent.

The term splitting is used for the process of building a partition.

```
Algorithm buildDecisionTree (trainingSet, majority) is
  if (empty(trainingSet)) { leaf(class = majority) }
  else {
    if (class(r1)==class(r2) for all r1!=r2 in trainingSet) {
         leaf(class = class(r1)) }
    else {
       P = set of testable partitions (trainingSet)
       Pbest = evaluate and select (P)
       for each subsetOfP in P {
         allSubtrees[subsetOfP] = buildDecisionTree(subsetOfP,
                                                    majority(trainingSet)) }
       tree(question according to Pbest, allSubtrees)
    }
  }
end algorithm
```

We describe now how to select a partition and the corresponding question based on the information gain. This approach is based on the entropy. The process assumes that all variables are categorical (nominal, i.e., without ordering on the set of terms), and, thus, each variable is described in terms of a set of terms. Let $\alpha_1, \alpha_2, \ldots, \alpha_r$ the set of terms for variable V. We also assume that the examples of the training set have only two classes (positives and negatives).

At any point, given a node of the tree with a set of records in the associated training set, we have that we can infer a probability distribution on the output classes.

For example, let us consider a node N for a decision tree for only classifying positive and negative records. If np and nn denote the number of positive and negative records, we have that we can define a probability distribution on the node N by $p_N = (np/(np + nn), nn/(np + nn))$. It is clear that we can also compute the entropy for this node N. It will be

$$H(N) = H(p_N) = -np/(np+nn)\log_2 np/(np+nn) - nn/(np+nn)\log_2 nn/(np+nn).$$

Now, for the node N we consider possible partitions (a new question to be added into the tree). We consider variable V with its terms $\alpha_1, \alpha_2, \ldots, \alpha_r$. When we consider the question generated by this variable, we will have that the node has r sons, one for each term. We call these sons N_{α_i}. When we divide the training set according to the terms, we will have that for α_i and the son N_{α_i}, we have np_i elements which are positive and nn_i which are negative for a total of $np_i + nn_i$ that have $V = \alpha_i$. Using this information, we can calculate the entropy for each node N_{α_i}. In a way similar to our previous computation for $H(p_N)$ we define

$$H(N_{\alpha_i}) = H(np_i/(np_i + nn_i), nn_i/(np_i + nn_i)).$$

Then, we compute for the variable V the average of all nodes N_{α_i} as follows.

$$H(N|V) = \sum_{i=1}^{r} \frac{np_i + nn_i}{p+n} H\left(\left(\frac{np_i}{np_i + nn_i}, \frac{nn_i}{np_i + nn_i}\right)\right).$$

The information gain we get when we use variable V is the difference between these two entropies. The one before selecting V and the one that uses the variable V in the partition. That is,

$$IG(N, V) = H(N) - H(N|V).$$

At a given node, we would select the variable that maximizes the information gain. That is, we would compute for all variables not already used in previous (higher) nodes of the tree the information gain, and then select the one with maximum value.

Figure 6 represents another decision tree, it is for the wine-black dataset [**?**].

[12] is a detailed survey published on 1998 on the construction of decision trees. It discusses almost 400 references, including different approaches for variable selection and splitting. [9] and [15] discuss the split selection bias. The former describes in detail the different approaches, including the formulas for computing the best selection. Greedy methods for selecting a split are biased when different variables have different number of split points. [15] studies this problem for binary classification problems.

2.5 Nearest neighbor and *k*-nearest neighbor

These are very simple methods that can be used for both regression and classification problems.

Given a training set C and a distance d on the space of C, the nearest neighbor defines the model M_C for a new data element x as the value of the record in C that is a minimum distance to x. That is,

$$M_C(x) = y(\arg\min_{x' \in C} d(x', x))$$

In this expression, given the training set $C = (X, Y)$, I use $y(x)$ to denote the class of x.

The k-nearest neighbor is similar, but in this case we retrieve the k nearest neighbors and we use them to compute $M_C(x)$. In classification problems, the output is usually the majority of the outputs of the retrieved set. In regression problems, we usually return the average of the retrieved set.

The nearest neighbor relates to case-based reasoning (CBR) and to metric learning.

Missing values and supervised machine learning. Some models are better suited than others to deal with missing values. A model based on *k*-

nearest neighbors can deal easily with missing values. We can define distances that can deal with them in an appropriate way.

2.6 Neural networks

Artificial neural networks are inspired in natural neural networks. The terminology we use in machine learning is inspired in the terminology in biology.

Formally, a neural network is represented by means of a graf where nodes are the computation units (neurons) and the edges represent information flow between neurons (connections between neurons). Each edge has weight, and each neuron has a function, the activation function, that represents in what extent the inference is propagated to the output.

In fact, there are neural networks for unsupervised learning and also for supervised learning. The latter are used for approximating functions from examples. They are usually used for regression problems. Self-organizing maps are seen as unsupervised neural networks.

The notation we use in this section is slightly different from what we are using in the remaining part of the chapter. Given an example (x, y), the input values are $x = (x_1, \ldots, x_N)$ with $x_i = A_i(x)$, and the output values are $y = (y_1, \ldots, y_M)$. In general, both input and output are numerical values. Given the training set (a set of pairs), the goal is to define an architecture of the network, and the weights for the connections so that when we apply the neural network NN to an element x of the training set (i.e., $NN(x)$) we get a result similar to y.

Figure 7 represents a neural network with an input layer of dimension N, an output layer of dimension M, and a hidden layer. Hidden layers represent intermediate computations. To operate, the network proceeds as follows. Each unit takes the values associated to their inputs and combine them with the weights of the corresponding connections, then the resulting value is propagated into the output using the activation function of each unit (neuron).

If for a given neuron, we have four input connections represented with values x_1, x_2, x_3, x_4 with weights, respectively, w_1, w_2, w_3, w_4, and an activation function f, the output of the neuron is:

$$f(w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4).$$

In some models, neurons include a bias that permit to influence the output. We use θ to represent this value.

Figure 8 (left) represents a neuron and (right) with a bias. In this case the output of the neuron is:

$$f(w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + \theta).$$

Approximation of functions using neural networks An important result when we are considering the use of neural networks as a knolwedge representation formalism is the theorem proven by Cybensko in 1989 [2]. This theorem states that neural networks are universal approximators.

Theorem 1. *Given a real continuous function Φ in the unit cube of $\mathbb{R}$ ($\Phi : [0,1]^N \rightarrow [0,1]$) then for all ϵ there exists a neural network NN with N input neurons, a single hidden layer with neurons with sigmoids $f(x) = 1/(1+e^{-x})$ as activation functions, and a single output with linear activation function $f(x) = x$ that satisfies*

$$sup\ |\Phi(x) - NN(x)| < \epsilon.$$

This theorem only proves the existence but does not give clues on how the networks can be built for particular problems. Nevertheless, it means that for a given problem, we can consider the use of neural networks.

There are different similar results about universal approximation using neural networks. See also [4,5]. [6] proves that neural networks can not only approximate a function but also its derivatives.

Learning using backpropagation Backpropagation is an iterative way to compute the weights of a neural network. The process takes an example in each step and then updates the weights taking into account the error of the network for this example. Because of that, we can see the algorithm in terms of two steps: propagation and adaptation.

- **Propagation (function computation).** We take an example of the training set, and compute the output for this example. Once the output value is obtained, we compute the error of this value with respect to the output expected for this example.
- **Adaptation (learning step).** We modify the weights using the error computed. We say that the modification of weights is done backwards, the error of the output is propagated backwards towards the input.

This process is done for each of the examples of the training set, and, in fact, examples are used as many times as needed so that the error is reduced. An iteration with all the examples is known as *epoch*.

When we achieve an appropriate error, we have that the internal representation permits to obtain suitable outputs for the inputs learnt. That is, we have a function that generalizes the examples given in the training set. Therefore, the network can then be applied to data that is not necessarily the one we have been using in the training of the network.

Let us consider the backpropagation of a neural network of three neuron layers, where only connections are permitted between contiguous layers. We do not permit here connections that go backward (this permits that previous computations/state are used later), so, no there are no cycles in the network. This type of network is known as feedforward neural network. In addition, we do not permit connections that skip layers.

We use the architecture of Figure 7. That is, an input layer with N neurons, a hidden layer with L neurons, and an output output with a single output. The hidden layer receives the data from the input layer, and the output from the hidden layer.

In order to express the computations, we need some additional notation. First, we need to express the weights of each connection. We use w^h_{ji} to denote the weights of the connections between the ith neuron of the input with the jth neuron of the hidden layer (so $1 \leq j \leq L$, $1 \leq i \leq N$), and we use w^o_{kj} to denote the weights of the connections between the jth neuron of the hidden layer and the kth neuron of the output layer (so $1 \leq j \leq L$, $1 \leq k \leq M$). Second, we need to denote the activation functions. We use f^h_j and f^o_k to denote, respectively, the activation function of the jth neuron of the hidden layer, the kth and of the output layer. Similarly, we use θ^h_j and θ^o_k to denote the bias of the jth neuron of the hidden and the kth of the output layers.

Using this notation, it is clear that the output of the jth neuron of the hidden layer is computed first accumulating the inputs of the neuron (weighting each input by its weight) as follows

$$acc^h_j = \sum_{i=1}^{N} w^h_{ji} x_i + \theta^h_j,$$

and then applying the activation function to this accumulated value

$$s_j = f^h_j(acc^h_j) = f^h_j \left(\sum_{i=1}^{N} w^h_{ji} x_i + \theta^h_j \right). \tag{1}$$

The propagation to the output is computed in a similar way, using the s_j values obtained for the hidden layer. That is, we first accummulate the inputs of the output layer using the weights and the bias as follows

$$acc^o_k = \sum_{j=1}^{L} w^o_{kj} s_j + \theta^o_k,$$

and then applying the activation function to this value

$$o_k = f^o_k \left(\sum_{j=1}^{L} w^o_{kj} s_j + \theta^o_k \right). \tag{2}$$

If the expected output is known for example x, we can compute the error of the network for this example. Recall that we have expressed the output as $y = (y_1, \ldots, y_M)$. Using the sum of squares of the errors (also known as the residual sum of squares) of each output neuron, we have that the overall error for this example is:

$$E = \frac{1}{2} \sum_{k=1}^{M} (y_k - o_k)^2.$$

We use $1/2$ in the expression for the error because this will lead later into a simpler expression for the weight updatings. Note that the multiplication of the error by the constant $1/2$ does not change its meaning.

With the computation of the output o_k for each output neuron, and the computation of the error, we have completed the propagation step. Now, we will describe the adaptation. The backpropagation algorithm follows the gradient descent.

Recall that for minimization functions, the gradient descent moves in the direction of the negative of the gradient. That is, given a point x_i, the next step will be

$$x_{i+1} = x_i - \eta \nabla NN(x).$$

Let us compute the gradient of the neural network. We start with the gradient of the weights that connect the hidden layer with the output. That is, the weights w^o_{kj}. To do so, we derivate the error E with respect to w^o_{kj} taking into account Equation 2 for o_k. As the weight w^o_{kj} appears in o_k but only there (i.e., not in o_r for $r \neq k$), the derivative of E with respect to w^o_{kj} is the derivative of $(1/2)(y_k - o_k)^2$. Therefore,

$$\frac{\partial E}{\partial w^o_{kj}} = \frac{\partial \frac{1}{2} \sum_{\kappa=1}^{M} (y_\kappa - o_\kappa)^2}{\partial w^o_{kj}} \tag{3}$$

$$= \frac{\partial \frac{1}{2} (y_k - o_k)^2}{\partial w^o_{kj}} \tag{4}$$

$$= -(y_k - o_k)(f^o_k)' \left(\sum_{j=1}^{L} w^o_{kj} s_j + \theta^o_k \right) \cdot s_j \tag{5}$$

Therefore, we define our increment of w^o_{kj} according to the gradient as

$$\Delta w^o_{kj} = \eta (y_k - o_k)(f^o_k)' \left(\sum_{j=1}^{L} w^o_{kj} s_j + \theta^o_k \right) \cdot s_j.$$

Now, let us turn into the weights between the input layer and the hidden layer. That is, the weights w^h_{ji}. The first step is to consider an expression for the error where it is clear the dependency with the weights under consideration. To do so, we first replace o_k by the corresponding expression. In this way we obtain,

$$E = \frac{1}{2} \sum_{k=1}^{M} (y_k - o_k)^2 \tag{6}$$

$$= \frac{1}{2} \sum_{k=1}^{M} \left(y_k - f^o_k \left(\sum_{j=1}^{L} w^o_{kj} s_j + \theta^o_k \right) \right)^2 \tag{7}$$

$$\tag{8}$$

As s_j is the output of the hidden layer, the dependency on the weights w^h_{ji} will be through s_j. Therefore, we have that the derivative of the error with respect to w^h_{ji} is in terms of the derivative of s_j with respect to these weights. Using the expression above for E, we obtain the following expression:

$$\frac{\partial E}{\partial w^h_{ji}} = -\sum_{k=1}^{M}(y_k - o_k)(f^o_k)'\left(\sum_{j=1}^{L} w^o_{kj}s_j + \theta^o_k\right) w^o_{kj}\frac{\partial s_j}{\partial w^h_{ji}}. \tag{9}$$

Now, let us recall Equation 1 that

$$s_j = f^h_j(acc^h_j) = f^h_j\left(\sum_{i=1}^{N} w^h_{ji}x_i + \theta^h_j\right)$$

and, therefore,

$$\frac{\partial s_j}{\partial w^h_{ji}} = (f^h_j)'\left(\sum_{i=1}^{N} w^h_{ji}x_i + \theta^h_j\right) x_i.$$

Therefore,

$$\frac{\partial E}{\partial w^h_{ji}} = -\sum_{k=1}^{M}(y_k - o_k)(f^o_k)'\left(\sum_{j=1}^{L} w^o_{kj}s_j + \theta^o_k\right) w^o_{kj}(f^h_j)'\left(\sum_{i=1}^{N} w^h_{ji}x_i + \theta^h_j\right) x_i \tag{10}$$

Using, acc^h_j and acc^o_k as defined above

$$acc^h_j = \sum_{i=1}^{N} w^h_{ji}x_i + \theta^h_j$$

$$acc^o_k = \sum_{j=1}^{L} w^o_{kj}s_j + \theta^o_k,$$

and taking into account that neither $(f^h_j)'(acc^h_j)$ nor x_i depend on k so they can be put outside the summatory, we have that we can rewrite the above expression as:

$$\frac{\partial E}{\partial w^h_{ji}} = -(f^h_j)'\left(acc^h_j\right) x_i \sum_{k=1}^{M}(y_k - o_k)(f^o_k)'\left(acc^o_k\right) w^o_{kj}. \tag{11}$$

Therefore,

$$\Delta w^h_{ji} = \eta(f^h_j)'\left(acc^h_j\right) x_i \sum_{k=1}^{M}(y_k - o_k)(f^o_k)'\left(acc^o_k\right) w^o_{kj}.$$

Using the expressions Δw^o_{kj} and Δw^h_{ji} we have that the adaptation (learning) step for a given example (x, y) corresponds to the following:

– For all i, j compute

$$w_{ji}^h(t+1) = w_{ji}^h(t) + \eta(f_j^h)' \left(acc_j^h\right) x_i \sum_{k=1}^{M} (y_k - o_k)(f_k^o)' \left(acc_k^o\right) w_{kj}^o$$

– For all k, j compute

$$w_{kj}^o(t+1) = w_{kj}^o(t) + \eta(y_k - o_k)(f_k^o)' \left(\sum_{j=1}^{L} w_{kj}^o s_j + \theta_k^o \right) \cdot s_j$$

In these expressions, η is the learning rate. It is usually a small value, near e.g. 0.15.

Deep learning Larger networks, with additional hidden layers (e.g., from 5 to 10). Related topics: deep neural networks and deep belief networks.

2.7 Support vector machines

Linear support vector machines (SVM) classify data in two classes by means of a hyperplane in the space of the data. The learning algorithm is to find an appropriate location for this hyperplane.

Non-linear support vector machines transform the original data translating them in a high-dimensional space, and then locates a hyperplane in this high-dimensional space. This results into a better classification because the hyperplane in this high-dimensional space corresponds to a non-linear hypersurface in the original space.

3 Unsupervised machine learning

3.1 Clustering

Clustering algorithms build structure for a set of data. They tend to put in the same cluster those objects that are similar and leave in different clusters those objects that are different. There are different ways to represent the structure of the data. The most usual case is one of the following ones.

– **Disjoint categories.** An object can belong only to a single category. This is the structure built by those algorithms that build a partition of the domain. All objects present in the training set are usually assigned to one of the clusters. Nevertheless, this is not always so for the objects not present in the training set. This is the case when logical descriptions are built for describing the clusters. An example of algorithm that leads to this type of structure is k-means.

– **Categories with overlapping.** Objects can belong to different memberships. This is the case of having fuzzy clusters. Fuzzy c-means is an algorithm that lead to fuzzy clusters. Fuzzy cluster is related to fuzzy sets, as the overlapping is represented by means of membership functions. A discussion on the origins of fuzzy clustering by Ruspini and Bezdek can be found in [14].
– **Categories with a hierarchical structure.** The relationship between categories is expressed in terms of a tree. We call dendrogram the tree-like graph to represent the hierarchical structure of clusters. Given a category we may have sub-categories and super-categories. All objects belonging to one category will also belong to all super-categories (categories found in the path to the root of the tree). Figure 9 is a dendrogram for the Iris dataset constructed using a hierarchical clustering algorithm.

Cluster description There are different ways to represent the clusters obtained by a clustering algorithm. Some of them are the following.

– Intensional representation. In this case, we have that categories are represented in terms of their properties. For example, using centroids or logical expressions.
  • Centroids.
  • Logical expressions
– Extensional representation. Categories are expressed by means of a exhaustive list of the objects that define them. The term lazy learning is used for those methods that use an extensional representation. This is the case, for example, of k-nearest neighbors, and of case-based reasoning.

Types of algorithms

– Partitive methods
– Agglomerative methods

– Combinatorial algorithms. Directly working on data. No assumption of any underlying probability distribution.
– Mixture modeling algorithms. Assumption on an underlying probability density function. Look for the parameters of the model. E.g., two Gaussian distributions are fitted to a set of points.
– Mode seeker algorithms. Assumption on an underlying probability density function. Nonparametric perspective.

Partitions and fuzzy partitions Some partitive clustering algorithms return the partition that minimize an objective function (OF). E.g.,

$$\Pi = \arg\min_{s \in S} OF(s).$$

This is the case of k-means and fuzzy c-means. The first for finding crisp partitions and the second for finding fuzzy partitions.

Algorithm 1 Clustering algorithm: c-means.

Step 1: Define an initial partition and compute its centroid P.
Step 2: Solve $min_{\chi \in M_c} OF(\chi, P)$ as follows:
- For all $x \in X$, define $k_0 = \arg\min_i ||A(x) - p_i||^2$
- $\chi_{k_0}(x) = 1$
- $\chi_j(x) = 0$ for all $j \in \{1, \ldots, c\}$ s.t. $j \neq k_0$

Step 3: Solve $min_P OF(\chi, P)$ as follows:
- for all $k \in \{1, \ldots, c\}$, define $p_k = \frac{\sum_{x \in X} \chi_k(x) A(x)}{\sum_{x \in X} \chi_k(x_i)}$

Step 4: Repeat steps 2 and 3 till convergence

k-means This is a partitive clustering algorithm that finds a given number of partitions. The algorithm looks for a partition that minimizes an expression, and each partition has a cluster center (its centroid). The expression computes the distance between the records and the cluster centers. So, the goal is to find a partition and centroids so that they are at a minimial distance of all points.

$$\begin{aligned} &\text{Minimize} \\ &\qquad OF(\chi, P) = \textstyle\sum_{k=1}^{c} \sum_{x \in X} \chi_k(x) ||A(x) - p_k||^2 \\ &\text{subject to} \\ &\qquad \chi \in M_c = \{\chi_k(x) | \chi_k(x) \in \{0, 1\}, \textstyle\sum_{k=1}^{c} \chi_k(x) = 1 \text{ for all } x \in X\} \end{aligned} \tag{12}$$

This cannot be solved optimally in an analytical form. Instead it is solved using the following algorithm.

Step 1: Define an initial partition and compute its centroid P.
Step 2: Solve $min_{\chi \in M_c} OF(\chi, P)$
Step 3: Solve $min_P OF(\chi, P)$
Step 4: Repeat steps 2 and 3 till convergence

For step 2 we use:

- $k_0 = \arg\min_i ||A(x) - p_i||^2$
- $\chi_{k_0}(x) = 1$
- $\chi_j(x) = 0$ for all $j \neq k_0$

For step 3 we use:

$$p_k = \frac{\sum_{x \in X} \chi_k(x) A(x)}{\sum_{x \in X} \chi_k(x_i)} \tag{13}$$

Algorithm 1 summarizes all the steps.

Algorithm 2 Fuzzy c-means

Step 1: Generate initial P

Step 2: Solve $min_{\mu \in M} OF_{FCM}(\mu, P)$ by computing for all $i \in \{1, \ldots, c\}$ and $x \in X$:

$$\mu_i(x) = \Big(\sum_{j=1}^{c} \Big(\frac{||x - p_i||^2}{||x - p_j||^2} \Big)^{\frac{1}{m-1}} \Big)^{-1}$$

Step 3: Solve $min_P OF_{FCM}(\mu, P)$ by computing for all $i \in \{1, \ldots, c\}$:

$$p_i = \frac{\sum_{x \in X} (\mu_i(x))^m x}{\sum_{x \in X} (\mu_i(x))^m}$$

Step 4: If the solution does not converge, go to Step 2; otherwise, stop.

Fuzzy c-means Fuzzy c-means is similar to k-means, but in this case partitions are fuzzy.

Definition 1. *[13] Let X be a reference set. Then, a set of membership functions $\mathcal{M} = \{\mu_1, \ldots, \mu_c\}$ is a fuzzy partition of X if for all $x \in X$ we have*

$$\sum_{i=1}^{c} \mu_i(x) = 1$$

The definition of the problem is:

$$\begin{array}{ll} \text{Minimize} & OF_{FCM}(\mu, P) = \{\sum_{i=1}^{c} \sum_{x \in X} (\mu_i(x))^m ||x - p_i||^2\} \\ \text{subject to} & \mu_i(x) \in [0,1] \text{ for all } i \in \{1, \ldots, c\} \text{ and } x \in X \\ & \sum_{i=1}^{c} \mu_i(x) = 1 \text{ for all } x \in X. \end{array} \tag{14}$$

This is solved with Algorithm 2, with the expressions below for cluster centers p_i and membership functions μ_i.

$$p_i = \frac{\sum_{x \in X} \mu_i(x) x}{\sum_{x \in X} \mu_i(x)} \tag{15}$$

$$\mu_i(x) = \frac{e^{-\lambda ||x - p_i||^2}}{\sum_{j=1}^{c} e^{-\lambda ||x - p_j||^2}} \tag{16}$$

The last expression for $\mu_i(x)$ can be rewritten as follows.

$$\mu_i(x) = \frac{1}{1 + \frac{\sum_{j \neq i}^{c} e^{-\lambda ||x - p_j||^2}}{e^{-\lambda ||x - p_i||^2}}} \tag{17}$$

Validity measures

– **Dunn's Index**. It was proposed in [3] and it is based on cluster compactness and cluster separatedness. Cluster compactedness for the ith cluster is denoted by $cc(\chi_i)$ and cluster separatedness for the ith and jth cluster is denoted by $cs(\chi_i, \chi_j)$. Then

$$DI = \frac{\min_{1 \leq i,j \leq c; i \neq j} cs(G_i, G_j)}{\max_{i \leq l \leq c} cc(G_l)}$$

where $cc(\chi_i) = \max_{x,y \in \chi_i} d(x, y)$, and $cs(\chi_i, \chi_j) = \min_{x \in \chi_i, y \in \chi_j} d(x, y)$.

3.2 Self-organizing maps

4 Reinforcement learning

The key features of reinforcement learning include (see [16] for details)

– Trade-off between exploration and exploitation.
– Consideration of the whole problem of a goal-directed agent interacting with an uncertain environment.
– Reinforcement learning is near to optimal control theory and stochastic approximation.

5 Others

Multidimensional scaling. See e.g. [1].

6 Discussion

Why some methods are not appropriate. `http://www.analyticbridge.com/profiles/blogs/the-8-worst-p`

References

1. Borg, I., Groenen, P. J. F. (2005) Modern multidimensional scaling: theory and applications, 2nd edition, Springer
2. Cybenko, G. (1989) Approximation by superpositions of a sigmoidal function, Mathematical control signals systems 2 303-314.
3. Dunn, C. (1974) Well separated clusters and optimal fuzzy partitions, J. of Cybernetics 4 95-104.
4. Funuhashi, K. (1989) On the approximate realization of continuous mappings by neural networks, Neural networks 2 183-192.
5. Hornik, K., Stinchcombe, M., White, H. (1989) Multilayer feedforward networks are universal approximators, Neural networks 2 359-366.

6. Hornik, K., Stinchcombe, M., White, H. (1990) Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks, Neural networks 3 551-560.
7. Hothorn, T., Zeileis, A. (2016) partykit: A Toolkit for Recursive Partytioning. `https://cran.r-project.org/web/packages/partykit/partykit.pdf`
8. Keener, R. W. (2010) Theoretical statistics, Springer. p. 130.
9. Martin, J. K. (1997) An exact probability metric for decision tree splitting and stopping, Machine Learning 28 257-291.
10. Mitchell, T. M., Utgoff, P. E., Banerji, R. (1983) Learning by experimentation: acquiring and refining problem-solving heuristics, in R. S. Michalski, J. G. Carbonell, T. M. Mitchell (eds.) Machine learning: an aritificial intelligence approach, Morgan Kaufmann 163-190.
11. Muggleton, S., Feng, C. (1992) Efficient induction of logic programs, in S. Muggleton (ed.) Inductive logic programming, Academic-Press, 281-298.
12. Murthy, S. K. (1998) Automatic construction of decision trees from data: A multidisciplinary survey, Data Mining and Knowledge Discovery 2 345-389.
13. Ruspini, E. H. (1969) A new approach to clustering, Inform. Control. 15 22-32.
14. Seising, R. (2014) On the history of fuzzy clustering: An interview with Jim Bezdek and Enrique Ruspini, Archives for the Philosophy and History of Softcomputing 2:1 1-14.
15. Shih, Y.-S. (2004) A note on split selection bias in classification trees, Computational statistics and data analysis 45 457-466.
16. Sutton, R. S., Barto, A. G. (2012) Reinforcement learning: an introduction, The MIT Press.
17. Torra, V., Narukawa, Y. (2007) Modeling decisions: information fusion and aggregation operators, Springer.

Entropy is a measure of information introduced by Shannon. Given a probability distribution $p = (p_1, p_2, \ldots, p_n)$ it is defined as

$$H(p) = \sum_{i=1}^{n} -p_i \log_2 p_i$$

with $0log0 = 0$. To illustrate this definition we consider the following two extreme cases.

- Note that if we have a fair coin $p = (p_h, p_t) = (1/2, 1/2)$ where p_h is the probability of heads and p_t is the probability of tails, then

  $$H(p) = -(1/2)\log_2(1/2) - (1/2)\log_2(1/2) = -\log_2(1/2) = 1.$$

  This situation corresponds to maximum uncertainty (as both heads and tails have the same probability) and, thus, informing about the result gives us the maximum information.
- Note that if we have an unfair coin $p = (p_h, p_t) = (1, 0)$, that is, it always flips to heads, then

  $$H(p) = -1\log_2 1 - 0\log_2 0 = 0.$$

  This situation corresponds to minimum uncertainty (we are sure that it will be heads) and, thus, informing about the result gives us no information at all.

In general, when we have n options and $p = (p_1, \ldots, p_n)$, we have maximum entropy (corresponding to the case of maximum uncertainty and, thus, when we have maximum information if we know the outcome) when

$$p = (p_1, \ldots, p_n) = (1/n, \ldots, 1/n).$$

In this case, the entropy is

$$H(p) = \sum_{i=1}^{n} -(1/n)\log_2(1/n) = -\log_2(1/n).$$

On the contrary, we have minimum entropy when we have $p_i = 1$ for one of the i in $\{1, \ldots, n\}$.

Fig. 5. Definition of entropy.

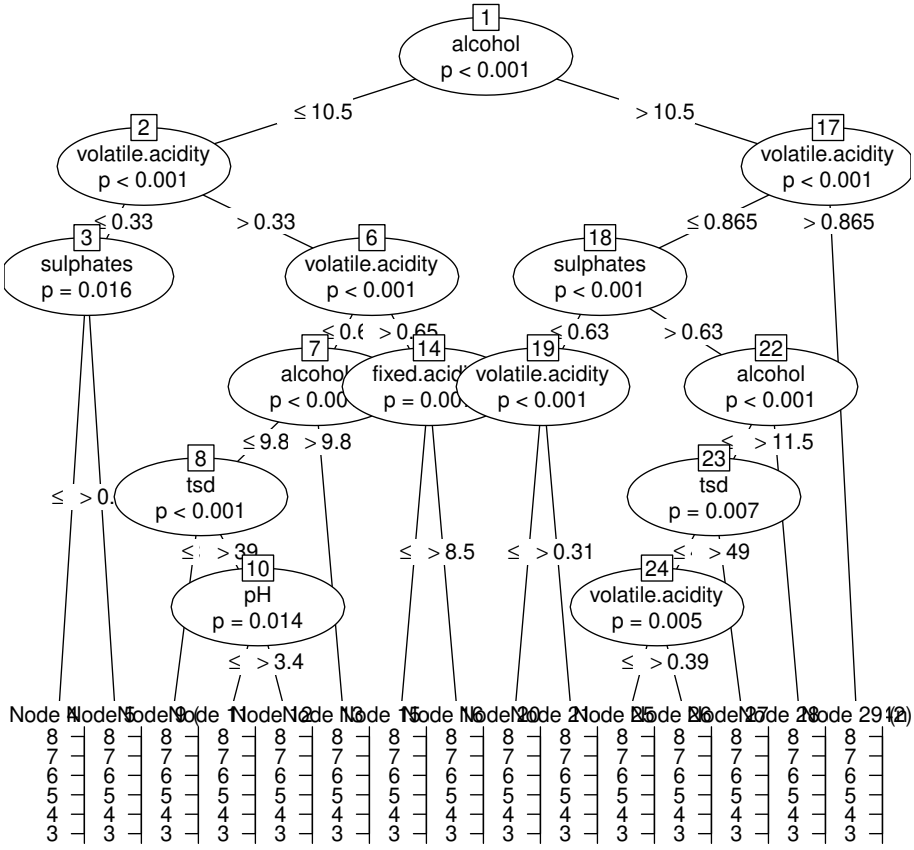

Fig. 6. Decision tree.

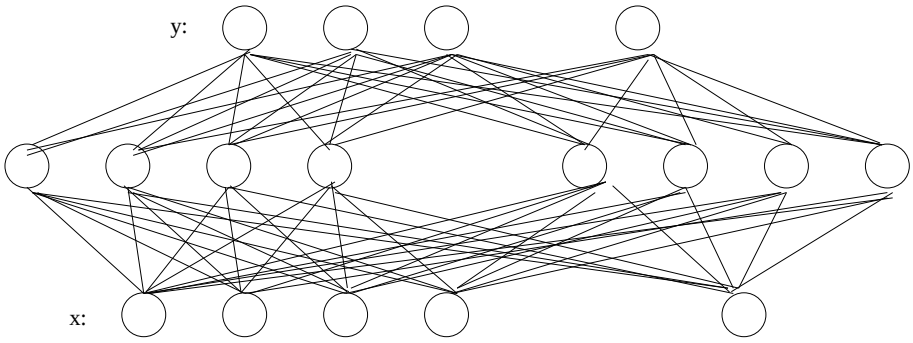

Fig. 7. Neural network.

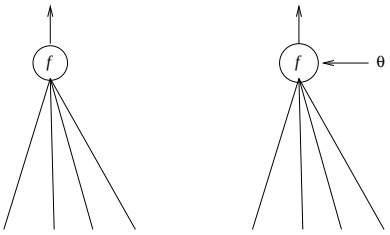

Fig. 8. Neuron with four inputs: (left) without bias, (right) with bias.

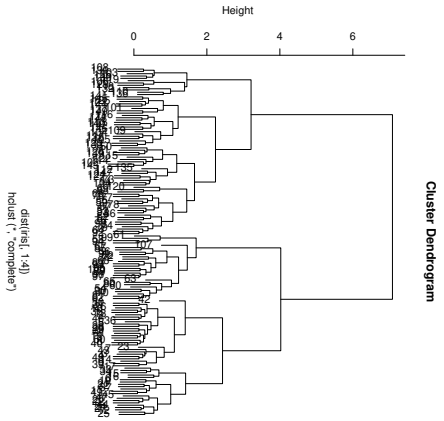

Fig. 9. Dendrogram corresponding to the Iris dataset using the function `hclust` of R.