Machine learning

An agent is **learning** if she improves her performance on future tasks after making observations of her environment and previous achievements.

This can raise two sorts of doubts.

First, the human intelligence seems inherently possessing the ability to learn. The human reasoning processes appear inseparable from the learning processes. We would not consider intelligent a person who hasn't learned from her experience, at least in the simplest ways. So why is it separate in artificial intelligence?

There is no clear answer to this. Most developed and widely used artificial intelligence paradigms in their basic form perform reasoning without learning. The ability to learn has to be added.

Another doubt about machine learning might be: if it is not inherently obvious, or obligatory, then why is it needed, or is it really? If we can program reasoning processes, and are able to tune them to perfection, then perhaps we can obtain an artificial intelligence agent, who does not have to learn, or cannot learn anything more.

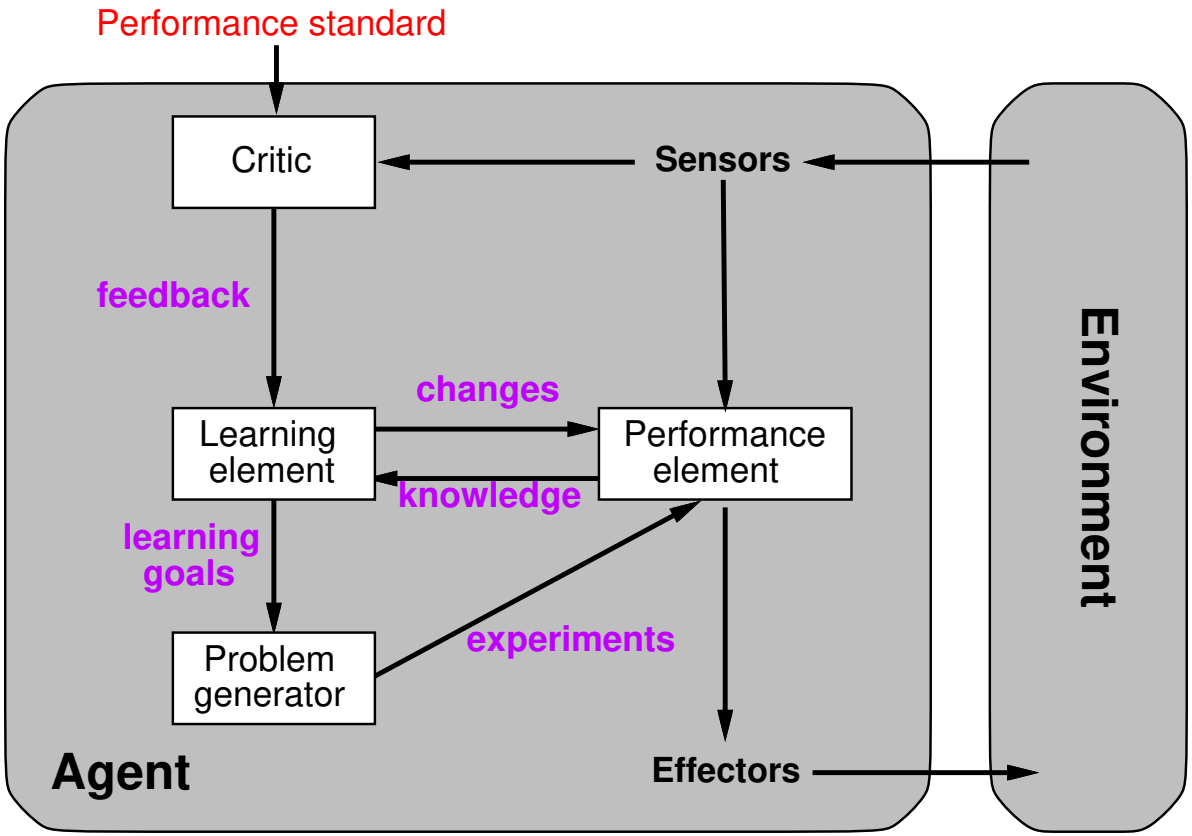
There is a good answer to this question, and there are several good reasons.

First, the designers of AI cannot anticipate all possible situations that the agent might find herself in. For example, a robot navigating a maze must learn the layout of each new maze she encounters.

Second, the designers cannot anticipate all changes over time. A program designed to predict tomorrow's stock market prices must learn to adapt when conditions change in an unpredictable way.

Third, sometimes human programmers have no idea how to program a solution themselves. For example, most people are good at recognizing the faces of their acquaintances, but even the best programmers are unable to program the computer to do this, except by using learning algorithms.

The general model



The elements of a learning agent (from mistakes):

performance element — makes decisions about acting, in a sense it is the whole intelligent agent sans learning

learning element — watches the effects of the agent's actions, and makes corrections in its actions algorithm

critic — evaluates the quality of the agent's actions, makes use of an external performance reference

problem generator — necessary if the agent is willing to experiment in order to discover new possibilities and action techniques; normally, the performance element would only utilize its abilities to make optimal (in its understanding) decisions, and would never discover anything new

Obviously, the construction of the learning element is heavily dependent upon the structure of the performance element, such as the knowledge representation scheme (eg. state space, rules, predicate logic, probabilistic networks, or other).

General learning schemes

"routine" learning

memorizing and generalizing solutions worked out by the agent in the course of its actions, in order to apply similar solutions more efficiently to similar problems in the future

inductive learning

the agent has the ability to watch the phenomenon for some time, eg. the values of some parameters in a collections of examples, and to build a model of this phenomenon, which it can later use to answer questions regarding this phenomenon (including its own questions, posed in the course of working on problems assigned to it)

"creative" learning

learning scheme not limited to modeling some unknown function, but making it possible to developing new techniques, algorithms, representation schemes, etc., for example, learning mathematics to the point of formulating new hypotheses, constructing its proofs, and also creating new concepts, only loosely related to those already known

“Routine” learning

Example: action planning

An agent may memorize the operator sequence leading to the solution, hoping that in case it is ever given the same problem to solve it could reuse the plan.

It can further generalize the plan, for example, substituting parameters for specific values, and as the result obtaining an **action plan scheme**, which is a kind of a macro-operator with easy to determine applicability conditions and effects.

Specific algorithms exist for such generalization of operator sequences. More sophisticated algorithms may introduce into the generalized plan such elements as conditions and loops, effectively turning an action plan into a small program.

Inductive learning

supervised (teacher) learning

The agent receives the information in the form of input-output pairs, and learns the mapping from inputs to outputs.

For example, the agent can obtain from the teacher instructions for her actions for the current percepts, eg. green light-move, obstacle-stop, etc. In another situation the agent can learn recognizing patterns in received input data, eg. recognizing traffic signs in images obtained from a camera. In yet another situation, the agent may learn the effects of her own actions in different conditions, eg. braking distance under various road conditions, weather, speed, and the braking force.

After receiving each new training pair (x, y) the agent can both improve her model of the environment, and verify it, by first trying to predict y by herself.

reinforcement learning

The agent receives from the teacher not the direct pairs of values (x, y), but only an evaluation of its performance in the form of an occasional reward (or punishment), called the **reinforcement**. The agent does not know which of her actions mostly influenced the reward received, or whether and how she should update her model of the environment. She must make such decisions based on her observations and comparing the reinforcements received.

For example, this is the situation of an agent learning to play chess entirely on the basis of the final game result.

unsupervised learning

In this case the agent does not receive any values of the outputs, or any hints. She may only observe the sequence of the **percepts** (observable parameter values) and learn the patterns contained therein so she can predict future values in the sequence.

This type of learning does not apply to planning the agent's action, since she cannot learn what she should do this way, unless she had a state utility function (in which case it would be the case of reinforcement learning).

Induction learning — the general model

Simplest form: learning an unknown function $f(x)$ from a series of pairs $\langle x, f(x)\rangle$.
Example of a set of training pairs: $\{\langle$  $, 0\rangle, \langle$  $, 1\rangle\}$.

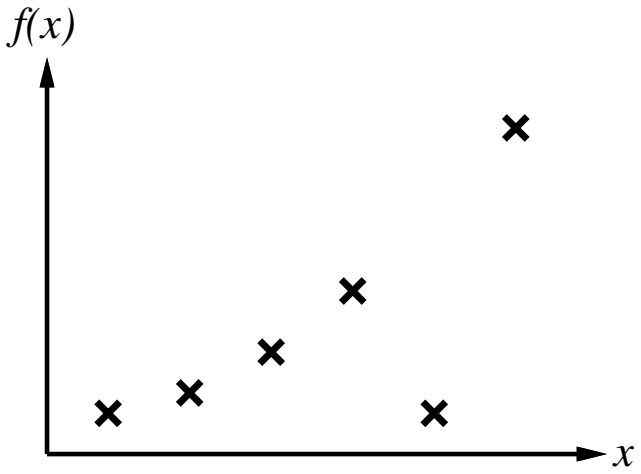
The problem: from the series of training examples find a hypothesis $h \in \mathcal{H}$ ($\mathcal{H}$ is the **hypothesis space**) such, that $h \approx f$, ie. the hypothesis h approximates the unknown function f according to some criterion. In the simplest case the criterion may be minimizing the number of elements of the training set, for which $h(x) \neq f(x)$.

In practice, the goal of learning is the generalization. We want not just to be able to correctly classify the elements of the training set, but to capture the principle according to which they were classified. The ability to learn in this sense depends on:

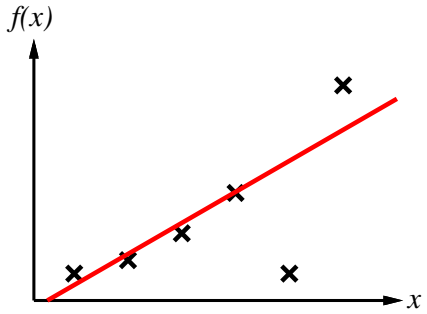
- the training set, both its cardinality and the specific choice of its members,
- the machine learning algorithm in use,
- the hypothesis space $\mathcal{H}$ (which is a derivative of the ML algorithm); if it is too modest then efficient learning might not be possible.

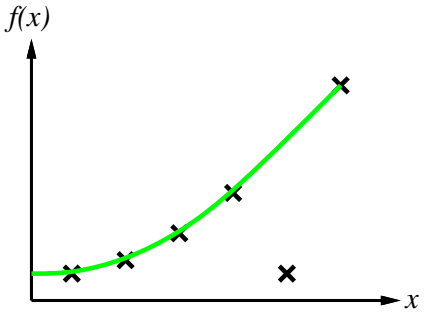
Induction learning — example

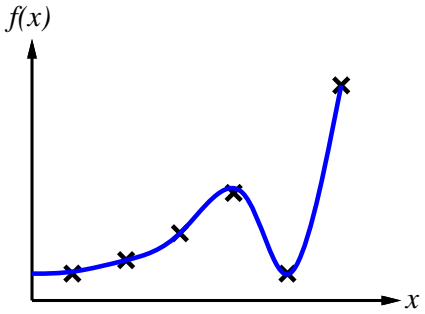
For example, fitting an unknown curve:

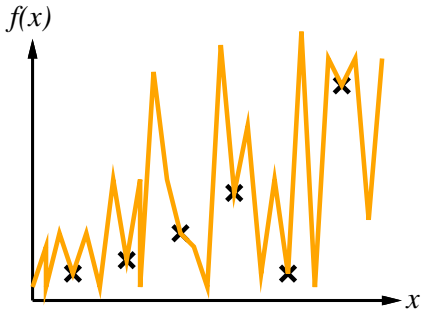


We may consider the following hypotheses:









The Ockham's razor principle: from among the hypotheses, which explain a given phenomenon equally well, choose the simplest one — the one which makes the minimum additional assumptions.

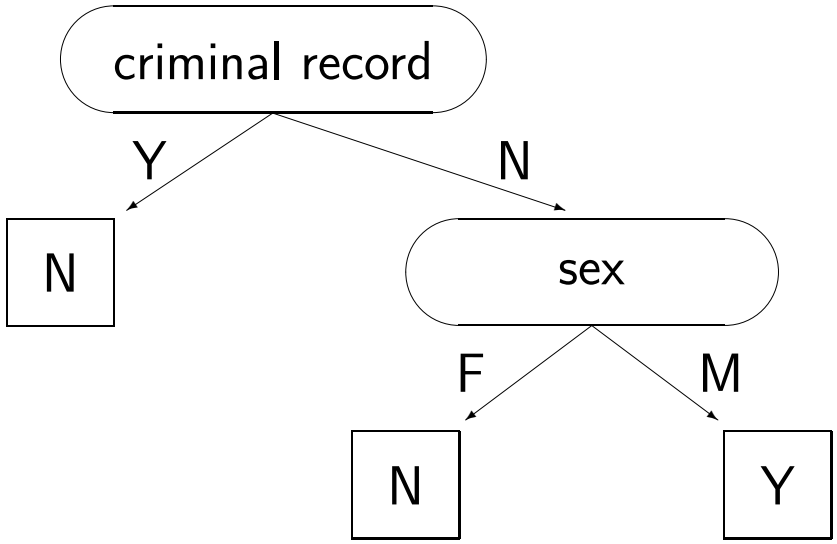
Decision trees — introduction

We want to automate making preliminary decisions about approving credit applications by a bank, to make it easier for the bank officers to make the final decisions. We have the history of the past credit applications — can we learn from them?

f-n	l-n	age	sex	income	educ.	empl.	crim.	...	approval
-	-	30	M	18,000	univ.	2	N	...	Y
-	-	42	F	12,000	univ.	8	N	...	N
-	-	46	M	58,000	univ.	14	N	...	Y
-	-	55	M	22,500	coll.	6	Y	...	N
-	-	35	M	36,000	univ.	4	N	...	Y
-	-	22	F	30,000	univ.	< 1	N	...	N
-	-	28	M	25,000	elem.	8	N	...	Y

We have a set of records with many attributes, we need to find an algorithm to determine the value of a designated attribute (approval) based on the values of the other attributes.

Such an algorithm can be constructed using the concept of a **decision tree**.



Note:

- Such a tree can always be constructed, unless there exist two vectors with identical attribute values, with differing credit approval decisions.
- The fact that such a tree could be constructed does not guarantee that it will work correctly, ie. correctly make the approval decisions. There may be various alternative such trees, and to select from among them, additional testing data could be necessary.
- Some attributes should be discarded (like the first or last name).
- Some attributes has specific value sets (like income, or the length of employment), so what should be done with them? Value ranges perhaps?

Decision trees — principles

- A decision tree can express a binary function represented by any set of examples: positive and/or negative, as long as it is consistent.

- A trivial construction of a decision tree: subsequent levels correspond to the attributes, and the tree branches growing at each levels correspond to the specific values of these attributes. The leaves would correspond to the specific examples and contain the answers.

- Such construction however only memorizes the set of cases: if an example from the training set occurs again then it will be correctly found on the tree. However, if an example not from the training set was given, then the tree would not contain the answer.

- It would be useful to have a method to construct a decision tree capable of **generalizing** the observations, eg. by grouping the examples, like the credit approval tree above.

- We need to note that, in general there exists very many binary functions (2^{2^n} for n attributes), and not all can be generalized by grouping.

  For example, the parity function (checking whether the number of cases with a specific attribute value is even), or the majority function, can only be expressed with a full, exponentially-sized decision tree, while both can be expressed using simple formulas.

- So it makes sense to employ the decision trees when they can represent some concept (given by the set of examples) in a compact, minimal way.

  The desired technique for building the decision trees can then construct a **minimal** tree correctly classifying the concept.

- Unfortunately, building the strictly minimal decision tree it hard — it requires building all the trees and choosing one.

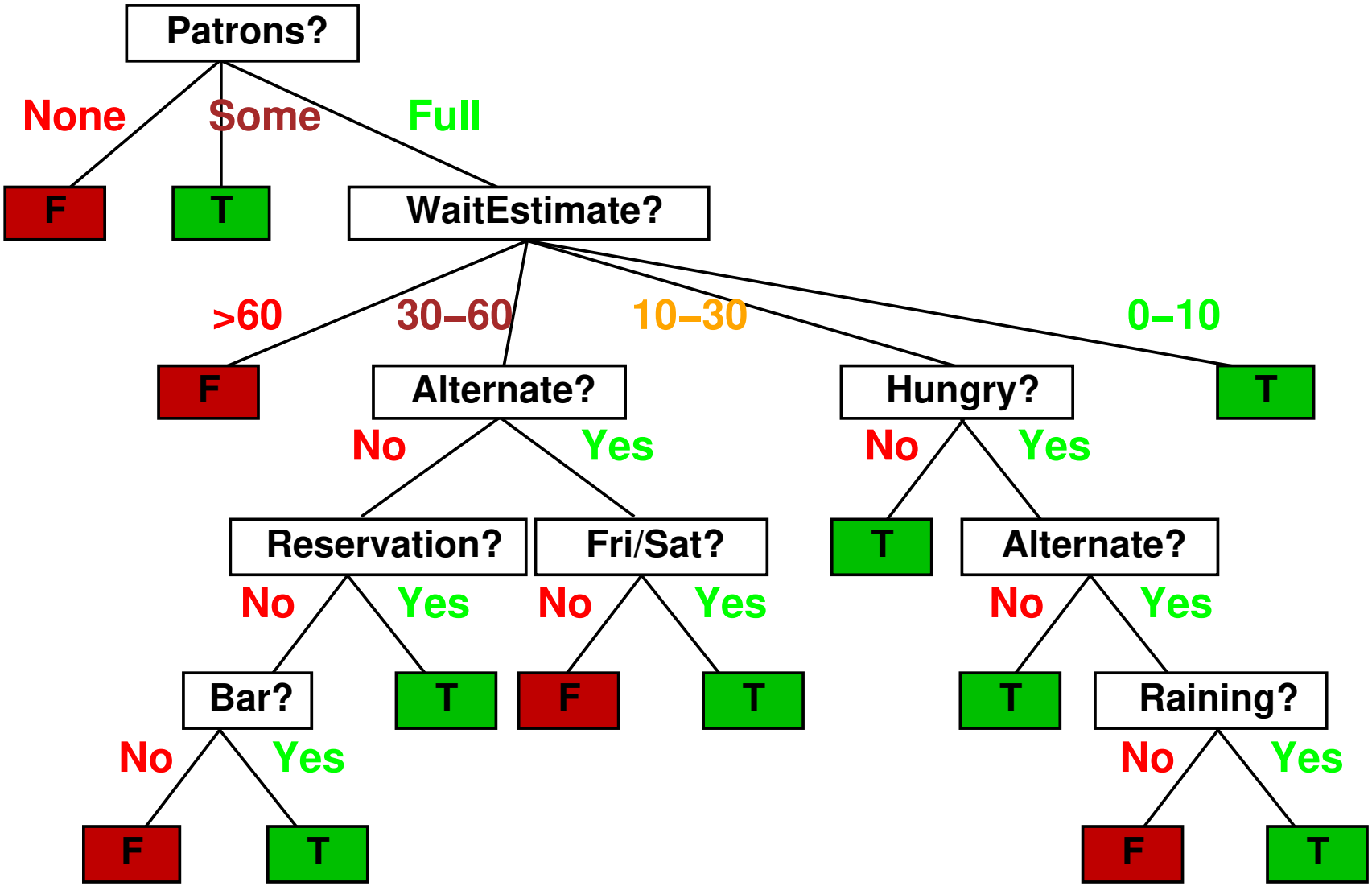
- Instead, a heuristic procedure can be used which selects the most important attributes in turn.

Example: whether to wait in a restaurant

Example	Attributes										Target
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
X_1	*T*	*F*	*F*	*T*	*Some*	*$$$*	*F*	*T*	*French*	*0–10*	*T*
X_2	*T*	*F*	*F*	*T*	*Full*	*$*	*F*	*F*	*Thai*	*30–60*	*F*
X_3	*F*	*T*	*F*	*F*	*Some*	*$*	*F*	*F*	*Burger*	*0–10*	*T*
X_4	*T*	*F*	*T*	*T*	*Full*	*$*	*F*	*F*	*Thai*	*10–30*	*T*
X_5	*T*	*F*	*T*	*F*	*Full*	*$$$*	*F*	*T*	*French*	*>60*	*F*
X_6	*F*	*T*	*F*	*T*	*Some*	*$$*	*T*	*T*	*Italian*	*0–10*	*T*
X_7	*F*	*T*	*F*	*F*	*None*	*$*	*T*	*F*	*Burger*	*0–10*	*F*
X_8	*F*	*F*	*F*	*T*	*Some*	*$$*	*T*	*T*	*Thai*	*0–10*	*T*
X_9	*F*	*T*	*T*	*F*	*Full*	*$*	*T*	*F*	*Burger*	*>60*	*F*
X_{10}	*T*	*T*	*T*	*T*	*Full*	*$$$*	*F*	*T*	*Italian*	*10–30*	*F*
X_{11}	*F*	*F*	*F*	*F*	*None*	*$*	*F*	*F*	*Thai*	*0–10*	*F*
X_{12}	*T*	*T*	*T*	*T*	*Full*	*$*	*F*	*F*	*Burger*	*30–60*	*T*

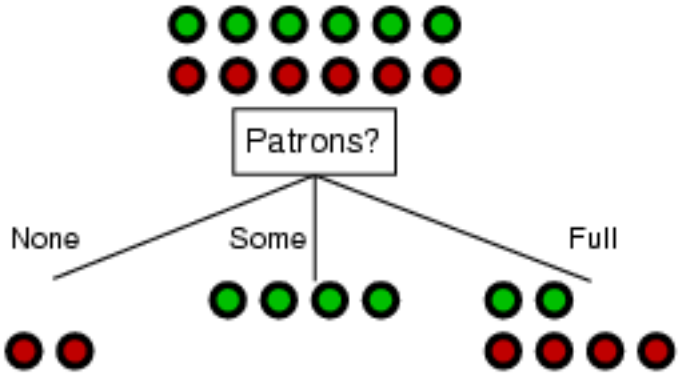
Alternate — whether there is a suitable alternative restaurant nearby
Bar — whether the restaurant has a comfortable bar area to wait in
Fri/Sat — true on Fridays and Saturdays
Hungry — whether we are hungry
Patrons — how many people are in the restaurant (None, Some, or Full)
Price — the restaurant's price range ($, $$, $$$)
Raining — whether it is raining outside
Reservation — whether we made a reservation
Type — the kind of restaurant (French, Italian, Thai, or burger)
WaitEstimate — the wait estimated by the host (0-10/10-30/30-60/>60m)

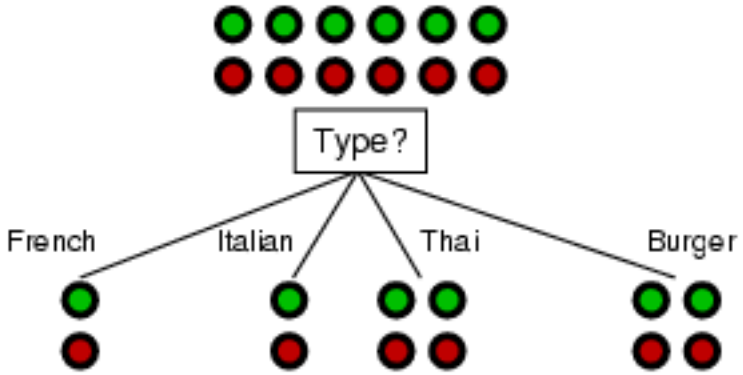
Example: an „ad-hoc" tree



Example: a systematic approach

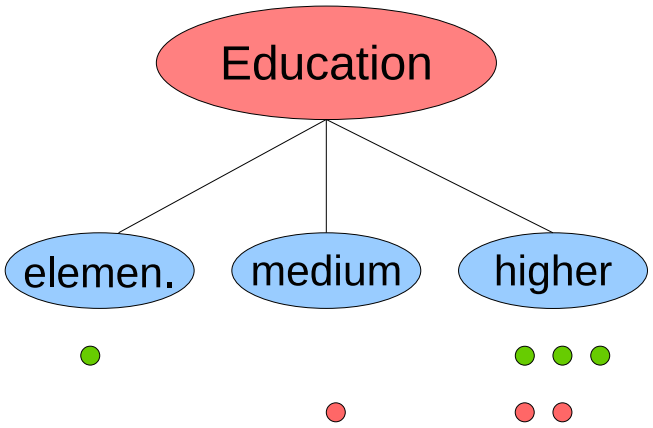
We collect the set of training examples, and "fit" different attributes:

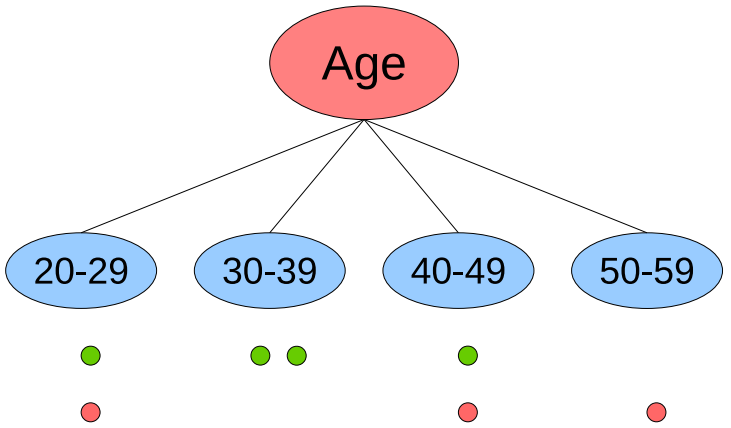




Which attribute is a better candidate for the construction of a decision tree? It seems that "Patrons". In two cases it clearly determines the correct result, and in the third case, although the result is not unambiguous, still one choice has a majority.

However, in the general case the choice will not necessarily be so obvious as above:





Information contents

How, then, can the idea of partitioning a dataset into subsets be formalized?

The key concept is the **information**. It is necessary to determine the answers to the questions. If initially we do not know the answer to some question, then by learning the answer we gain information.

The quantity used in the information theory to measure this **information gain** is the **entropy**. The unit used to measure the entropy is the bit. If a random variable has two possible values (eg. the outcome of a coin flip) with a uniform probability distribution $\langle \frac{1}{2}, \frac{1}{2} \rangle$, the entropy of this variable, equal to the information gain from learning its value, is 1 bit.

However, if the coin was not fair, eg. would come up heads 99% of time, the entropy would be less. In the extreme, the outcome of flipping a loaded coin, which always comes up heads, has the entropy of zero (bits), because it does not bring any information.

The entropy

The entropy of a random variable V with the value set v_k is defined as:

$$H(V) = \sum_k P(v_k) \log_2 \frac{1}{P(v_k)} = -\sum_k P(v_k) \log_2 P(v_k)$$

In a random selection from the set of n^+ positive and n^- negative samples, with a uniform probability distribution, we get the entropy:

$$H(\langle \frac{n^+}{n^+ + n^-}, \frac{n^-}{n^+ + n^-} \rangle) = -\frac{n^+}{n^+ + n^-} \log_2 \frac{n^+}{n^+ + n^-} - \frac{n^-}{n^+ + n^-} \log_2 \frac{n^-}{n^+ + n^-}$$

For the 12 samples of the restaurant example we have $n^+ = n^- = 6$ and the entropy:

$$H(\langle \frac{6}{12}, \frac{6}{12} \rangle) = -\frac{1}{2}(-1) - \frac{1}{2}(-1) = 1$$

The entropy of an entirely homogenous set of only negative samples $n^- = 12$ is tricky:

$$H(\langle 0, \frac{12}{12} \rangle) = -0 \cdot \log_2 0 - 1 \cdot \log_2 1 = -0 \cdot -\infty - 1 \cdot 0 = -0 \cdot -\infty = ?$$

To make a numerical sense of this quantity we will assume, exclusively for the purposes of computing entropy, that $0 \cdot \infty \equiv 0$. Thus: $H(\langle 0, \frac{12}{12} \rangle) = 0$.

Calculating entropies

Calculating entropy requires computing with base 2 logarithms, which is not easy in memory, and results in ugly irrational numbers. For example, the outcome of flipping the coin, which comes up heads in 99% cases, has the entropy:

$$H = -(0.99 \log_2 0.99 + 0.01 \log_2 0.01) \approx 0.08 \text{ bits}$$

Here is a useful formula to calculate base 2 logarithms on a calculator which only has base 10, or natural (base e) logarithms:

$$\log_N X = \frac{\log_M X}{\log_M N}$$

Finally, a reminder of some useful formulas to compute logarithms by hand:

$$\begin{array}{rcl} \log_2 0 & = & -\infty \\ \log_2 1 & = & 0 \\ \log_2 2 & = & 1 \end{array} \qquad \begin{array}{rcl} \log_2 4 & = & 2 \\ \log_2 8 & = & 3 \\ \log_2 16 & = & 4 \end{array} \qquad \begin{array}{rcl} \log_N A \cdot B & = & \log_N A + \log_N B \\ \log_N A/B & = & \log_N A - \log_N B \end{array}$$

This way, logarithms of larger numbers can be computed as a sum of the logarithms of their factors down to prime numbers. Having a table of some small prime number logarithms allows an easy calculation of a logarithm of a larger value. For example having $\log_2 3 \approx 1.585$:

$$\log_2 18 = \log_2(2 \cdot 3 \cdot 3) = \log_2 2 + 2 \cdot \log_2 3 \approx 1 + 2 \cdot 1.585 = 4.17$$

Building decision trees

We divide a set of samples belonging to a number of classed $c_1, c_2, ...c_n$ into groups according to some key (feature) and we install these groups as branches of a tree, whose root is the initial set of samples.

n_b^c – the number of samples from class c in branch b

n_b – total number of samples in branch b

n_t – total number of samples in all branches

The entropy of a single branch can be expressed with the formula:

$$H_b = \sum_c \frac{n_b^c}{n_b}(-\log_2 \frac{n_b^c}{n_b})$$

The total entropy of all the branches resulting from the division of the set of samples based on the values of some attribute can be computed as:

$$H = \sum_b \frac{n_b}{n_t} \times H_b = \sum_b \frac{n_b}{n_t} \times (\sum_c \frac{n_b^c}{n_b}(-\log_2 \frac{n_b^c}{n_b}))$$

By computing this way the total entropy values for the divisions generated by various attributes we can determine the attribute which gives the lowest entropy, or maximum information gain helping to identify the class of a sample. If, after doing this, some branches of the tree still contain non-uniform (not single class) sets of samples, then this procedure should be repeated individually for all such branches, eventually building a complete tree with uniform sample sets in all the leaves.

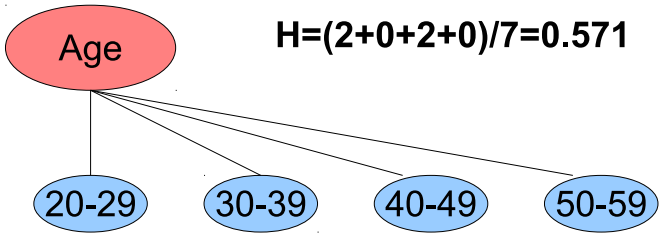
The tree built using this method is a minimal tree ensuring a classification of a given set of samples. Learning unknown concepts this way is consistent with the Ockham's razor principle, stating that the simplest structure compatible with the observation of a concept is probably the most correct.

Decision trees — selecting the main attribute: example

Let us compute the entropy of all the attributes for the credit decisions example:

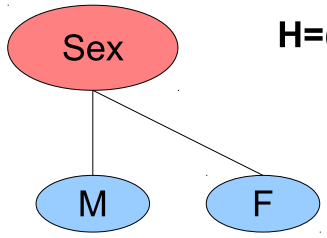
f-n	l-n	age	sex	income	educ.	empl.	crim.	...	approval
-	-	30	M	18,000	univ.	2	N	...	Y
-	-	42	F	12,000	univ.	8	N	...	N
-	-	46	M	58,000	univ.	14	N	...	Y
-	-	55	M	22,500	coll.	6	Y	...	N
-	-	35	M	36,000	univ.	4	N	...	Y
-	-	22	F	30,000	univ.	< 1	N	...	N
-	-	28	M	25,000	elem.	8	N	...	Y

The entropy of the whole set is $-\frac{3}{7}\log_2(\frac{3}{7}) - \frac{4}{7}\log_2(\frac{4}{7}) \approx 0.985$



H=(2+0+2+0)/7=0.571

Y	1	2	1	0
N	1	0	1	1
H	1	0	1	0



H=(0.722*5+0)/7=0.516

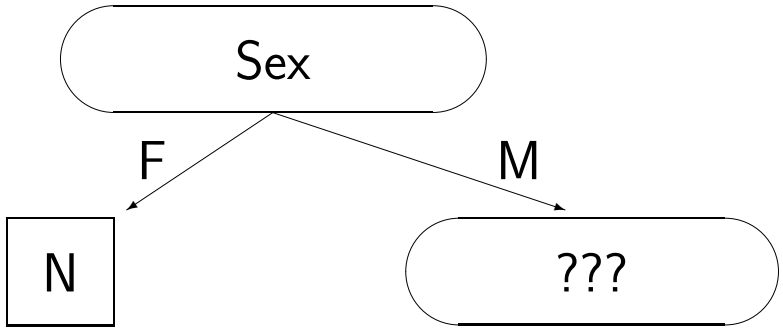
Y	4	0
N	1	2
H	0.722	0

The complete set of entropy values for all attributes:

$$\begin{array}{lcll} E_{\text{age}} & = & 4.00/7 & = & 0.571 \\ E_{\text{sex}} & = & 3.61/7 & = & 0.516 \\ E_{\text{income}} & = & 4.76/7 & = & 0.680 \\ E_{\text{educat.}} & = & 4.85/7 & = & 0.694 \\ E_{\text{employ.}} & = & 6.76/7 & = & 0.965 \\ E_{\text{crimin.}} & = & 5.51/7 & = & 0.787 \end{array}$$

The attribute with the lowest entropy is Sex, with entropy of 0.516. This compares to the initial entropy of the whole set of 0.985, so the information gain of Sex is 0.469.

The initial step of constructing the decision tree is therefore:



Exercise: determine the next attribute for the branch Sex=M.

Some remarks on the information entropy

The concept of entropy used in the information theory comes from Shannon (1948). In physics, a slightly different concept of thermodynamic entropy is used, with units of J/K (joul/kelvin).

In computing entropy, logarithms are used, which corresponds to the length of a string necessary to encode the given information. Eg. if we combine two systems, each with a 1000 internal binary states (for simplicity assume it is 1024 binary states), then the full state of each of these can be described with 10 bits of information. But to encode the full state of the combined systems (1048576 states), 20 bits are necessary. This also explains the use of base 2 logarithms.

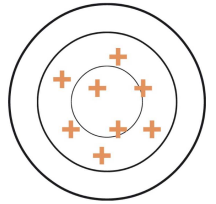
The entropy and information gain of a single variable can be greater then 1. For example, if a variable has four possible values, then to encode its value we need 2 bits. Indeed, computing the entropy of a uniform distribution of this variable we get:

$$H = \sum_{i=1}^{4} P(v_i)(-\log_2 P(v_i)) = \sum_{i=1}^{4} 0.25 \cdot (-\log_2 0.25) = 4 \cdot (0.25 \cdot \log_2 4) = 2$$

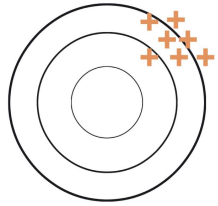
Errors in data

All inductive learning processes are sensitive to errors in data, to a higher or lower degree. Such errors include: incorrectly or inaccurately recorded, or entirely missing attribute values, or incorrectly recorded classification results. These errors can be of different nature and origins. But in a real production environment, such errors are unavoidable when processing thousands or hundreds of thousands of samples. Therefore the sensitivity of the learning process to such errors should always be considered.

Random errors, also known as **noise**, are inherent in all measurement processes, are unpredictable and uncontrollable, and typically exhibit high variance and low bias.



Systematic errors occur in measurement processes due to consistent errors in hardware or software, incorrect assumptions, methods, parameter settings, etc. They tend to have low variance and high bias.



In machine learning systematic errors are worse because they can result in producing models with significant bias. Random errors, on the other hand, tend to cancel each other out, and many machine learning methods can effectively deal with them, producing models with low bias and variance.

Problems with missing and/or erroneous data

missing values for some attributes in some samples

Many real datasets, recorded over a period of time by humans or electronic sensors, have some data missing.

To solve this we can eliminate the affected samples, or attribute(s). But either way we lose data, which may be acceptable, or may not. An alternative approach is to substitute missing values with artificially generated data. They must be generated randomly, carefully reproducing the distribution of the specific attribute, scaling the number of such samples appropriately.

erroneous values for some attributes

There may also be incorrect data present in the dataset. This is worse, since this may not be obvious, and may lead to suboptimal learning, or even the inability to learn some datasets.

Such problems can be detected by comparing the learning outcome on different subsets of data. The „suspicious" samples can be eliminated from the dataset, of the error values may be eliminated from the affected samples, and replaced as above.

The stopping condition

The original procedure for building a decision tree classifier stops when the set of samples is uniform in the recursive tree-building process.

However, considering real-life cases, with data errors present, we should extend this condition. Whenever the data set in the recursive step contains samples with identical input attribute values, but different class attribute value, the procedure should also stop. But what should be the classification value returned, when classifying a new sample with such attribute vector? Clearly, the situation in building the classifier indicates an error, so one might try to resolve the problem.

If there is a clear majority value in the training subset, then this majority value could be taken. If there is a significant set of class values with some distribution, the situation is different. If the classifier being constructed permits the "Undecided" response, indicating the some other action should be taken, then it is perhaps the most proper. But if the classifier must be fully automatic, with no human intervention possible then a possibility might be to draw a random class value using the distribution of the original subset.

The stopping condition continued

The above corrections do not definitely resolve the stopping condition problem. There might be errors in data without multiple samples with identical x-values and different y-value. Samples with identical y-values could be differentiated by some insignificant attribute, or by a small numerical value. Such insignificant attribute, or a numerical difference, would never make it into the decision tree classifier, if it was not for the erroneous samples.

To detect, and properly resolve such cases, a more sophisticated stopping condition might be used. The information gain measure could be used to evaluate whether the recursive decision tree node should be built, or whether it should be stopped, and the value computed using the above procedure.

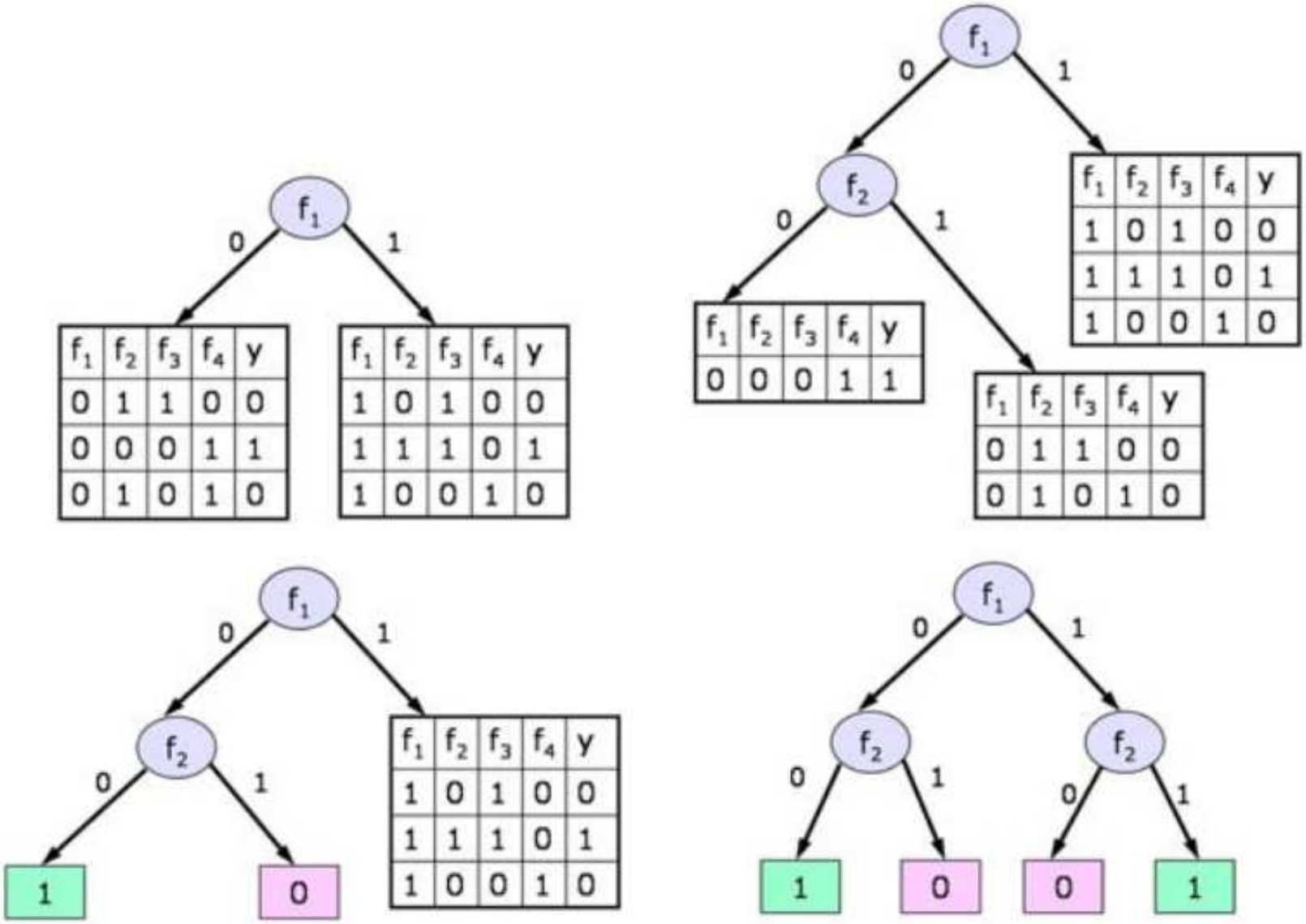
The stopping condition (3)

Unfortunately, this also does not always work well. Some difficult cases in classification result from data patterns which are like the logical "exclusive-OR." Consider the following dataset, which results from the $(f_1 \wedge \neg f_2) \vee (\neg f_1 \wedge f_2)$ function:

f_1	f_2	f_3	f_4	y
0	1	1	0	0
1	0	1	0	0
1	1	1	0	1
0	0	0	1	1
1	0	0	1	0
0	1	0	1	0

The entropy of the whole data set is 0.92, but the entropies of the splits on all four attributes are also 0.92. The above stopping condition would make the algorithm stop growing this tree.

If, instead, we decided to make a split on the f_1 and then on the f_2 attributes, we would get the following results:



Decision tree pruning

As we have seen, it might be useful to proceed building the decision tree even with little or no information gain. But how to tell this situation apart from overfitting?

An advanced version of the decision tree algorithm grows the tree as far as it is possible, and upon completion decides whether the result is a useful classifier, or whether the tree should be pruned and one of the above stopping conditions applied.

Problems with numerical data

The classification learning algorithms assume each attribute has a finite set of values. For numerical, or multi-valued parameters, we have problems:

continuous or infinite domains of attributes

Discretization can be used, but the specific ranges of values are often critical. They can be selected automatically, using statistical analysis, or manually, if we know what ranges are important for the specific case.

discrete but multi-valued attributes

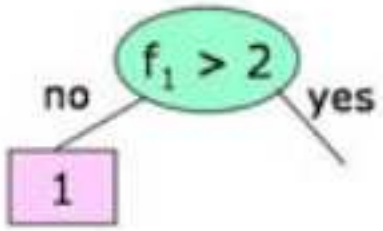
The groups defined by their values will be relatively small, and therefore often highly uniform. So they may appear to be useful for classification, while in fact being worthless. Such cases can be detected by computing the "relative gain" of an attribute as gain relative to its information contents.

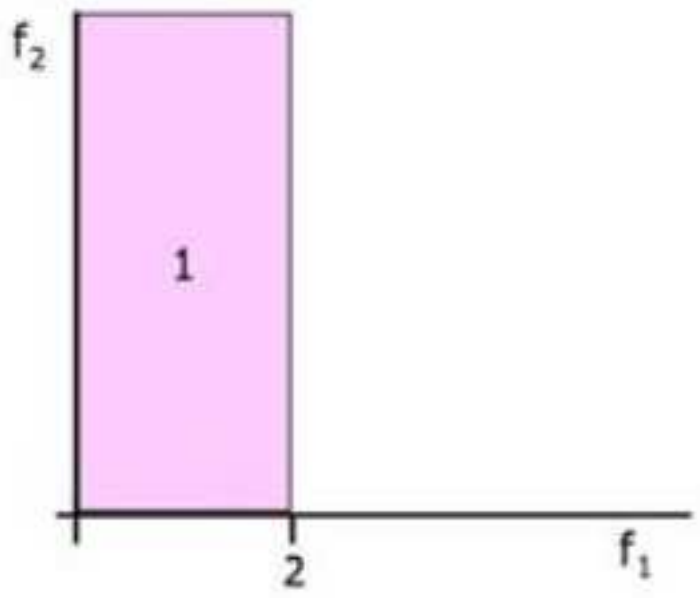
continuous output (class) attribute

In this case we not want to select the output value from a small discrete set, but compute a numerical value. This is a problem of **regression**, which is different from classification. There are different methods to deal with it.

Binary decision trees

The previous approach to building decision trees was to discretize the numerical attributes and treat them as qualitative. An alternative approach is to use only binary tests of the form $x_j > c$ in the branches of the tree.

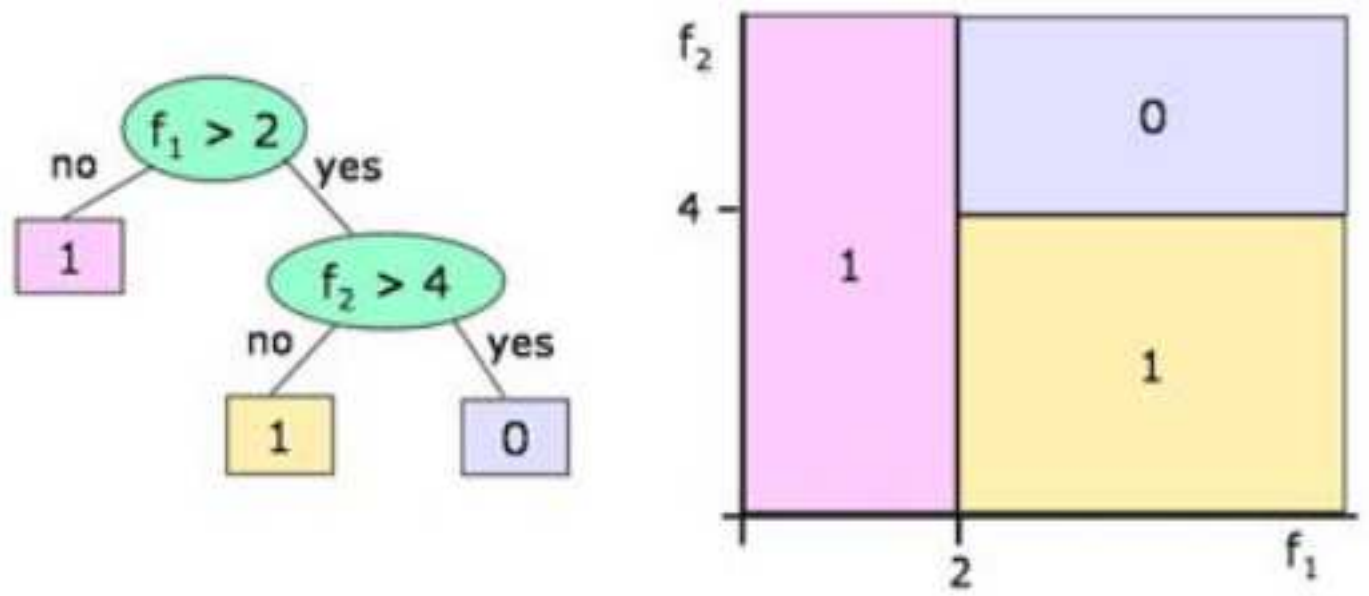




This will divide the feature space into rectangles, or hyper-rectangles in high-dimensional space.

Decision trees with binary splits on numerical attributes

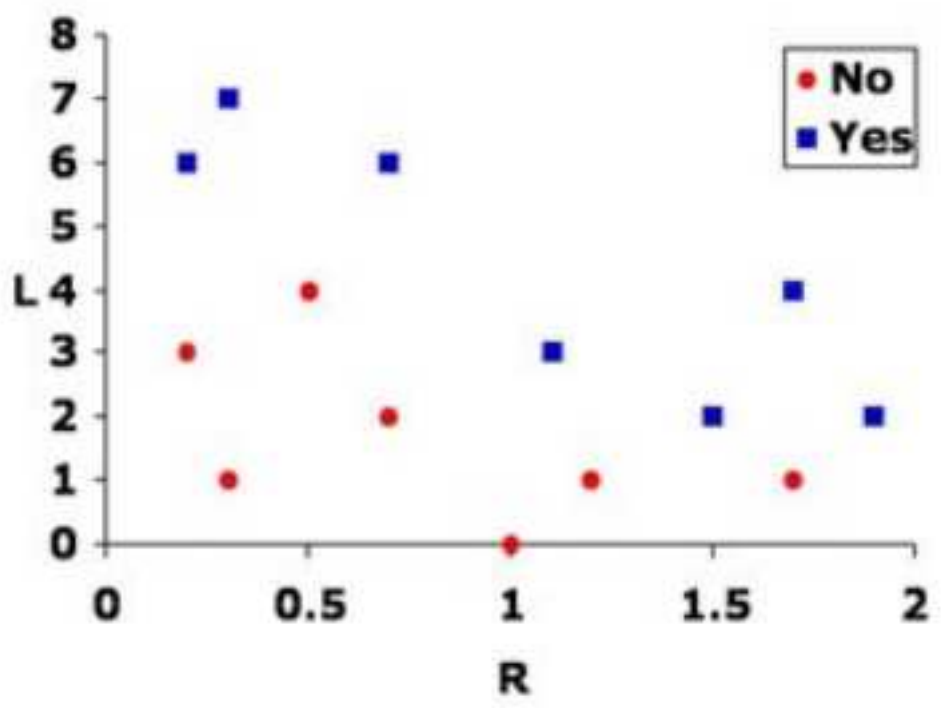
The class of hypotheses allowed in this model is fairly rich, although may not be able to express some concepts.



Predicting Bankruptcy

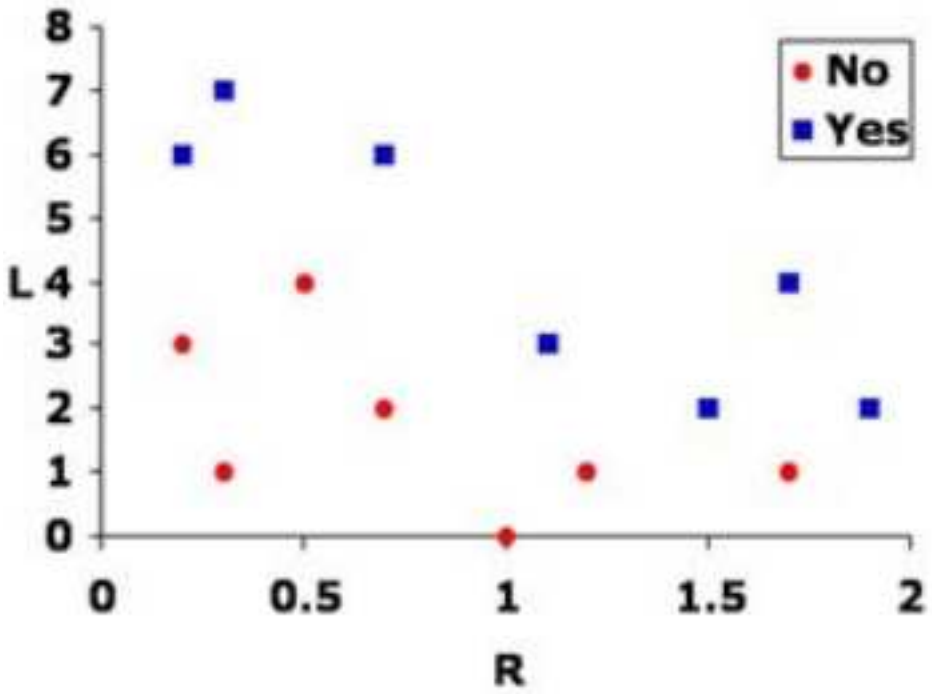
Suppose we are trying to predict some person's near future bankruptcy by observing two parameters: number of late credit payments during the year, and the ratio of their expenses to their income. If these parameters are large, the chances of going bankrupt are high.

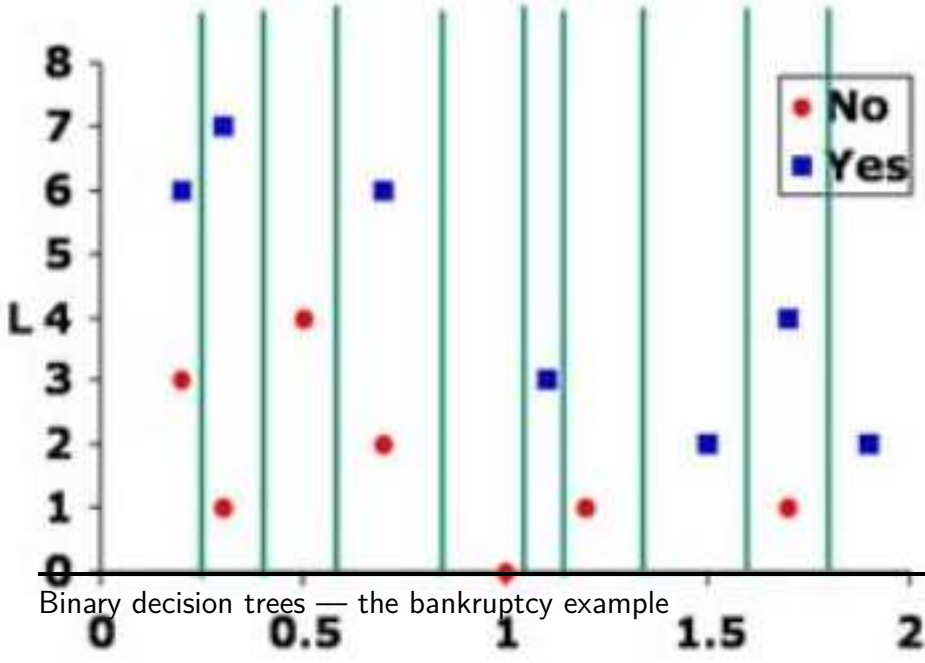
L	R	B
3	0.2	No
1	0.3	No
4	0.5	No
2	0.7	No
0	1.0	No
1	1.2	No
1	1.7	No
6	0.2	Yes
7	0.3	Yes
6	0.7	Yes
3	1.1	Yes
2	1.5	Yes
4	1.7	Yes
2	1.9	Yes

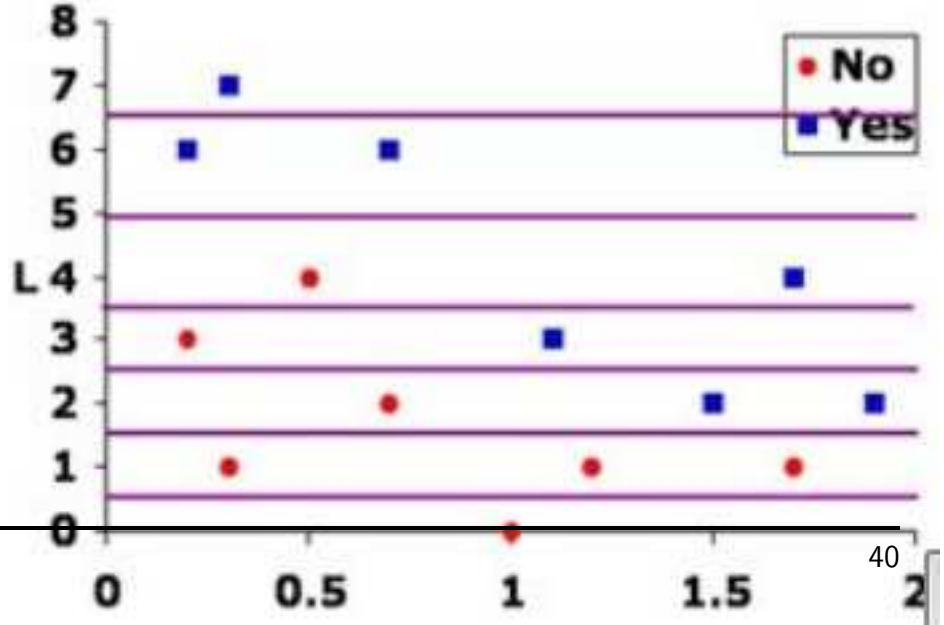


Considering splits

We want to consider splitting between any two points in each dimension.

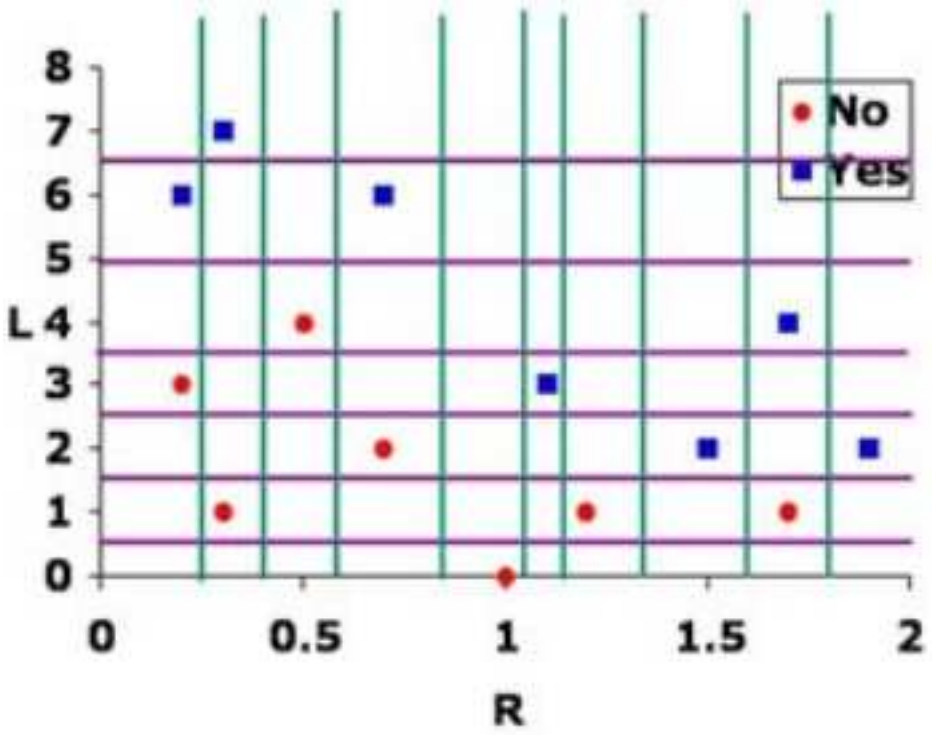






Considering splits (ctd.)

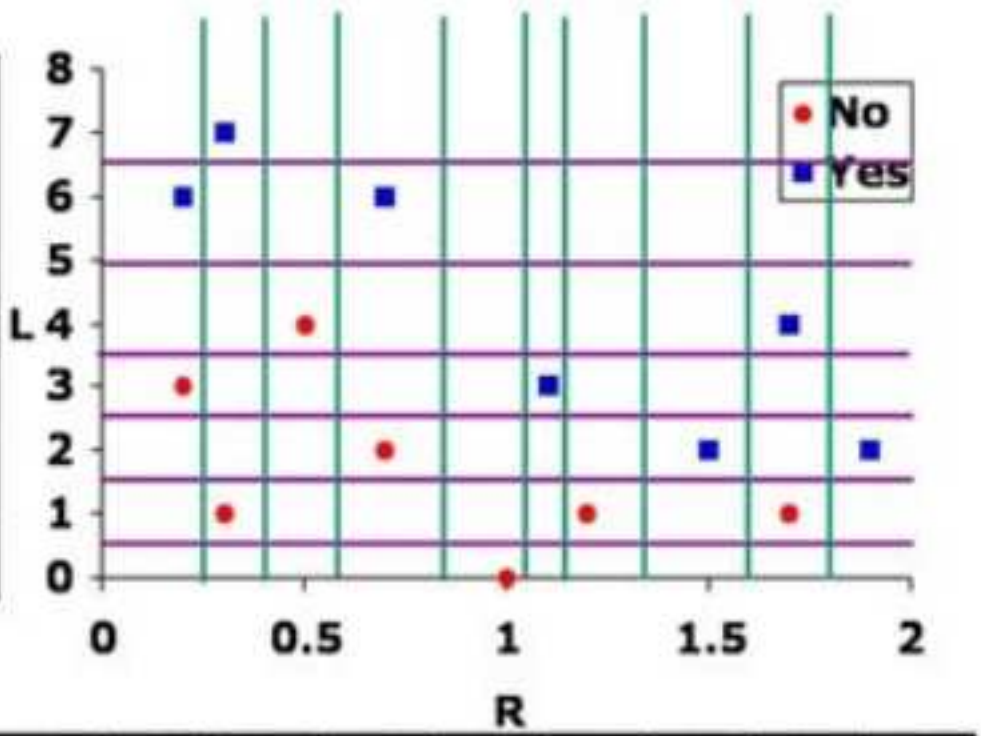
We have quite many possible splits to consider. We are only going to select one, and this will be the one that minimizes the average entropy of its resulting child nodes.



Decision tree for the bankruptcy dataset

Computing the average entropies for all possible splits:

L<y	NL	PL	NR	PR	AE
6.5	7	6	0	1	0.93
5.0	7	4	0	3	0.74
3.5	6	3	1	4	0.85
2.5	5	2	2	5	0.86
1.5	4	0	3	7	0.63
0.5	1	0	6	7	0.93



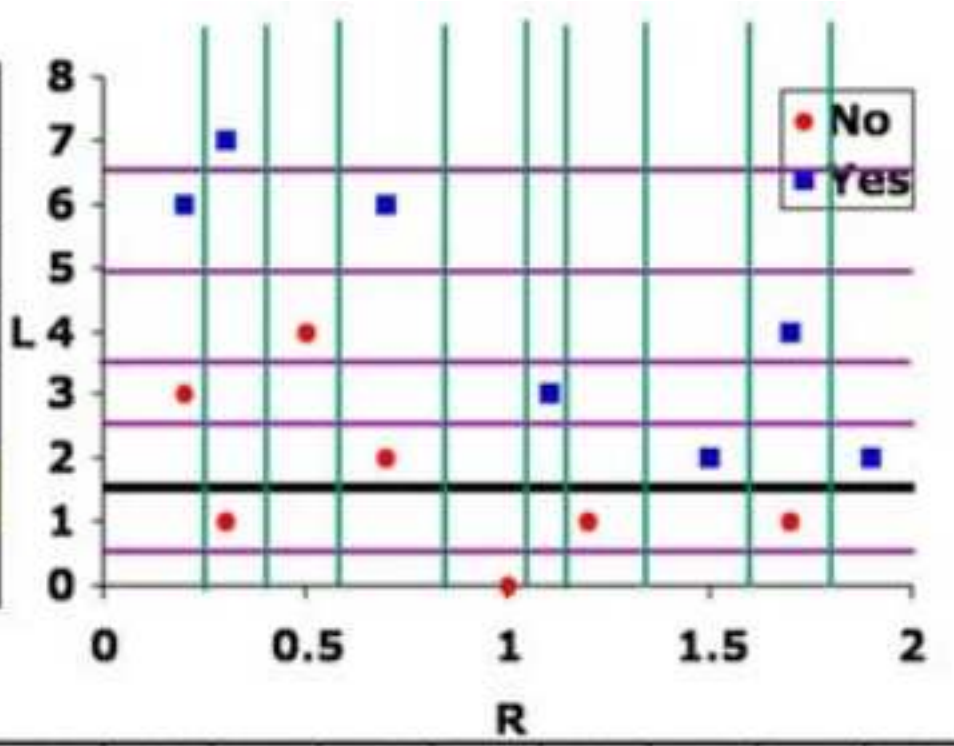
neg to left | # pos to left | # neg to right | # pos to right

AE	1.00	1.00	0.98	0.98	0.94	0.98	0.92	0.98	0.92
R<x	0.25	0.40	0.60	0.85	1.05	1.15	1.35	1.60	1.80

Decision tree for the bankruptcy dataset (ctd.)

The split on $L > 1.5$ gives the minimum entropy for the first split:

L<y	NL	PL	NR	PR	AE
6.5	7	6	0	1	0.93
5.0	7	4	0	3	0.74
3.5	6	3	1	4	0.85
2.5	5	2	2	5	0.86
1.5	4	0	3	7	0.63
0.5	1	0	6	7	0.93

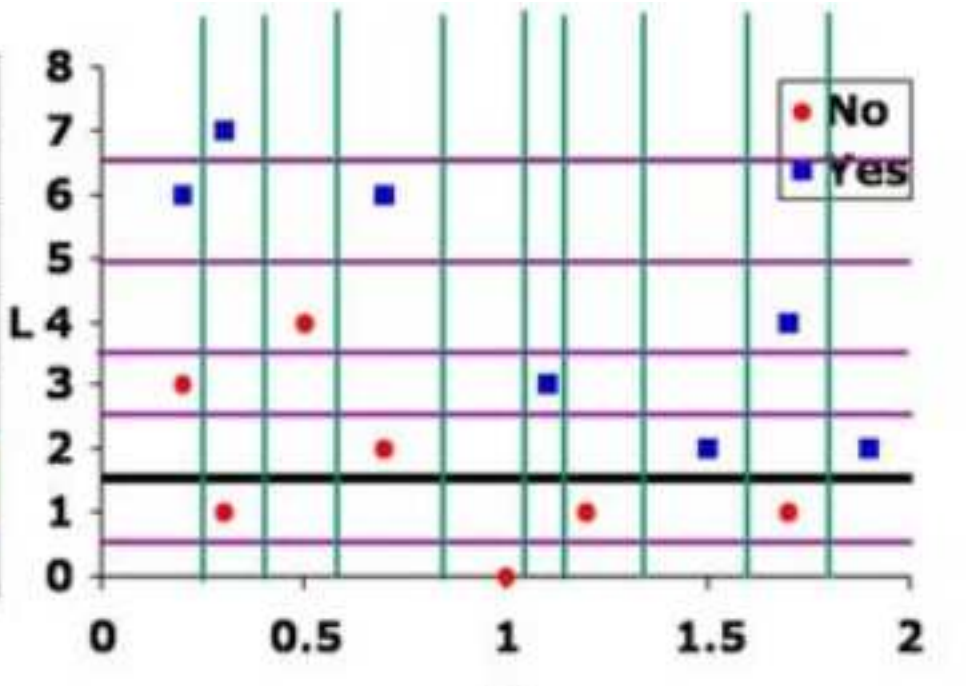


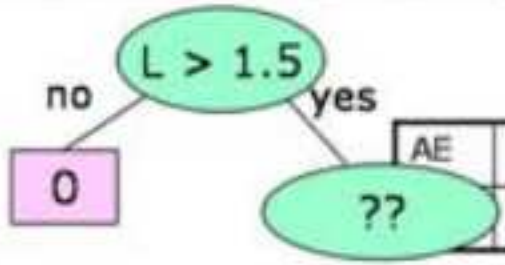
AE	1.00	1.00	0.98	0.98	0.94	0.98	0.92	0.98	0.92
R<x	0.25	0.40	0.60	0.85	1.05	1.15	1.35	1.60	1.80

Decision tree for the bankruptcy dataset (ctd.)

Since all the points satisfying $L < 1.5$ are in class red (No bankruptcy), then we may create the leaf in the tree.

L<y	NL	PL	NR	PR	AE
6.5	7	6	0	1	0.93
5.0	7	4	0	3	0.74
3.5	6	3	1	4	0.85
2.5	5	2	2	5	0.86
1.5	4	0	3	7	0.63
0.5	1	0	6	7	0.93



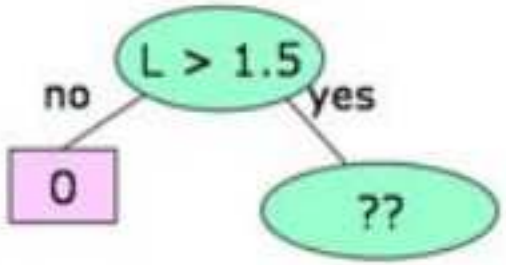


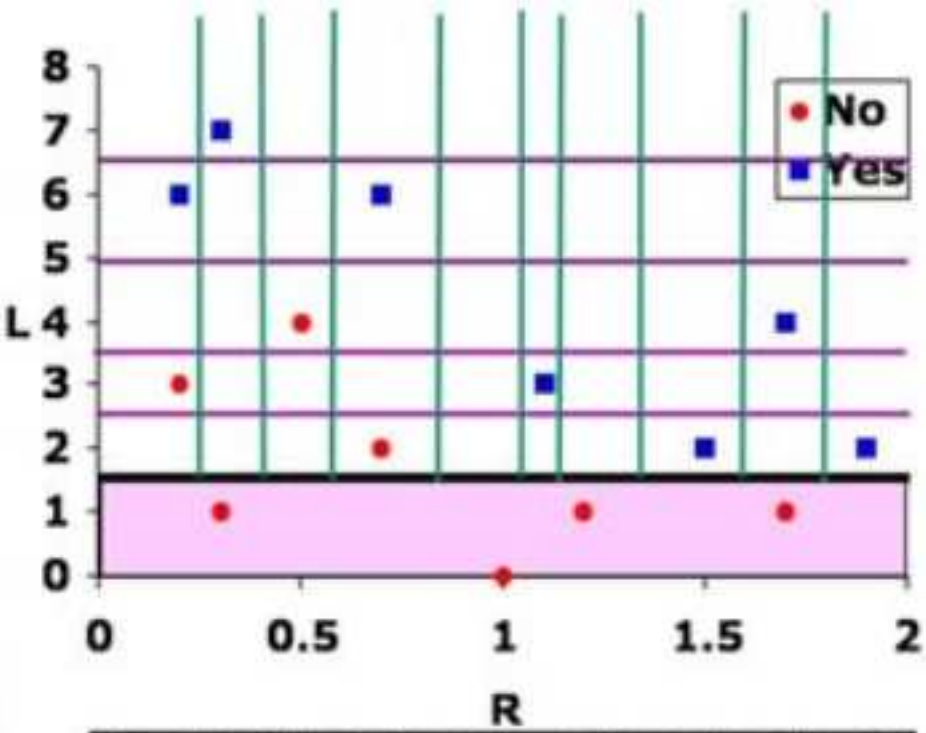
AE	1.00	1.00	0.98	0.98	0.94	0.98	0.92	0.98	0.92
	0.25	0.40	0.60	0.85	1.05	1.15	1.35	1.60	1.80

Decision tree for the bankruptcy dataset (ctd.)

We now calculate the second split in the tree. For this all entropies need to be recalculated, because the leaf samples are taken out of consideration.

L<y	NL	PL	NR	PR	AE
6.5	6	3	0	1	0.83
5.0	4	3	0	3	0.69
3.5	3	2	4	1	0.85
2.5	2	1	5	2	0.88



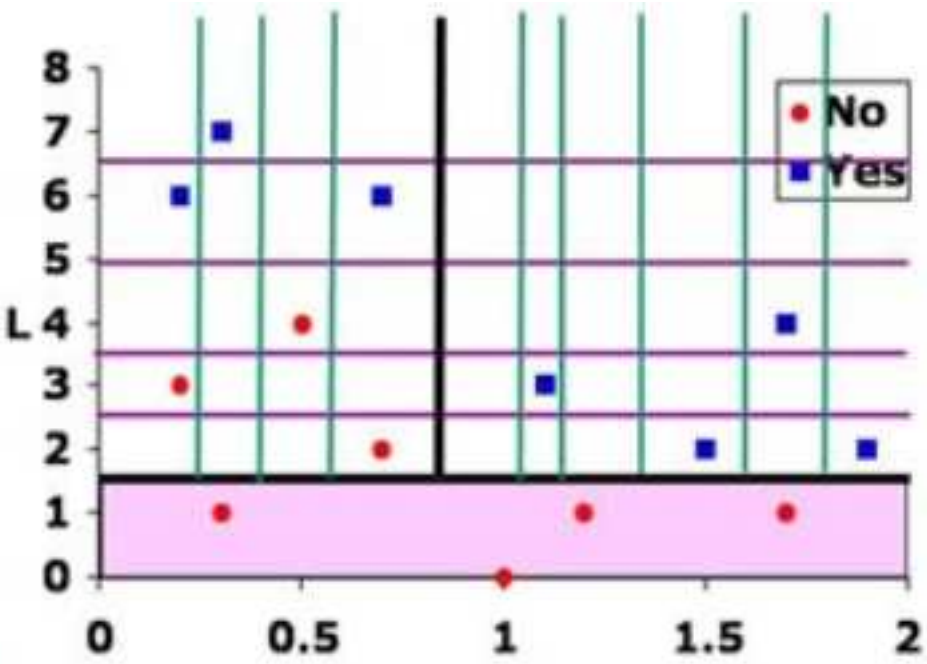


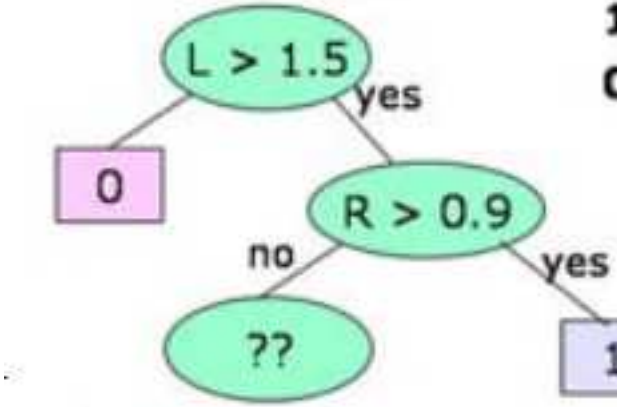
AE	0.85	0.88	0.79	0.60	0.69	0.76	0.83
R<x	0.25	0.40	0.60	0.90	1.30	1.60	1.80

Decision tree for the bankruptcy dataset (ctd.)

In this case the split on $R > 0.9$ has the least entropy. The tree is extended and the new leaf created and labeled.

L<y	NL	PL	NR	PR	AE
6.5	6	3	0	1	0.83
5.0	4	3	0	3	0.69
3.5	3	2	4	1	0.85
2.5	2	1	5	2	0.88



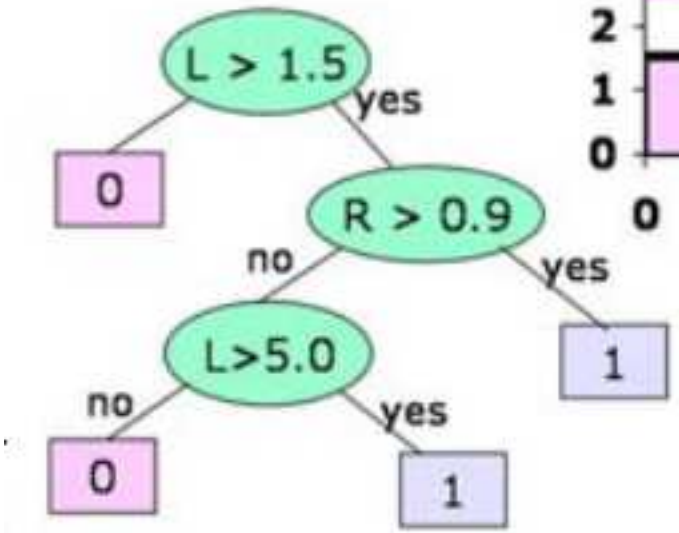


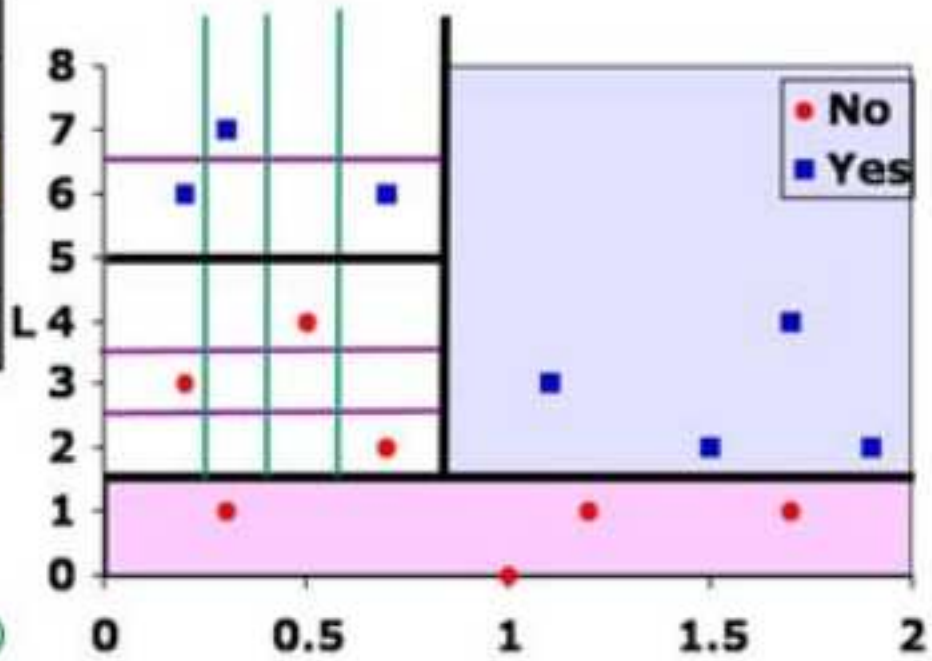
AD	0.85	0.88	0.79	0.60	0.69	0.76	0.83
R<x	0.25	0.40	0.60	0.90	1.30	1.60	1.80

Decision tree for the bankruptcy dataset (ctd.)

Once again we calculate the complete set of entropies of the remaining samples. For $L > 5$ we get the next entropy-minimal split, which also finally separates the output classes.

L<y	NL	PL	NR	PR	AE
6.5	3	2	0	1	0.81
5.0	3	0	0	3	0.00
3.5	2	0	1	3	0.54
2.5	1	0	2	3	0.81

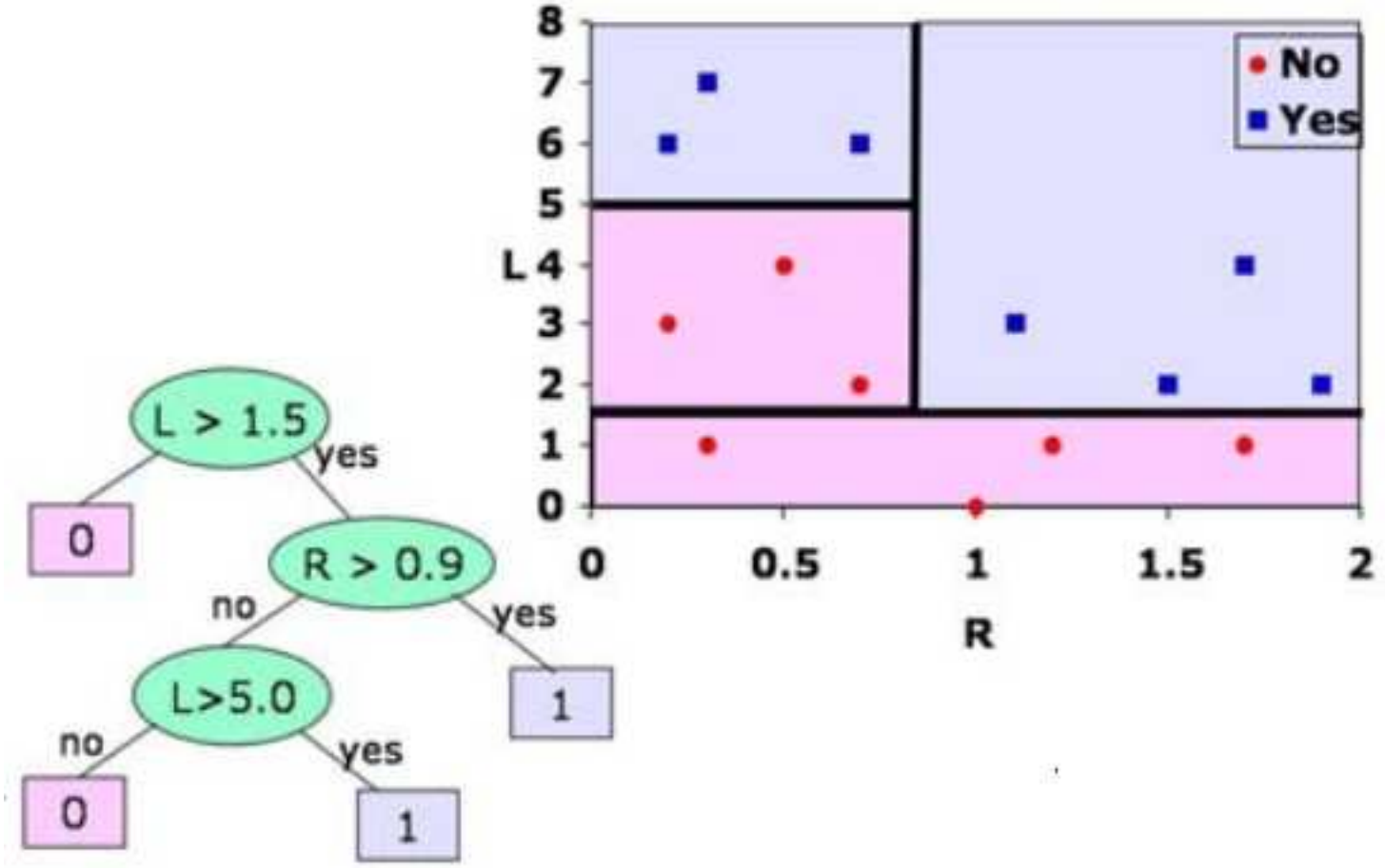




AE	1.00	0.92	1.00
R<x	0.25	0.40	0.60

Decision tree for the bankruptcy dataset (ctd.)

The complete tree has zero error on the training set.



Errors in induction learning

The simplest way to evaluate the results of the learning process is to classify all samples from some set for which the true class is known, and compute **Accuracy**:

$$\text{Accuracy} = \frac{\#\text{ of correctly classified samples}}{\#\text{ of all classified samples}}$$

Alternatively, we may compute **Error** as the complement of Accuracy:

$$\text{Error} = 1 - \text{Accuracy}$$

Accuracy may be used to compare the results of learning of the same data by different learning algorithms. However, it does not tell the whole story, of what has been learned and what has not.

Problems with Accuracy

The Accuracy is the single measure of the trained classifier's performance on some dataset. However, its value may often be hard to interpret or compare.

The absolute value of Accuracy does not have a universal meaning. Accuracy=10% may be wonderful for picking a ticket in a national lottery. But Accuracy=90% may be considered low when choosing the antidote for a venomous snake bite.

Accuracy=50% in binary classification is equivalent to random guessing. However, for a 13-class case the same 50% means that half of the samples will be assigned the exact, 1-of-13 class. For a specific purpose this might or might not be useful, but it is a significantly positive result.

Further, consider the screening test for cancer. Suppose five in 1000 people have a developing disease and it can be detected early by a screening test. Classifier A correctly identifies 4 of the 5 cancer cases, and incorrectly marks as cancerous 25 out of 995 healthy people, for the total Accuracy=97.1%. Classifier B only recognizes 2 out of 5 people, and incorrectly classifies 15 healthy people, thus its Accuracy=98.2% is better than A's. But its ability to detect cancer is obviously much worse.

Training, validation, and testing sets

Each supervised learning algorithm produces a classifier based on some set of data called the **training set**, for which both the input feature values and the class are known. The classifier can then be evaluated by classifying the samples from the same training set. Alternatively, it can be tested on samples from another set, called the **testing set**, for which both the input features and the class must also be known.

The testing set, however, cannot be examined at any time during learning. By definition, it can only be used for the final, one-time evaluation of the performance of a classifier. In an academic setting this set typically is available to the programmer building the classifier. But in a real-world industrial scenario such set should remain inaccessible (or rather top-secret), to assure the objectivity of the final test. Otherwise there might arise a doubt that the manufacturer, in the quest to gain market edge, might fine tune the classifier for the top performance on the testing set, disregarding the general data performance.

But while building a classifier, selecting the model and fine tuning its parameters, multiple testing is required in the iterative build cycle. For that purpose, and additional testing set is created, called the **validating set**, different from the training set, but which can be used in the training, for the evaluation and fine tuning of the classifier.

Training, validation, and testing sets (ctd.)

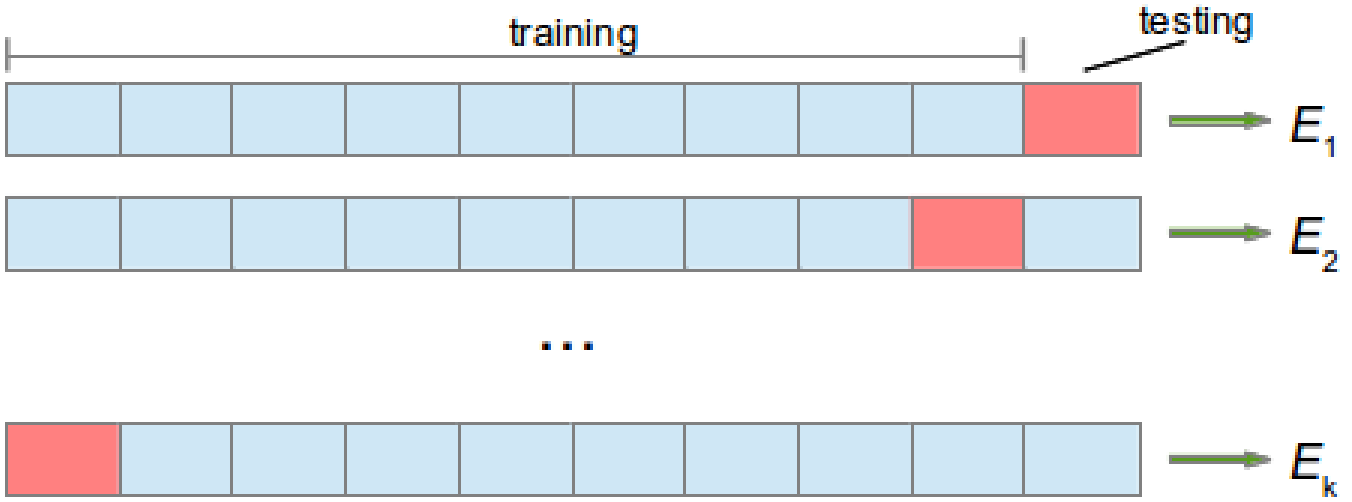
Obviously, for a successful construction of a classifier both the training and validation (and for reliable test also the testing) datasets should have properties most similar to the real data. This is the reason why samples are randomly selected for randomized medical therapy trials, social opinion polling, etc.

Likewise, the training and validation datasets should be randomly generated from the available data. However, each data selection, despite random, is biased, in both mean value and variance. In other words, the samples of the validation set may be either particularly easy or particularly hard to classify, or could be altogether different, from the training set. This distorts the ability to evaluate the objective performance of a classifier, and may lead to its suboptimal tuning.

To ensure statistical reliability, both sets should be as large as possible, which is easy to achieve when there is a lot of available data, or there is an unlimited ability to acquire new data. However, in many cases the available dataset is limited.

Cross-validation

A simple and powerful method for using a limited dataset for both training and validation is **cross-validation**. The available dataset is divided randomly into k equal subsets (k-folds). Then the first k-1 subsets are used for training, and the last subset for validation, with the resulting validation error E_1. The procedure is repeated k times, each time using a different subset for validation.



Then the overall cross-validation Error is computed as the average $E = \frac{\sum_i E_i}{k}$. It estimates the Error value for any set with properties similar to the available dataset. The best value for k may be found experimentally for the specific data, but often used is k=10.

LOO validation

Cross-validation is the basis for machine learning work. However, for very small dataset (eg. below 100 samples), and particularly for very small values of k, each of the validation experiments eg. with k=5 may be based on training with a non-representative training set. In such cases a special form of validation may be used called the **LOO** (Leave-One-Out), or **LOOCV** (Leave-One-Out Cross-Validation). It works by selecting in turn each individual sample from the available dataset, and using it for validation after training with the remaining N-1 samples. The Error is averaged, like in cross-validation.

Note that LOO validation is equivalent to cross-validation with k=N.

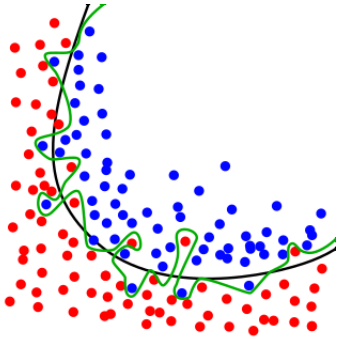
Errors in learning — overfitting

Overfitting is a common problem in machine learning. It appears in many forms, and can lead to severely incorrect results. A typical scenario is that a model is trained, and appears to work successfully, when tested on the training set. However, when tested on a different dataset (testing set) the results are much worse, or completely unacceptable.

One possibility is, that some very regular pattern is learned from a limited dataset, and the resulting model is perturbed by some particular phenomena, such as **outliers** (individual highly irregular samples) or randomness (eg. a subset exhibiting similar, very specific, but not typical, properties).

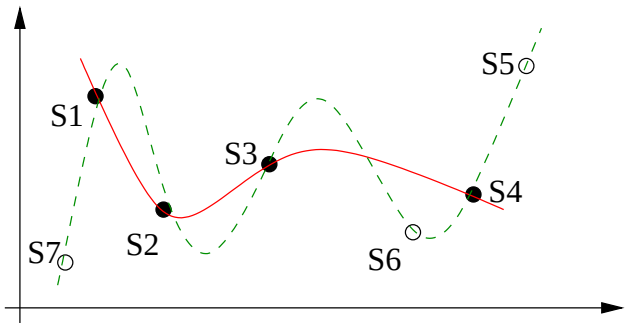
Another possibility is that the model is trained to adapt to the dataset so accurately, that it reflects not only regular properties, but also the noise contained in the dataset. This is particularly likely when using very advanced machine learning algorithms capable of producing very complex and highly flexible models.

In this example it appears, that the black line fits the data well, but the continued training can lead to the green line, as it exhibits zero error on the training set.



Errors in learning — underfitting

Much larger error values on the test data than on the training data do not always indicate overfitting, however. In general, the larger the training set, relative to the number of input attributes and the size of the hypothesis space, the lower is the danger of overfitting. Large errors on the test set may also result from **underfitting**, which may be due to a too small training set, or a too simple model, inadequate machine learning algorithm, insufficient learning, etc.



Learning the points S1,...,S4 might lead to the red continuous curve. The validation/test points S5,S6,S7 exhibit large errors, but in this case this is due to underfitting. Repeated learning with these points has lead to the green broken line.

Determining whether errors obtained in testing result from overfitting or underfitting, or perhaps something else, is a difficult problem in machine learning, and often requires running many experiments.

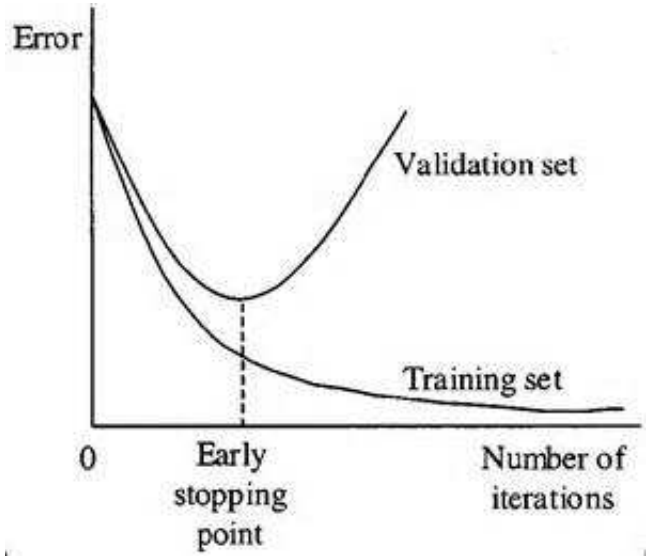
Errors in learning — detecting overfitting

Many approaches have been developed in machine learning to combat overfitting. Some of them are specific to, and/or built into a particular ML algorithm, while others are generic and can be used universally. We will learn a number of these.

A simple and very often effective approach is to aquire more good quality data. As we have seen, one of the facets of overfitting, is a biased model obtained on a noisy dataset. With a larger dataset, even with the same noise present, many ML algorithms will be able to produce a better quality models, as in a large set the noise will often have a tendency to cancel out.

Another general approach to avoid overfitting is the **early stopping** method.

It works by iteratively improving the trained model, while observing its performance by comparing the error calculated on the training set and a separate validation set. Each round of classifier optimization makes both errors to go down. It is natural and expected for the validation error to be slightly larger, but also decreasing, following the training set error. However, if and when the validation error stops dropping and starts growing, while the training set error continues to drop, it suggests that overfitting has occurred.



Training and validation

Because testing can be used to detect over- and underfitting, and indicate that more training (on more data) is required, the traditional division of data into the training and testing sets is not always adequate. Testing not only serves the purpose of final assessment of the classifier's performance, but also helps in the construction of the classifier.

This can be either done by cross-validation (especially if the available data are few), or by the following **data window** approach (useful when there is plenty of data). We may use only a small fraction of data for training, and use the rest for validation.

In case the validation ever fails, the learning is repeated for the training set with the addition of samples giving errors. After that, validation is resumed on the remaining data. This leads to splitting the data into: training, validation, and testing sets, with the latter only used for the final evaluation.

The Naive Bayes Classifier

We will introduce one of the most interesting methods of machine learning, one of the simplest and very efficient, the Naive Bayes Classifier, by studying and example. For the set of samples we compute the following fractions for each of the features:

$X_1 X_2 X_3 X_4$	Y
0 1 1 0	1
0 0 1 1	1
1 0 1 0	1
0 0 1 1	1
0 0 0 0	1
1 0 0 1	0
1 1 0 1	0
1 0 0 0	0
1 1 0 1	0
1 0 1 1	0

$R_1(1,1) = 1/5$ - fraction of all positive samples, that have $X_1 = 1$

$R_1(0,1) = 4/5$ - fraction of all positive samples, that have $X_1 = 0$

$R_1(1,0) = 5/5$ - fraction of all negative samples, that have $X_1 = 1$

$R_1(0,0) = 0/5$ - fraction of all negative samples, that have $X_1 = 0$

$R_1(1,1) = 1/5 \quad R_1(0,1) = 4/5$
$R_1(1,0) = 5/5 \quad R_1(0,0) = 0/5$

$R_2(1,1) = 1/5 \quad R_2(0,1) = 4/5$
$R_2(1,0) = 2/5 \quad R_2(0,0) = 3/5$

$R_3(1,1) = 4/5 \quad R_3(0,1) = 1/5$
$R_3(1,0) = 1/5 \quad R_3(0,0) = 4/5$

$R_4(1,1) = 2/5 \quad R_4(0,1) = 3/5$
$R_4(1,0) = 4/5 \quad R_4(0,0) = 1/5$

The computed fractions correspond to the probabilities of a sample with the specific class value having the specific feature value.

$$
\begin{array}{llll}
R_1(1,1) = 1/5 & R_2(1,1) = 1/5 & R_3(1,1) = 4/5 & R_4(1,1) = 2/5 \\
R_1(1,0) = 5/5 & R_2(1,0) = 2/5 & R_3(1,0) = 1/5 & R_4(1,0) = 4/5 \\
R_1(0,1) = 4/5 & R_2(0,1) = 4/5 & R_3(0,1) = 1/5 & R_4(0,1) = 3/5 \\
R_1(0,0) = 0/5 & R_2(0,0) = 3/5 & R_3(0,0) = 4/5 & R_4(0,0) = 1/5
\end{array}
$$

When we need to predict the class value of a new sample, we look at its feature values, and compute the probabilities of having each subsequent class value by multiplying the probabilities of having the specific feature value with that class value.

Additionally, we include the primary probabilities of all class values, eg.:

$$
\begin{aligned}
\text{new } X &= \langle 0,0,1,1 \rangle \\
S(1) &= P(1) * R_1(0,1) * R_2(0,1) * R_3(1,1) * R_4(1,1) = 0.1024 \\
S(0) &= P(0) * R_1(0,0) * R_2(0,0) * R_3(1,0) * R_4(1,0) = 0
\end{aligned}
$$

Since here $S(1) > S(0)$ then we declare the class of the new sample as 1.

More formally, we compute the following quantities R_j for each feature X_j by counting samples i with the specific values of X_j^i and Y^i:

$$\begin{aligned} R_j(1,1) &= \frac{\#(X_j^i = 1 \wedge Y^i = 1)}{\#(Y^i = 1)} \\ R_j(0,1) &= 1 - R_j(1,1) \end{aligned}$$

$$\begin{aligned} R_j(1,0) &= \frac{\#(X_j^i = 1 \wedge Y^i = 0)}{\#(Y^i = 0)} \\ R_j(0,0) &= 1 - R_j(1,0) \end{aligned}$$

Given a new sample X^k, we compute:

$$\begin{aligned} S(1) &= P(1) * \prod_j R_j(X_j^k, 1) \\ S(0) &= P(0) * \prod_j R_j(X_j^k, 0) \end{aligned}$$

Output 1 iff $S(1) > S(0)$

The operation of the classifier might be sped up using logarithms, since it is easier to add, than to multiply small numbers. Computing the logarithms occurs only once during the construction of the classifier.

$$\begin{aligned} \log S(1) &= \log P(1) + \sum_j \log R_j(X_j^k, 1) \\ \log S(0) &= \log P(0) + \sum_j \log R_j(X_j^k, 0) \end{aligned}$$

And output 1 iff $\log S(1) > \log S(0)$

The Laplace corrections

The above scheme works well, except in case when for one of the features there are no samples (zero) with a specific feature and class values. The corresponding R_j factor is 0 then, and such class will never be predicted for that feature value, regardless of other features.

This can be avoided by adding a small numerical quantity l to each sample count (in the numerator of the R_j expression). In order to normalize the resulting probabilities, each R_j denominator is correspondingly incremented by $2l$. This operation is called **smoothing**of the estimate, and the value l is the strength of this smooting. The frequently used l=1 case is called the Laplace smooting.

$$
\begin{aligned}
R_j(1,1) &= \frac{\#(X_j^i = 1 \wedge Y^i = 1) + 1}{\#(Y^i = 1) + 2} \\
R_j(0,1) &= 1 - R_j(1,1)
\end{aligned}
$$

$$
\begin{aligned}
R_j(1,0) &= \frac{\#(X_j^i = 1 \wedge Y^i = 0) + 1}{\#(Y^i = 0) + 2} \\
R_j(0,0) &= 1 - R_j(1,0)
\end{aligned}
$$

X_1	X_2	X_3	X_4	Y
0	1	1	0	1
0	0	1	1	1
1	0	1	0	1
0	0	1	1	1
0	0	0	0	1
1	0	0	1	0
1	1	0	1	0
1	0	0	0	0
1	1	0	1	0
1	0	1	1	0

$R_1(1,1) = 2/7 \quad R_1(0,1) = 5/7$

$R_1(1,0) = 6/7 \quad R_1(0,0) = 1/7$

$R_2(1,1) = 2/7 \quad R_2(0,1) = 5/7$

$R_2(1,0) = 3/7 \quad R_2(0,0) = 4/7$

$R_3(1,1) = 5/7 \quad R_3(0,1) = 2/7$

$R_3(1,0) = 2/7 \quad R_3(0,0) = 5/7$

$R_4(1,1) = 3/7 \quad R_4(0,1) = 4/7$

$R_4(1,0) = 5/7 \quad R_4(0,0) = 2/7$

The Naive Bayes Classifier — the theory

Let us try to come up with a semi-formal justification for the above procedure. We may consider the features $X_1, \ldots, X_n$ to be random variables. We try to learn from the series of samples the $P(Y = 1|X_1, \ldots, X_n)$. Then, given a new sample, we use the learned distribution to compute the probability it has the Y value 1. If that probability is > 0.5, we will answer 1, otherwise 0.

So we need to estimate the distribution $P(Y = 1|X_1, \ldots, X_n)$ from the training data. In the Naive Bayes Classifier method we do this using the Bayes' rule.

The naive assumption

In general, the Bayes rule is:

$$P(A|B) = P(B|A)\frac{P(A)}{P(B)}$$

Specifically, with n features:

$$P(Y = 1|X_1...X_n) = P(X_1...X_n|Y = 1)\frac{P(Y = 1)}{P(X_1...X_n)}$$

The term $P(X_1...X_n)$ is characteristic of the data set, and for a given sample is constant. For the purpose of the sample class determination we need to focus on:

$$P(Y = 1) * P(X_1...X_n|Y = 1)$$

Assuming that the features are all independent, we can compute this as:

$$P(X_1, ...X_n|Y = 1) = \prod_j P(X_j|Y = 1)$$

Such assumption is very often false, yet this method works well in many domains. This is why the classifier is called Naive.

Logistic Regression: introduction

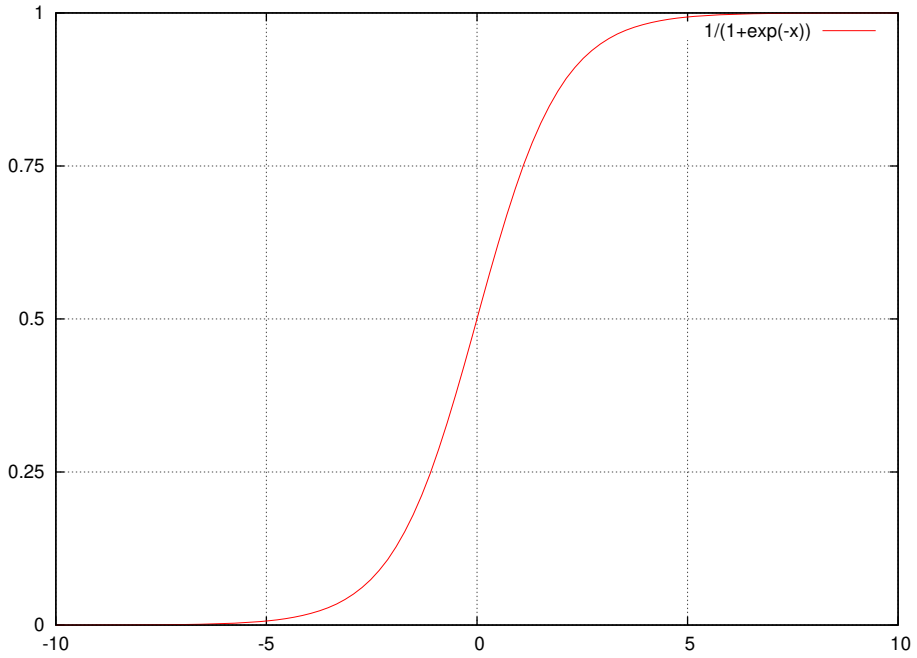
Logistic Regression is a form of classification. It is an approach to learn a function of the form $f : X \to Y$, or $P(Y|X)$ in the case where Y is discrete-valued, and $X = \langle X_1 ... X_n \rangle$ is a vector containing discrete or continuous variables.

The **logistic function** is used to model the probability distribution. The "S" shape of the function is also called the **sigmoid curve**. It is given by the following equation:

$$f(x) = \frac{L}{1 + e^{-k(x-x_0)}}$$

where: x_0 is the x-value of the sigmoid's midpoint, L is the curve's maximum value, and $k =$ the steepness of the curve.

```
$ gnuplot
set grid ytics lt 0 lw 1 lc rgb "#444444"
set grid xtics lt 0 lw 1 lc rgb "#444444"
set ytics .25
plot 1/(1+exp(-x))
```

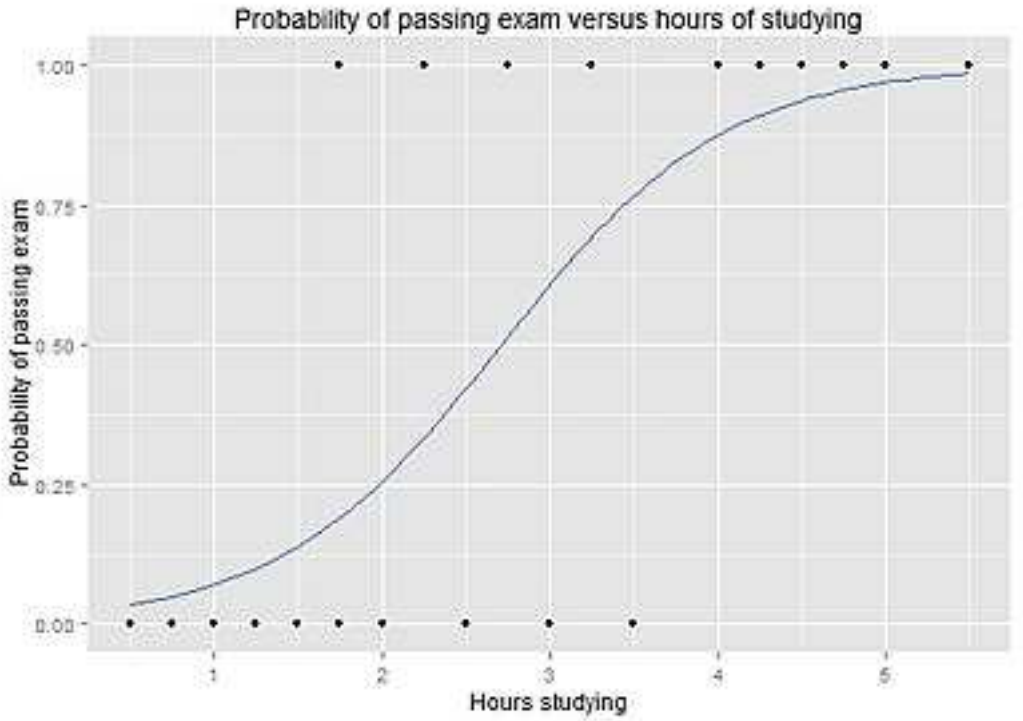


Logistic Regression: example

Consider the following example (Wikipedia): A group of 20 students spent between 0 and 6 hours studying for an exam; then some have passed and some have failed.

What is the probability of passing the exam given the hours studied?

hours	pass	hours	pass
0.50	n	2.75	t
0.75	n	3.00	n
1.00	n	3.25	t
1.25	n	3.50	n
1.50	n	4.00	t
1.75	n	4.25	t
1.75	t	4.50	t
2.00	n	4.75	t
2.25	t	5.00	t
2.50	n	5.50	t



Logistic Regression: theory

First we will consider the case where Y is a boolean variable, for simplicity. Later we will extend the presentation to the multi-valued discrete case.

Logistic Regression assumes the following parametric form for the distribution $P(Y|X)$:

$$P(Y=1|X) = \frac{1}{1+\exp(w_0+\Sigma_{i=1}^{n} w_i X_i)}$$

$$P(Y=0|X) = \frac{\exp(w_0+\Sigma_{i=1}^{n} w_i X_i)}{1+\exp(w_0+\Sigma_{i=1}^{n} w_i X_i)}$$

The parameters w_i will be learned from the training data.

Logistic Regression: theory (2)

Since the result of the learned logistic function is to be used for classification, we only need to be concerned with the following condition:

$$\frac{P(Y=0|X)}{P(Y=1|X)} > 1$$

If the condition holds, then we will assign $Y = 0$. After substituting the general parametric formulas the above condition becomes:

$$\exp(w_0 + \sum_{i=1}^{n} w_i X_i) > 1$$

and after taking a natural logarithm from both sides becomes:

$$w_0 + \sum_{i=1}^{n} w_i X_i > 0$$

Logistic Regression: learning the parameters

One reasonable approach to choose the Logistic Regression parameter values that maximize the conditional likelihood of the training data. The conditional data likelihood is the probability of the observed Y values in the training data, conditioned on their corresponding X values. So we choose parameters W that satisfy:

$$W \leftarrow \arg\max_W \prod_s P(Y^s|X^s, W)$$

where $W = \langle w_0, w_1 ... w_n \rangle$ is the vector of parameters to be estimated, and Y^s, X^s denote the observed value of Y, X in the sth training sample. The expression to the right of the arg max is the conditional data likelihood. W is included in the conditional to emphasize that the expression is a function of the W we are attempting to maximize.

Equivalently, we can work with the log of the conditional likelihood:

$$W \leftarrow \arg\max_W \sum_s \ln P(Y^s|X^s, W)$$

Logistic Regression: learning the parameters (2)

This conditional data log likelihood $\Sigma_s \ln P(Y^s|X^s, W)$, which we will denote $l(W)$, can be written as

$$l(W) = \sum_s Y^s \ln P(Y = 1|X^s, W) + (1 - Y^s) \ln P(Y^s = 0|X^s, W)$$

since Y^s can take only values 0 or 1, only one of the two terms in the expression will be non-zero for any given Y^s. This can be rewritten to:

$$l(W) = \sum_s Y^s \ln \frac{P(Y^s = 1|X^s, W)}{P(Y^s = 0|X^s, W)} + \ln P(Y^s = 0|X^s, W)$$

and by plugging in the parametric formulas for P to:

$$l(W) = \sum_s Y^s (w_0 + \sum_i^n w_i X_i^s) - \ln(1 + \exp(w_0 + \sum_i^n w_i X_i^s))$$

Logistic Regression: learning the parameters (3)

Unfortunately, there is no closed form solution maximizing $l(W)$ with respect to W. Fortunately, the $l(W)$ function is **concave** with respect to W. Therefore, a natural approach is to use the gradient ascent search in the space of partial derivatives of $l(W)$. The ith component of the vector gradient has the form:

$$\frac{\partial l(W)}{\partial w_i} = \sum_s X_i^s (Y^s - \hat{P}(Y^s = 1 | X^s, W))$$

where $\hat{P}(Y^s | X^s, W)$ is the Logistic Regression prediction of the probability. To accommodate weight w_0, we assume an imaginary $X_0 = 1$ for all s.

This expression for the derivative has an intuitive interpretation: the term inside the parentheses is simply the prediction error; that is, the difference between the observed Y^s and its predicted probability! Note if $Y^s = 1$ then we wish for $\hat{P}(Y^s = 1 | X^s, W)$ to be 1, whereas if $Y^s = 0$ then we prefer that $\hat{P}(Y^s = 1 | X^s, W)$ be 0 (which makes $\hat{P}(Y^s = 0 | X^s, W)$ equal to 1). This error term is multiplied by the value of X_i, which accounts for the magnitude of the $w_i X_i$ term in making this prediction.

Logistic Regression: learning the parameters (4)

To conduct the standard gradient ascent search to optimize the weights W, we may begin with initial weights of zero, and repeatedly update the weights in the direction of the gradient, on each iteration changing every weight w_i according to:

$$w_i \leftarrow w_i + \eta \sum_s X_i^s (Y^s - \hat{P}(Y^s = 1 | X^s, W))$$

where η is a small constant (e.g., 0.01) which determines the step size. Because the conditional log likelihood $l(W)$ is a concave function in W, this gradient ascent procedure will converge to a global maximum.

Logistic Regression: Regularization

Regularization is the technique used in many statistical algorithms do combat overfitting, which is possible especially with high-dimensional but sparse data. Regularization works by introducing an additional term in the formulas to penalize large values of W, on the assumption that large coefficients appear in highly overfitted functions. One possible approach is to penalize the log likelihood function:

$$W \leftarrow \arg\max_W \sum_s \ln P(Y^s|X^s, W) - \frac{\lambda}{2}|W|^2$$

by adding a penalty proportional to the squared magnitude of W. λ determines the strength of this penalty term. Maximizing this goal corresponds to calculating the MAP (Maximum A Posteriori) estimate for W under the assumption of a normal (Gaussian) distribution for $P(W)$ with a zero mean and variance (σ^2) related to $\frac{1}{\lambda}$.

It is not hard to derive the following gradient ascent update rule, similar to the case with no regularization:

$$w_i \leftarrow w_i + \eta \sum_s X_i^s (Y^s - \hat{P}(Y^s = 1|X^s, W)) - \eta\lambda w_i$$

Logistic Regression: Regularization(2)

The above procedure uses the normal (Gaussian) model for the distribution $P(W)$, which leads to a so-called **L_2 regularization**. It tries to minimize the square norm $|W|^2$.

Another way to regularize is the L_1 regularization, which minimizes $|W|$.

Logistic Regression: multivalued functions

In the case of a multivalued function where Y has a number of discrete values $y_1, ..., y_K$, the form of $P(Y = y_k|X)$ for $Y = y_1, Y = y_2, ..., Y = y_{K-1}$ is:

$$P(Y = y_k|X) = \frac{\exp(w_{k0} + \Sigma_{i=1}^{n} w_{ki} X_i)}{1 + \Sigma_{j=1}^{K-1} \exp(w_{j0} + \Sigma_{i=1}^{n} w_{ji} X_i)}$$

and for the final $Y = y_K$:

$$P(Y = y_K|X) = \frac{1}{1 + \Sigma_{j=1}^{K-1} \exp(w_{j0} + \Sigma_{i=1}^{n} w_{ji} X_i)}$$

Here w_{ji} denotes the weight associated with the jth class $Y = y_j$ and with input X_i. It is easy to see that our earlier expressions for the case where Y is boolean are a special case of the above expressions. Note also that the form of the expression for $P(Y = y_K|X)$ assures that $[\Sigma_{k=1}^{K} P(Y = y_k|X)] = 1$.

The primary difference between these expressions and those for boolean Y is that when Y takes on K possible values, we construct $K - 1$ different linear expressions to capture the distributions for the different values of Y. The distribution for the final, Kth, value of Y is simply one minus the probabilities of the first $K - 1$ values.

Logistic Regression: multivalued functions (2)

In the case of a multivalued function, the gradient ascent rule becomes:

$$w_{ji} \leftarrow w_{ji} + \eta \sum_s X_i^s (\delta(Y^s = y_j) - \hat{P}(Y^s = y_j | X^s, W))$$

where $\delta(Y^s = y_j) = 1$ if the sth training value, Y^s, is equal to y_j, and $\delta(Y^s = y_j) = 0$ otherwise. Note our earlier learning rule is a special case of this new learning rule, when $K = 2$. As in the case for $K = 2$, the quantity inside the parentheses can be viewed as an error term which goes to zero if the estimated conditional probability $P(Y^s = y^j | X^s, W))$ perfectly matches the observed value of Y^s.

With regularization the above formula changes to:

$$w_{ji} \leftarrow w_{ji} + \eta \sum_s X_i^s (\delta(Y^s = y_j) - \hat{P}(Y^s = y_j | X^s, W)) - \eta \lambda w_{ji}$$

The Confusion Matrix

We have seen that Accuracy/Error can be the single basic learning efficiency metric. We have also seen how they do not give a complete view of the performance of the trained classifier.

It is desirable to have a reliable measure of the classification performance. Unfortunately, no single statistical error measure is universally accepted. Therefore, a number of such measures are defined and are in use. Most of them can be computed from the structure called the **confusion matrix** representing the number of samples from each class classified as each other class. In the general case it has the form:

	classified as class:			
	1	2	...	m
true class: 1	$N_{1,1}$	$N_{1,2}$	...	$N_{1,m}$
true class: 2	$N_{2,1}$	$N_{2,2}$	...	$N_{2,m}$
...	...	...	...	...
true class: m	$N_{m,1}$	$N_{m,2}$	...	$N_{m,m}$

We can see that: Accuracy $= \frac{\sum_i N_{i,i}}{\sum_i \sum_j N_{i,j}}$

The Confusion Matrix for Binary Case

In the binary classification case, the classifier should simply select the „positive" samples, and leave out the negatives. The following terminology is used:

$$\begin{aligned} \text{TP } (\textit{true positives}) &= N_{1,1} \\ \text{TN } (\textit{true negatives}) &= N_{0,0} \\ \text{FN } (\textit{false negatives}) &= N_{1,0} \\ \text{FP } (\textit{false positives}) &= N_{0,1} \end{aligned}$$

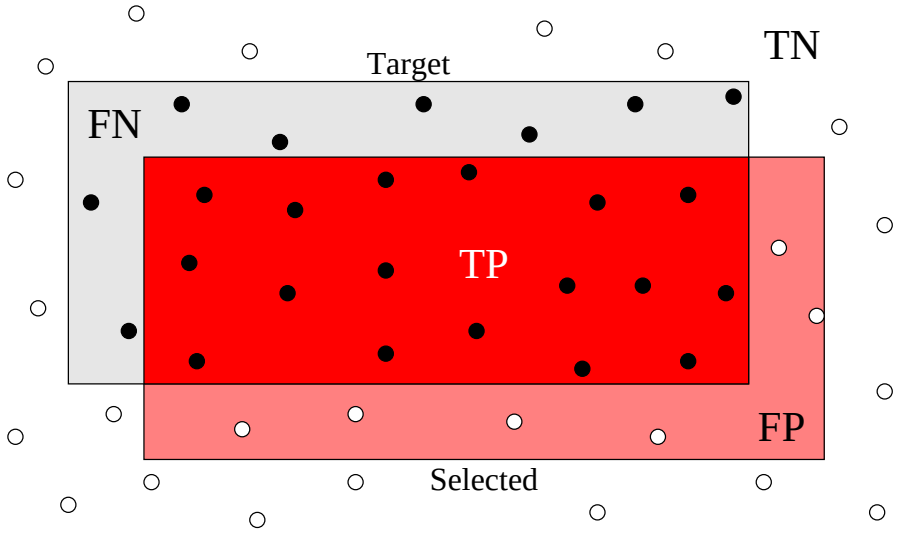
	classified as:	
	class 0	class 1
true class 0	TN	FP
true class 1	FN	TP

We can see that:

$$\text{Accuracy} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}}$$

Precision and Recall

The **Precision** is the error measure originating from the binary classification and expressing the fraction of examples indicated by the classifier as positive, which are indeed positive. The **Recall** is the fraction of positive examples which are recognized as such by the classifier.



$$\text{Precision} = \frac{TP}{TP+FP}$$

$$\text{Recall} = \frac{TP}{TP+FN}$$

A typical compromise between Precision and Recall is that trying to maximize one might result in reducing the other, and vice versa. Consider a brain surgeon removing a cancerous tumor from a patient's brain (an example from Wikipedia). He needs to remove all of the tumor since any remaining cancer cells will regenerate the tumor. However, he must not remove healthy tissue since that might leave the patient impaired. Removing more liberally, the surgeon increases recall but reduces precision. Proceeding more conservatively, he increases precision, but reduces recall.

Precision and Recall (cntd.)

Note that the Precision and Recall are computed just for the class of positive samples; it describes the process of selecting some examples from the population. If the selection of negative examples are equally important, we may compute the Precision and Recall for the negative class.

In the general multiple class case there are separate Precision or Recall for each class:

$$\text{Precision(C)} = \frac{N_{C,C}}{\sum_i N_{i,C}} \qquad \text{Recall(C)} = \frac{N_{C,C}}{\sum_i N_{C,i}}$$

A classification Precision score of 1.0 for class C means that every item classified as class C does indeed belong to class C, but says nothing about members of class C that were not classified correctly.

A Recall of 1.0 for class C means that all its members have been correctly classified as C, but says nothing about how many other items were incorrectly also labeled as belonging to class C.

We can compute average Precision or Recall weighted by class membership.

The $\hat{\kappa}$ (Kappa) statistic

The Kappa parameter is referred to as the **interrater agreement**. It reflects the agreement between two ways of classifying objects. In fact, it provides a measure how much better the agreement between two classifiers is than random. The formula:

$$\hat{\kappa} = \frac{NT \, \Sigma_i \, N_{i,i} - \Sigma_i (NR_i \times NC_i)}{NT^2 - \Sigma_i (NR_i \times NC_i)}$$

where:

$$\begin{aligned} NT &= \sum_i \sum_j N_{i,j} \quad \text{(total samples)} \\ NR_i &= \sum_j N_{i,j} \quad \text{(total samples per row } i\text{)} \\ NC_i &= \sum_j N_{j,i} \quad \text{(total samples per column } i\text{)} \end{aligned}$$

The value of Kappa may range from -1 to 1, but for positive correlation it is > 0. Values above 0.5 indicate a significantly correct classification.

Also in use is a weighted version of Kappa with weights selected for each class.

Neighborhood in the feature space

In general, having many historical points in the feature space, and the new point with an unknown class, we may examine its neighborhood in the space.

Then we locate the nearest neighbor of the new point, and use its class for the new point. This is the **nearest neighbor** algorithm.

Computing distance in the feature space

A good question is how we measure the „nearest". We need a distance metric for features. It is typical to use the Euclidean distance:

$$D(x_i, x_k) = \sqrt{\sum_j (x_{i_j} - x_{k_j})^2}$$

However, the dimensions of the feature space correspond to different features, which may be different quantities, expressed in different units, perhaps of different orders of magnitude.

Scaling the inputs

To bring the different feature space dimensions to be comparable, scaling may be used. One typical approach is to compute the mean value $\overline{x}$, and standard deviation σ_x of each input feature x, and then rescale such feature x as:

$$x' = \frac{x - \overline{x}}{\sigma_x}$$

This universal form of scaling is called **normalization**. Instead, one may use some other scaling scheme, for example to boost the influence of some feature relative to others. This can be obtained experimentally, by the gradient descent (hill-climbing) procedure with cross-validation, to get the feature weight vector which performs best on some specific data set.

The Bankruptcy example again

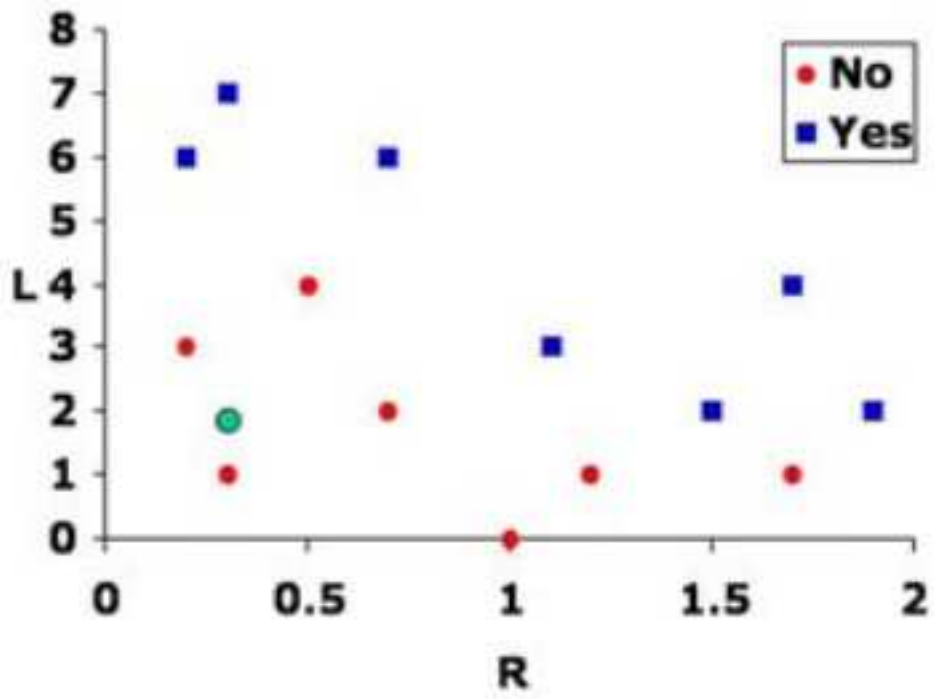
Suppose some such procedure has been performed for the bankruptcy data set considered earlier, and the result was that the R feature (rate of spending versus income) should be scaled by 5, relative to L (number of late monthly payments in a year).

$$D(x_i, x_k) = \sqrt{(L_{x_i} - L_{x_k})^2 + (5R_{x_i} - 5R_{x_k})^2}$$

(This roughly corresponds to the scale of the axes in the image below.)

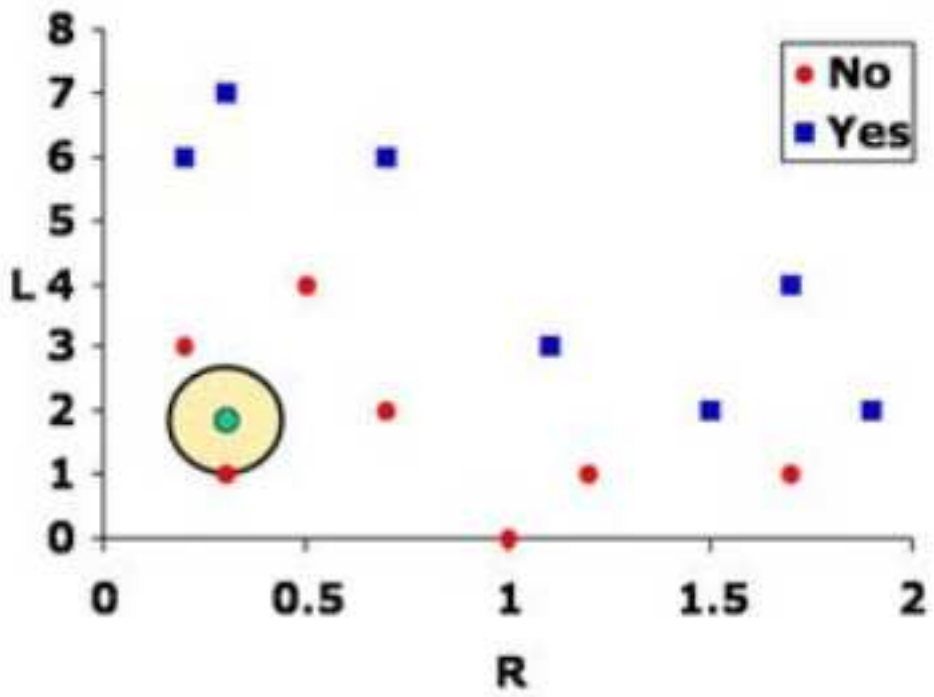
Suppose now there is a new person to classify with $R = 0.3$ and $L = 2$.

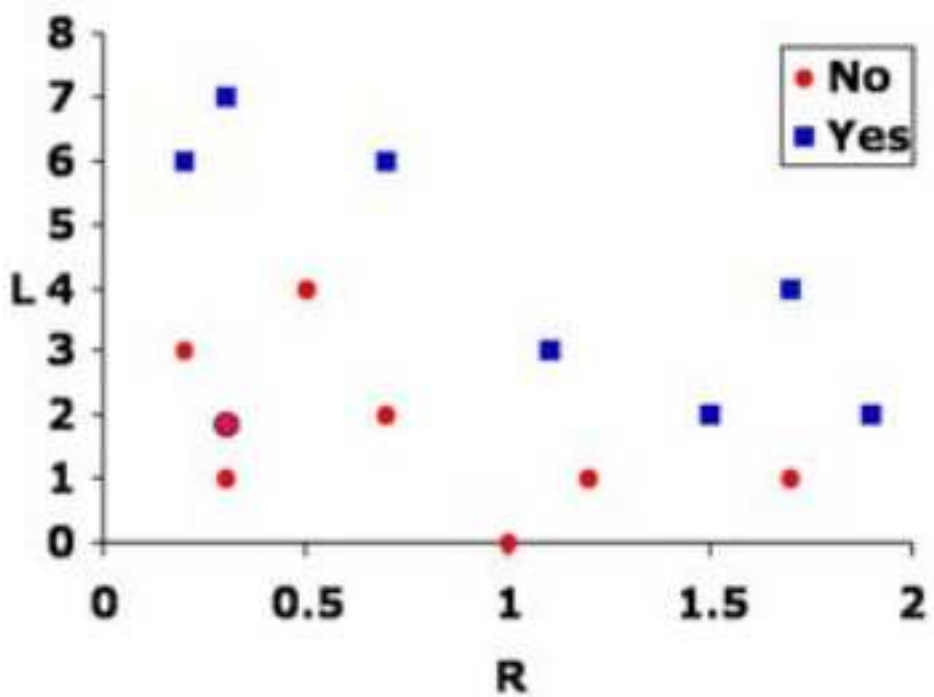
What class value should we predict?



The Bankruptcy example continued

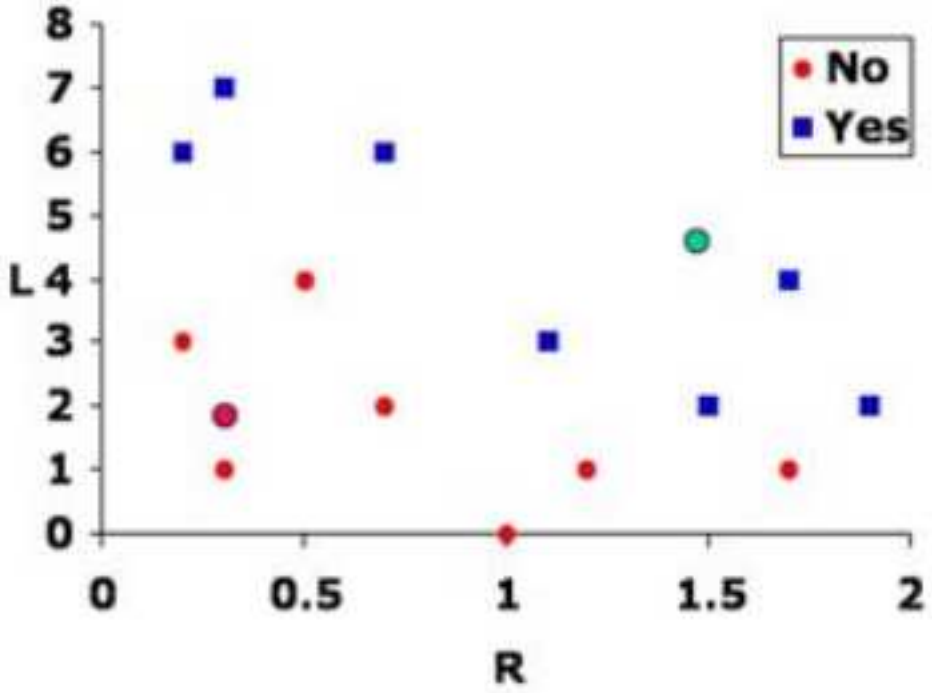
We may compute the nearest neighbor ...

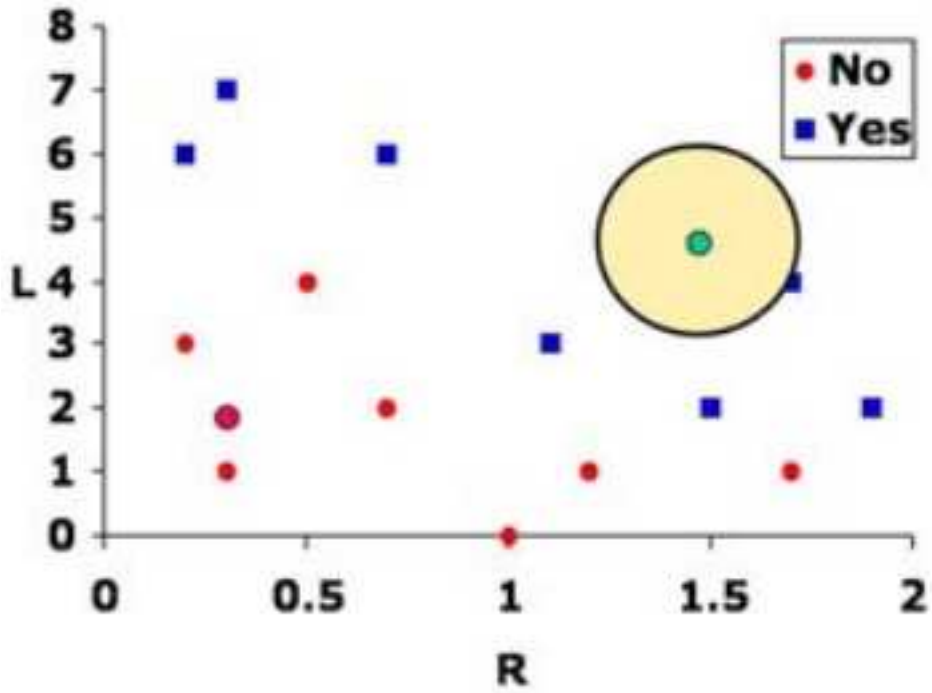


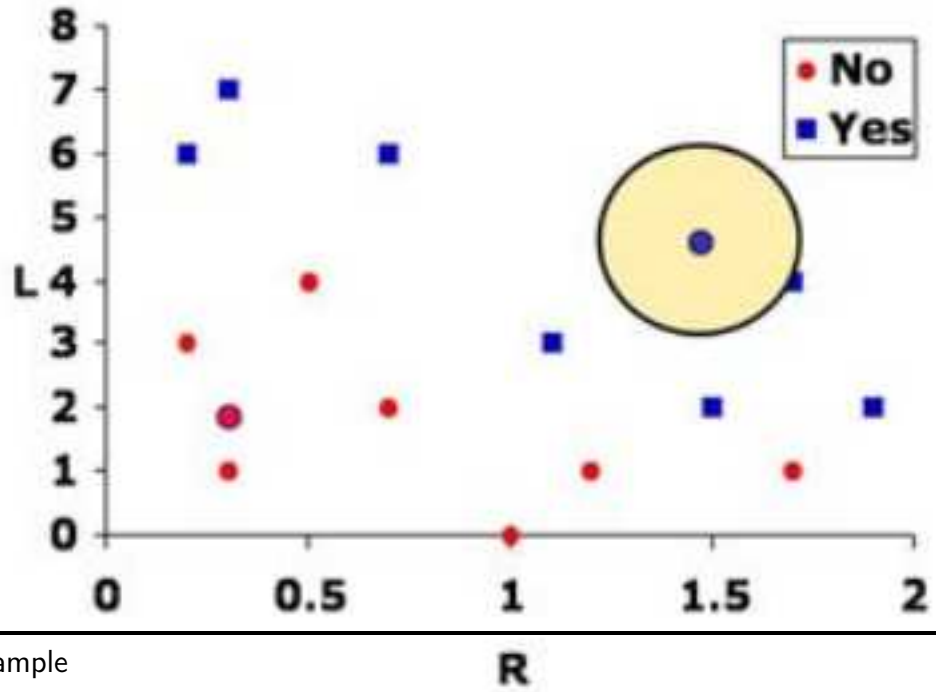


... and conclude that the new person's class should be red (No bankruptcy).

And likewise for another new sample:

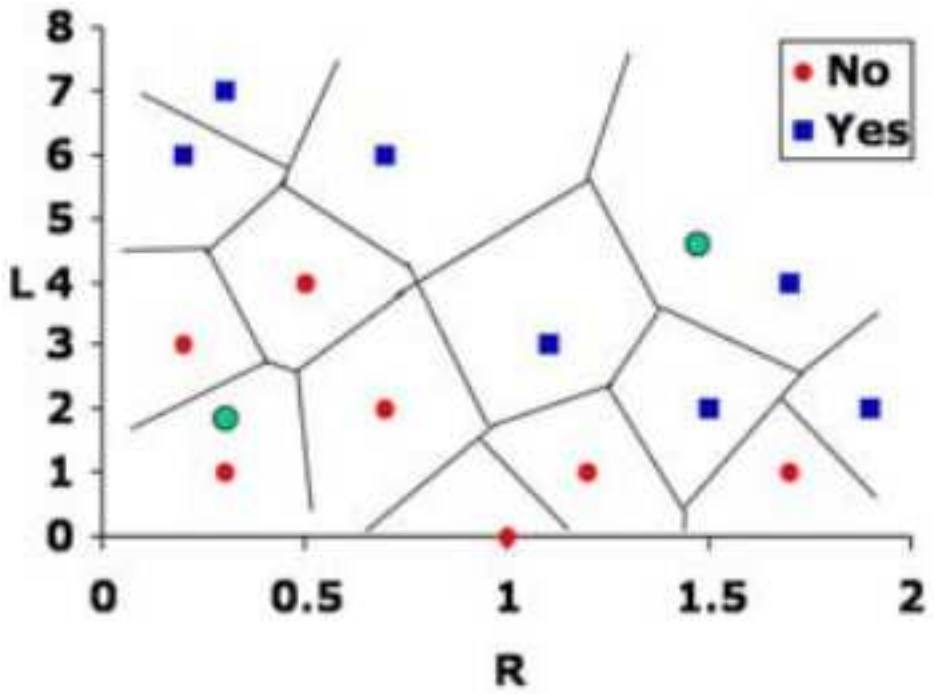






The nearest neighbor classification hypothesis

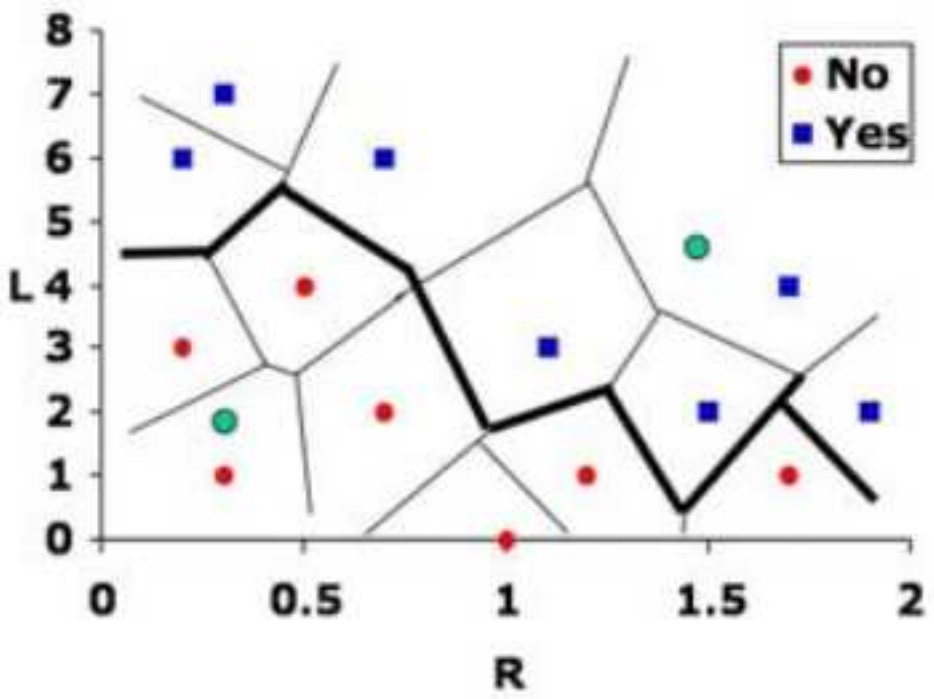
The nearest neighbor algorithm outlined here effectively divides the feature space into regions:



The diagram dividing the (two-dimensional) space with line segments equi-distant from the set of points is called a **Voronoi diagram**.

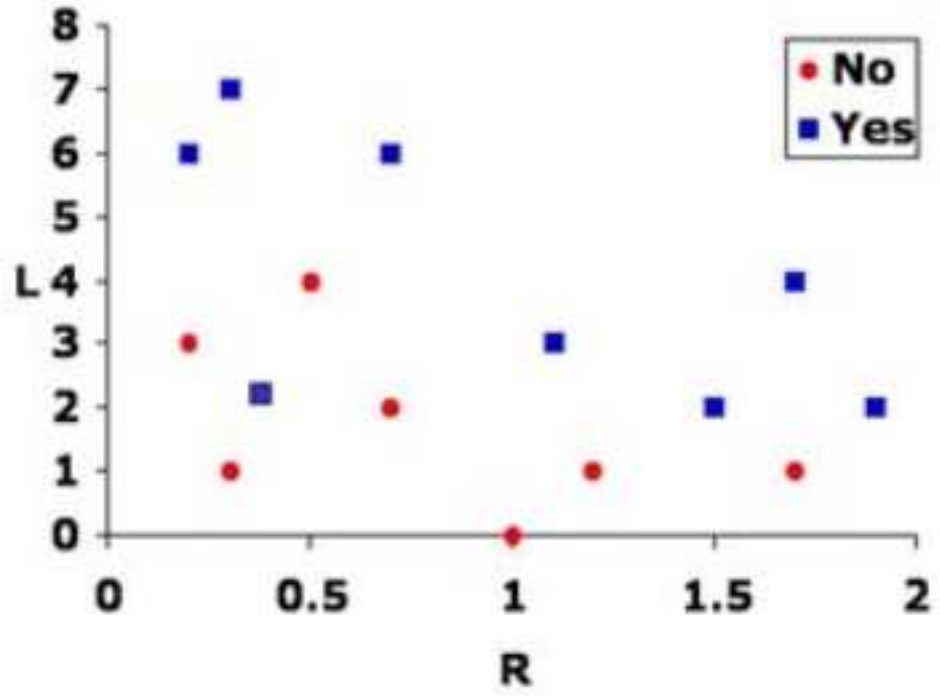
Time and Space

Remembering all the samples from the dataset, which may be huge, and then calculating the distance from all these points for any new sample, may be intensive. A clever data structure (K-D tree) allows the calculations to be fast, on the average $O(\log(m) * n)$ with n features and m samples in the training set. Training samples far from the boundary lines (or k-surfaces) can also be forgotten, reducing the memory consumption.



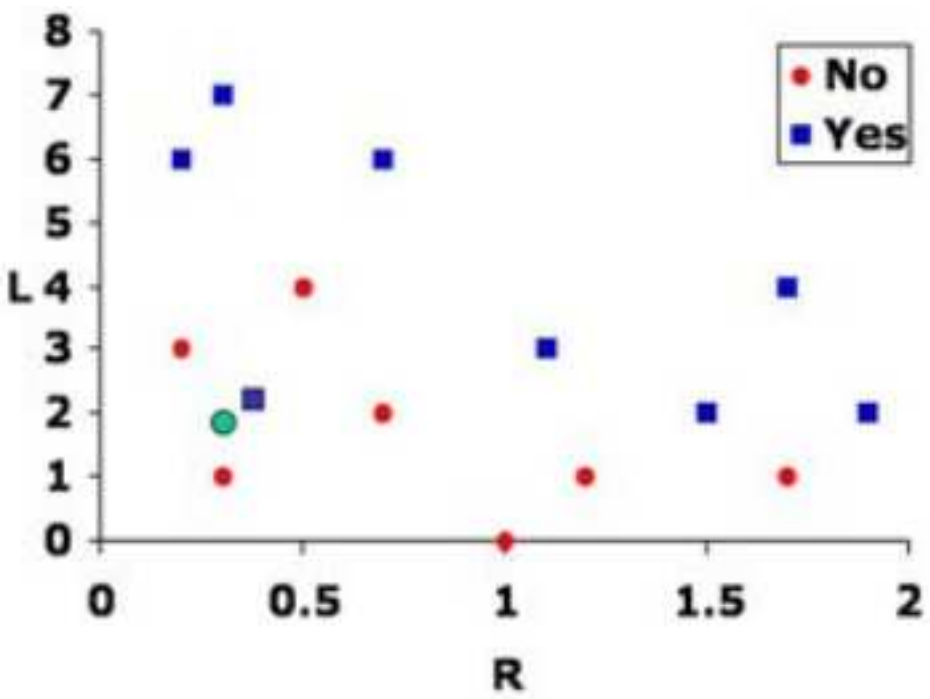
Noise

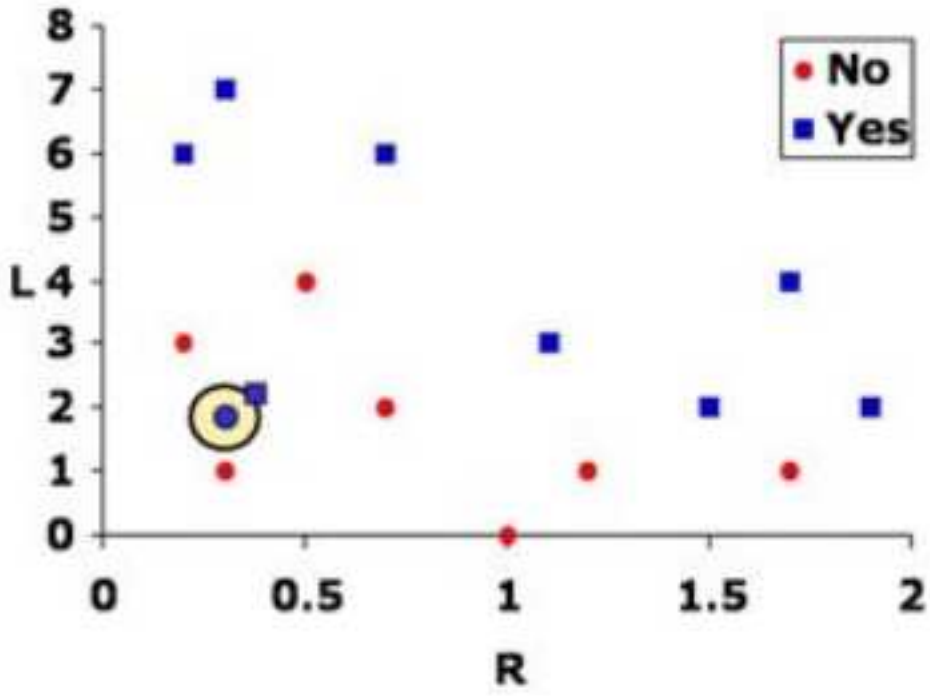
When the training dataset is reliable, the boundary between the two output classes are clear. However, problems arise when a person with good financial parameters had gone bankrupt.



Such point may be treated in two ways: either being strictly observed, and allowed to affect future classification, or treated as an unusual case, which should be ignored. However, in a real dataset, with many features and many values, we may not be able to spot such such training set members.

Noise — classifying a new sample

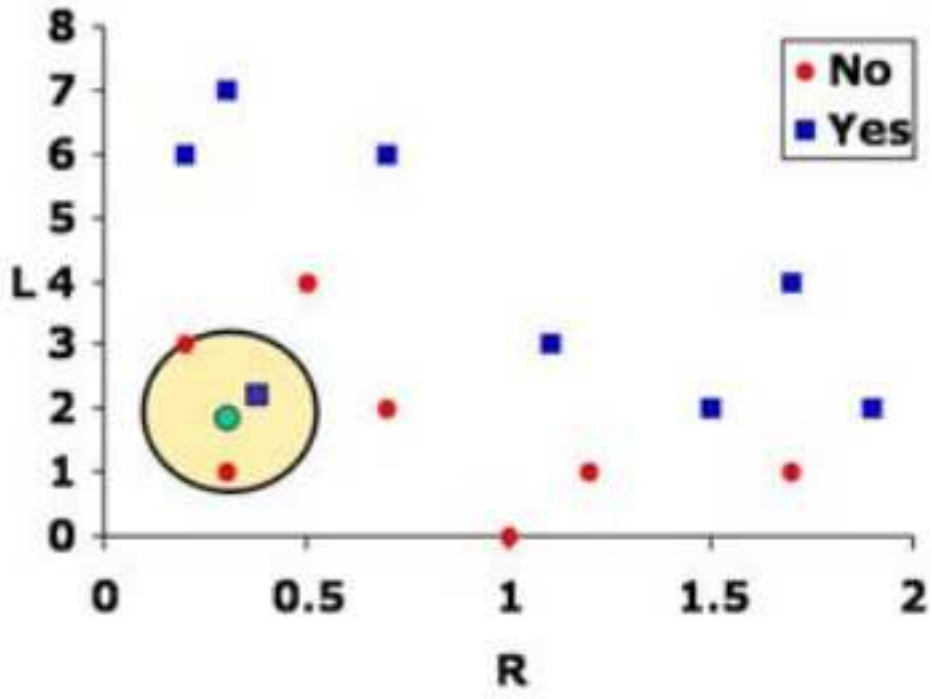


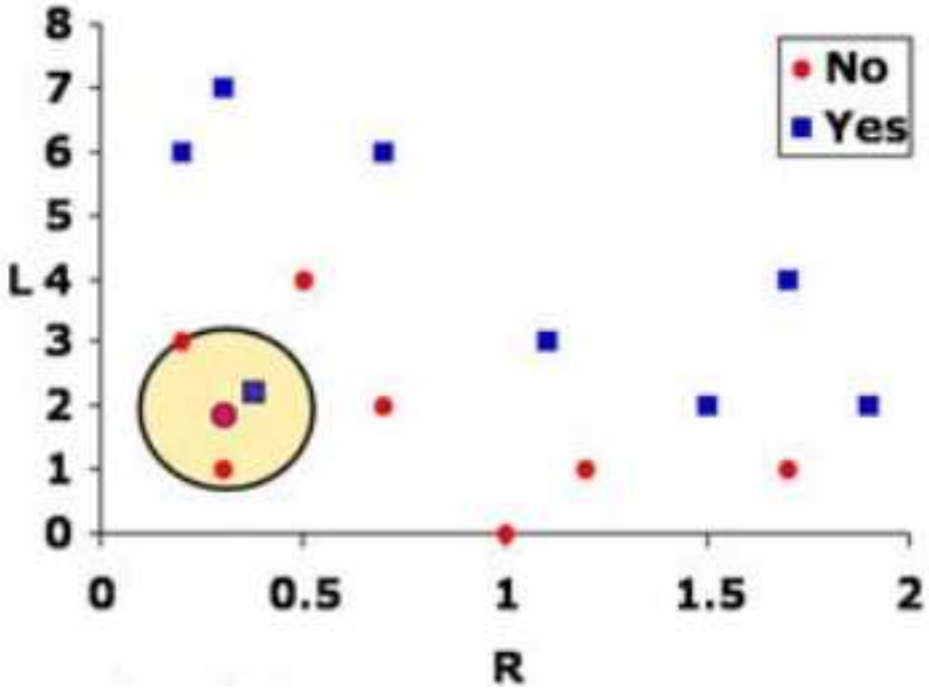


The standard nearest neighbor algorithm may classify the new point based on the „noise" training sample.

k-Nearest Neighbors algorithm

The simple extension of the algorithm takes into account some number k of nearest points, and use the class of the majority of the elements.





With larger k values the classifier becomes more robust and resistant to noise, but less precise. The optimal k values can be determined experimentally using cross-validation. For obvious practical reasons, it makes sense to use odd k values: 3, 5, 7, 9, ...

The curse of dimensionality

The nearest neighbor algorithm works very well and is one of the most powerful classification machine learning algorithms.

A limiting factor for it are the very high dimensional feature spaces. In high dimensions, almost all points are far apart, if not under one then under another dimension. This is called **the curse of dimensionality**, a phenomenon often seen in machine learning work.

The curse of dimensionality can be illustrated as following. Assume the $\mathcal{D}$-dimensional space is a unit cube with uniformly distributed training samples. To classify a sample we uniformly grow a hyper-cube around it until it contains a desired fraction p of the training set. The expected edge length of this cube will be $e_{\mathcal{D}}(p) = p^{1/\mathcal{D}}$.

For $\mathcal{D}$=10 and p=1% we have $e_{10}(0.01) = 0.63$, with the unit cube. This means that the nearest neighbor method is not really local, since it requires consulting so distant neighbors (practically in the other half of the hyper-cube). Such points may not be good predictors for the class of the point being classified.

The curse of dimensionality — dimension reduction

Because of this, it is often desirable to reduce the dimensionality of the space, by feature reduction. This may seem counter-productive, since the many features may carry important information about the samples. However, this is in line with the Ockham's razor principle: if the shorter feature vector works equally well (or almost), then we should use it. It will give faster learning, more robust classification, and permit easier visualization or interpretation of the classification process.

Ensemble Learning

The classification machine learning algorithms presented so far produced a single model for making predictions for the class of any new data samples. They could be optimized to obtain the best performance using various criteria.

Then systematic empirical comparisons showed that the best learning algorithm varies from application to application. Therefore it makes sense to try several (many) approaches for any new machine learning project. Initially, effort went into trying many variations of many algorithms, and still selecting just the best one.

But more recently it was observed that, if instead of selecting the single best variation found, it is better to combine many variations, and use the combination of their results by: averaging, voting, or by another level of machine learning.

This approach is called **ensemble learning**. There are some variants of it, which we will now examine.

Ensemble Learning — Bagging

Bagging, also called **bootstrap aggregating**, generates multiple (K) random variations of the training set by resampling (duplicating and dropping random samples), trains a classifier on each, and combines the results by voting.

Bagging often significantly reduces variance, especially on a noisy or limited dataset. It also works well when the base model is found to be overfitting. While at the same time the bias can rise, this is usually limited.

Bagging can be used with any base learning model, but is most commonly used with decision trees. This is because this model is inherently unstable: a slightly different set of examples can lead to a completely different decision tree. Bagging smoothes out the variance which can appear.

Another advantage of this approach is that bagging can be easy to run in parallel on many computers.

Ensemble Learning — Random Forest

Often the result of using bagging with decision trees results in K trees that are highly correlated. This happens because an attribute with a very high information gain is likely to become the root of most of the trees.

Therefore special techniques can be applied to make the K trees more diverse, to reduce variance. This approach is called the **random forest**.

The key idea of random forest is to vary the attribute choices. In selecting the root attribute (and also at each split point in recursive construction of the tree), we do not consider all of the attributes, but a selected subset. One rule is to select $\sqrt{n}$ from n attributes for the classification problems, and $n/3$ for regression problems.

Unlike the basic decision trees, which are prone to overfitting, and require pruning, random forests are often used unpruned, and yet they are resistant to overfitting. As the number of trees in the forest grows, the validation-set error rates tend to improve. There is also a theoretical proof for this effect for almost all cases.

Random forests are one of the most popular algorithm applied to a wide variety of application domains.

Ensemble Learning — Stacking

In **stacking**, multiple classifiers are generated, just like in bagging, and the outputs of individual classifiers become the inputs of a "higher-level" classifier trained to make a final prediction.

Whereas bagging aggregates the "votes" of base classifiers — which all represent the same machine learning algorithm — by simple majority counting, here the final decision can be learned based on the full input data vectors, augmented with the classification results from the base classifiers. It makes therefore sense to utilize different machine learning algorithms for them, since the upper level algorithm will have to learn to use them anyway.

The logistic regression model is often used for the upper level in stacking, although it can be any classification algorithm, either used in the base level as well, or not.

The term "stacking" reflects the scheme of putting the ensemble model on top of the layer of base models. It is even possible to stack multiple layers, each subsequent one operating on the output of the previous layer.

Ensemble Learning — Boosting

The most popular ensemble learning method is called **boosting**. In boosting, the classifiers are generated sequentially, and the training samples have weights, initially all equal. After training the first classifier, call it hypothesis h_1, the data samples classified by it incorrectly have their weights increased, while those classified correctly have them decreased.

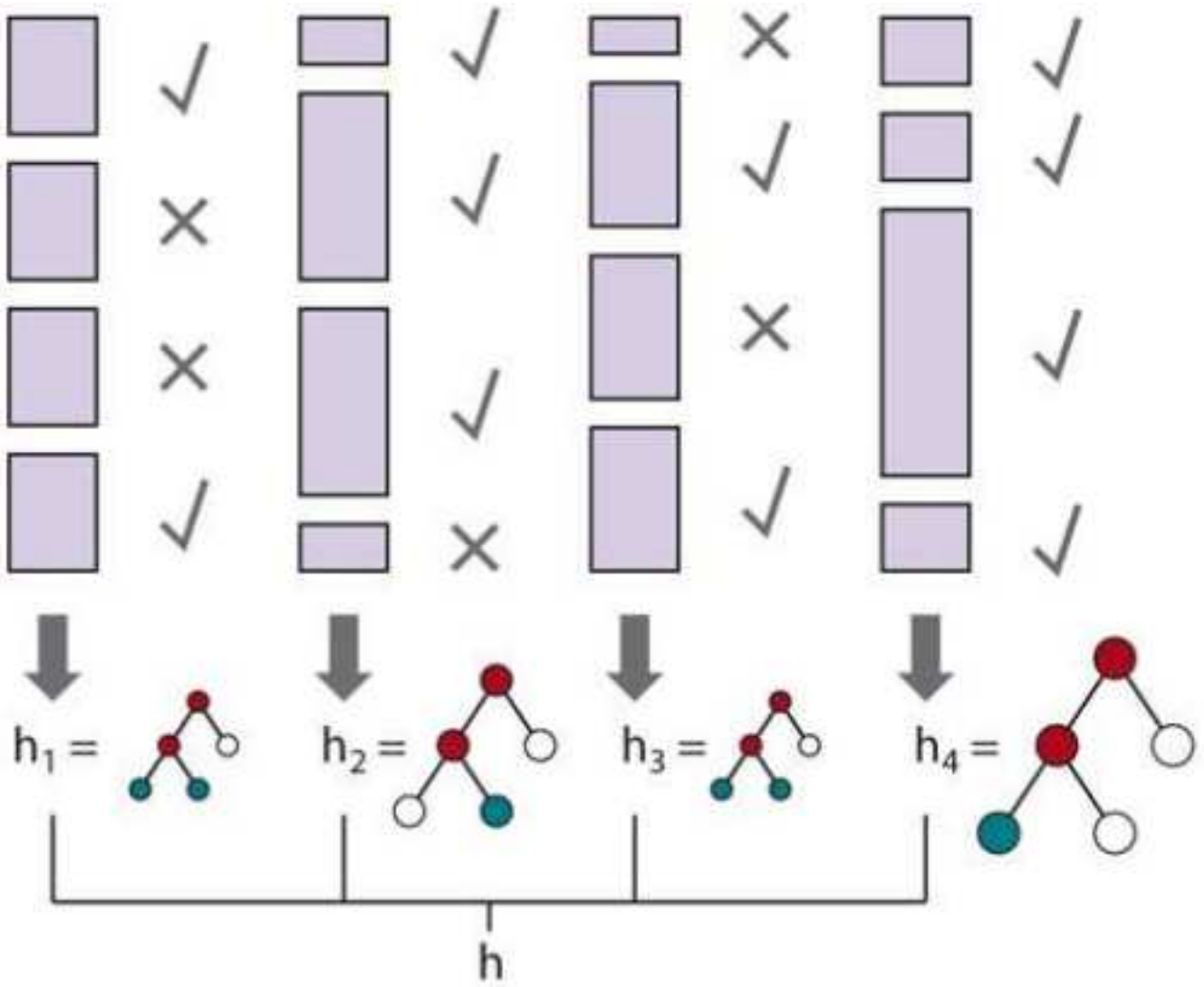
Using this modified training set, the new hypothesis h_2 is generated. The process continues to generate K hypotheses, where K is a parameter to the boosting algorithm. The examples that are difficult to classify will get increasingly larger weights, until the algorithm is forced to generate a hypothesis which classifies them correctly.

Boosting is a greedy algorithm, as it does not backtrack. Even if it ever generates a hypothesis which is worse than the previous one, the algorithm only goes forward, trying to fix it.

The final ensemble uses all generated hypotheses in a weighted voting system, with the weights increasing for the hypothesis which obtained better results on their weighted training set.

It works well particularly in situations when the generated classifiers exhibit low variance and high bias.

Below is the illustration of the boosting process. Each shaded rectangle corresponds to a data sample, with its height reflecting its weight. The checks and crosses indicated whether the sample was classified correctly by the current hypothesis. The size of the decision tree indicates the weight of that hypothesis in the final ensemble.



Ensemble Learning — AdaBoost

The **AdaBoost** algorithm was one of the first examples of this approach, and remains very popular. AdaBoost is usually applied with decision trees as base models.

A very important property of AdaBoost is that even if the base learning algorithm performs only marginally better than random guessing (eg. $50\% + \epsilon$ in the binary classification case), then AdaBoost generates the ensemble that classifies the training data perfectly, for large enough K.

In other words, boosting can overcome any amount of bias in the base model, as long as this model is ϵ better than random guessing.

Ensemble Learning — Gradient Boosting

Another variant of boosting is called **gradient boosting**. Unlike original boosting, in which we pay increasingly more attention to the examples which the previous hypothesis got wrong, in gradient boosting we generate new hypotheses based on the gradient between the right answers and the answers given by the previous hypotheses.

Just as is done in other algorithms based on gradient descent, we start by a differentiable loos function, such as logarithmic loss for classification, or squared error for regression. We then build a decision tree and use gradient descent to update the parameters to build the next one.

Summary: classification machine learning

The machine learning methods presented so far allow creating classifiers. A classifier is a system that inputs a vector of discrete and/or continuous feature values and outputs a single discrete value, the class. The learning algorithm inputs a training set of examples, and produces a classifier. The test of the learner is whether this classifier gives the correct output for future examples.

Learning algorithms consists of combinations of just three components:

Representation: choosing the set of classifiers that it can possibly learn. This set is called the **hypothesis space** of the learner. If a classifier is not in the hypothesis space, it cannot be learned.

Evaluation: An evaluation function (also called objective function or scoring function) is needed to distinguish good classifiers from bad ones. The evaluation function used internally by the algorithm may differ from the external one that we want the classifier to optimize.

Optimization: a method to search among the classifiers for the highest-scoring one. The choice of optimization technique is key to the efficiency of the learner.

Learning = Representation + Evaluation + Optimization

Representation	Evaluation	Optimization
Instances	Accuracy/Error rate	Combinatorial optimization
K-nearest neighbors	Precision and recall	Greedy search
Support vector machines	Squared error	Beam search
Hyperplanes	Likelihood	Branch-and-bound
Naive Bayes	Posterior probability	Continuous optimization
Logistic regression	Information gain	Unconstrained
Decision trees	K-L divergence	Gradient descent
Sets of rules	Cost/Utility	Conjugate gradient
Propositional rules	Margin	Quasi-Newton methods
Logic programs		Constrained
Neural networks		Linear programming
Graphical models		Quadratic programming
Bayesian networks		
Condition.random fields		

Not all combinations of one component from each column make equal sense. For example, discrete representations naturally go with combinatorial optimization, and continuous ones with continuous optimization. Nevertheless, many learners have both discrete and continuous components.

It's generalization that counts

The key idea of machine learning is generalization. No matter how much data is available to the program developer, the program will likely have to operate on still different data.

Think: for 100,000 words in a dictionary, the hypothetical spam classifier may face $2^{100,000}$ different inputs.

Because machine learning algorithms have many parameters by which optimization can be achieved, it is important to separate training and testing data, so that the results of this fine-tuning are not evaluated on the training data.

So it is normal in machine learning, that we are optimizing a different function than the one we have access to.

Data alone is not enough

We know we need to generalize from examples, but how to do it? There are $2^{100,000}$ possible inputs for the hypothetical spam classifier, and even if we have a training database of millions of emails, it is still only a very small fraction of all the possibilities.

How is the learner supposed to figure out the outcome of all the unseen samples? The shortest answer is: knowledge.

Even very simple additional assumptions about the type of the hypothesis function — like smoothness, similar samples having similar classes, limited dependencies, or limited complexity — are often enough for effective learning.

In fact, one of the main criteria for choosing a representation is which kinds of problem knowledge are easily expressed in it.

Overfitting has many facets

What if the knowledge and data we have are not sufficient to completely determine the correct classifier? Then we run the risk of creating a classifier that is not grounded in reality, and is simply encoding random quirks in the data.

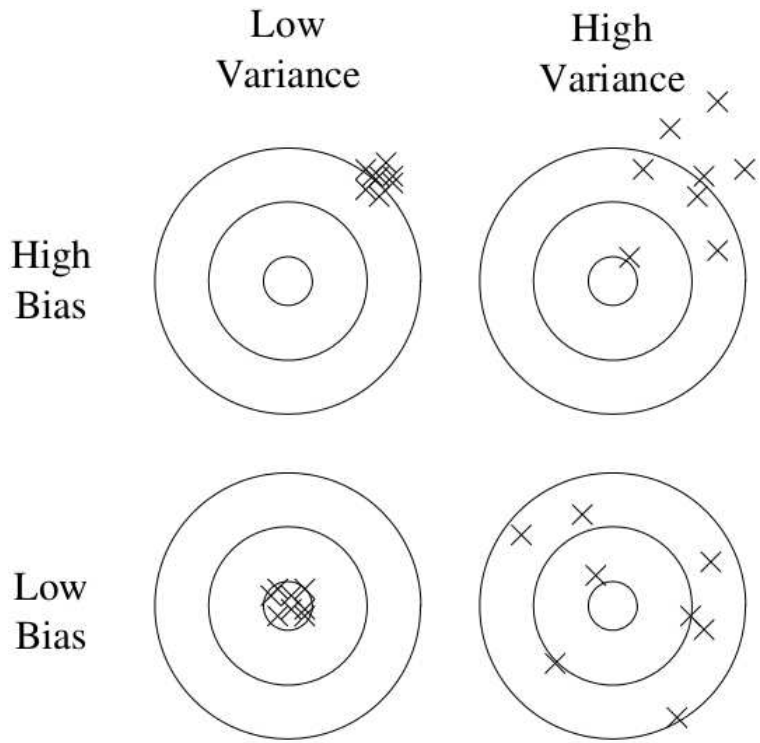
This problem is called overfitting, and is one of the biggest headaches of machine learning. When the learner produces a classifier that is 100% accurate on the training data but only 50% accurate on test data, when in fact it could have produced one that is 75% accurate on both, then we know it has overfit.

Bias and variance

One way to understand overfitting is by decomposing the generalization error into bias and variance.

Bias is a learner's tendency to consistently learn the same wrong thing.

Variance is the tendency to learn random things irrespective of the real signal.

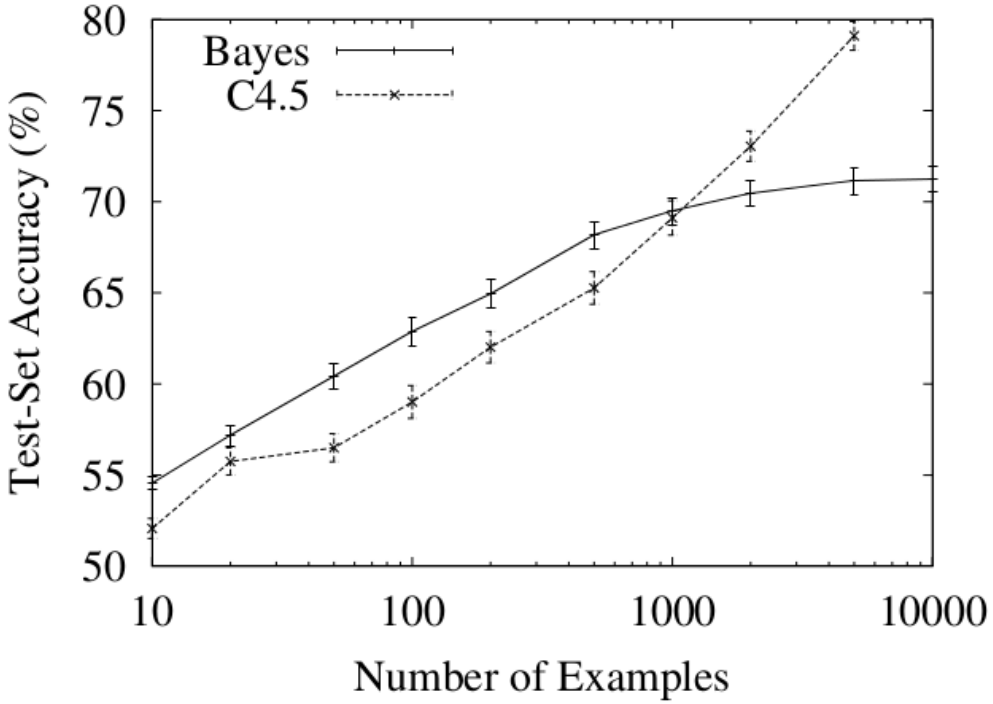


A linear learner has high bias, because when the frontier between two classes is not a hyperplane the learner is unable to induce it. Decision trees don't have this problem because they can represent any Boolean function, but on the other hand they can suffer from high variance: decision trees learned on different training sets generated by the same phenomenon are often very different, when in fact they should be the same.

Similar reasoning applies to the choice of optimization method: beam search has lower bias than greedy search, but higher variance, because it tries more hypotheses.

More powerful is not necessarily better

This figure illustrates an experiment in which data originally classified with a rule-based classifier have subsequently been machine learned using a basic Naive Bayes and state-of-the-art rule C4.5rules algorithms.



Even though the original classifier is a set of rules, with up to 1000 examples naive Bayes is more accurate than a rule learner. This happens despite naive Bayes's false assumption that the frontier is linear! Situations like this are common in machine learning: strong false assumptions can be better than weak true ones, because a learner with the latter needs more data to avoid overfitting.

Problems with multi-dimensional spaces

After overfitting, the biggest problem in machine learning is the curse of dimensionality. Its basic meaning is that many algorithms that work fine in low dimensions become intractable when the input is high-dimensional.

But in machine learning it refers to much more. Generalizing correctly becomes exponentially harder as the dimensionality (number of features) of the examples grows, because a fixed-size training set covers a dwindling fraction of the input space. Even with a moderate dimension of 100 and a huge training set of a trillion examples, the latter covers only a fraction of about 10^{-18} of the input space!

Problems with multi-dimensional spaces (2)

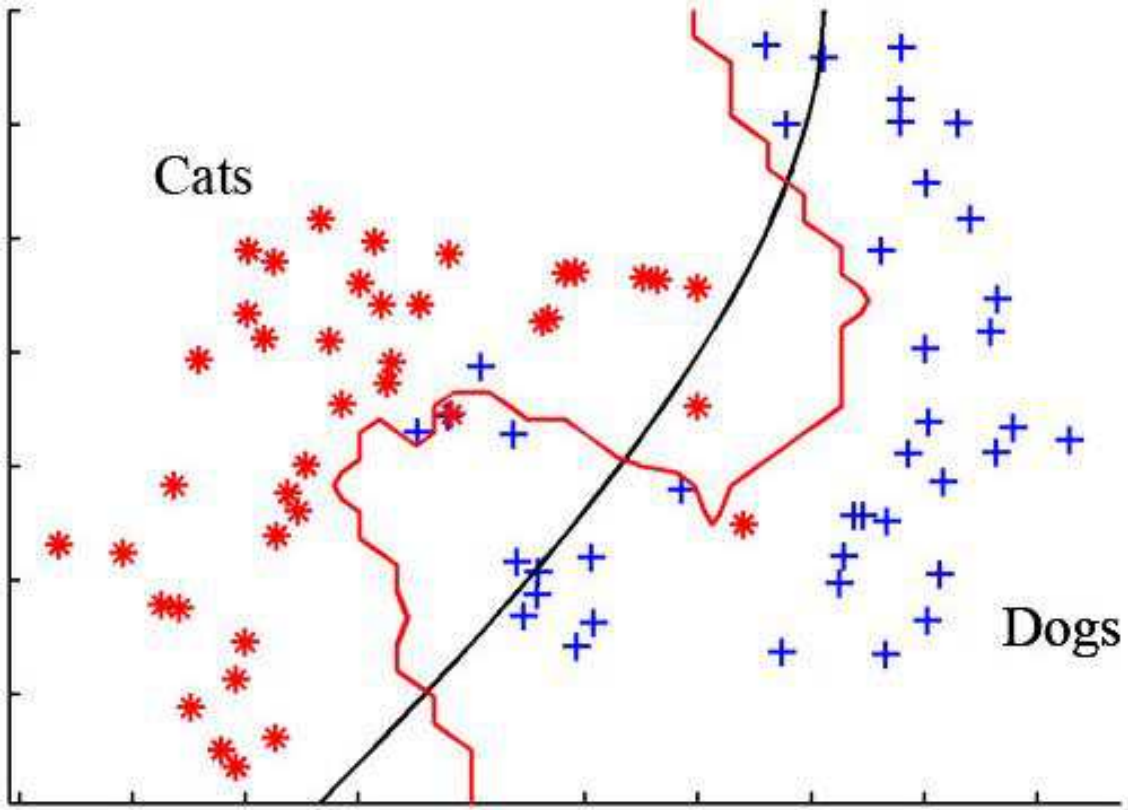
The similarity-based reasoning that machine learning algorithms depend on (explicitly or implicitly) breaks down in high dimensions. Consider a nearest neighbor classifier with Hamming distance as the similarity measure, and suppose the class is just $x_1 \wedge x_2$. If there are no other features, this is an easy problem. But if there are 98 irrelevant features $x_3, ..., x_{100}$, the noise from them completely swamps the signal in x_1 and x_2, and nearest neighbor effectively makes random predictions.

Even more disturbing is that nearest neighbor still has a problem even if all 100 features are relevant! This is because in high dimensions all examples look alike. Suppose, for instance, that examples are laid out on a regular grid, and consider a test example x_t. If the grid is d-dimensional, x_t's $2d$ nearest examples are all at the same distance from it. So as the dimensionality increases, more and more examples become nearest neighbors of x_t , until the choice of nearest neighbor (and therefore of class) is effectively random.

Intuition fails in high dimensions

Building a classifier in two or three dimensions is easy. We can find a reasonable frontier between examples of different classes just by visual inspection.

(It's even been said that if people could see in high dimensions, machine learning would not have been necessary.)



In high dimensions it's hard to understand what is happening. This in turn makes it difficult to design a good classifier. Naively, one might think that gathering more features never hurts, since at worst they provide no new information about the class. But in fact their benefits may be outweighed by the curse of dimensionality.

Feature engineering is the key

If there is the single most important factor which can make a machine learning project is a success or a failure then it is the set of features used. If we have many independent features that each correlate well with the class, learning is easy. If the class is a very complex function of the features, we may not be able to learn it.

Often the raw data are not in a form that is amenable to learning, but we can construct features from it that are. This is another place where we can bring knowledge into the process, and where most effort in a machine learning project can be spent. It is often also one of the most interesting parts, where intuition, creativity, and "black art" are as important as the technical stuff.

Feature engineering is the key (2)

The feature engineering process can be automated, to some degree, by automatically generating large numbers of candidate features and selecting the best by their information gain with respect to the class. But features that look irrelevant in isolation may be relevant in combination. For example, if the class is an XOR of k input features, each of them by itself carries no information about the class.

On the other hand, running a learner with a very large number of features to find out which ones are useful in combination may be too time-consuming, or cause overfitting. So there is ultimately no replacement for the smarts we put into feature engineering.

More data beats a cleverer algorithm

Suppose we have constructed the best set of features we can, but the classifiers we are getting are still not accurate enough. What can we do now?

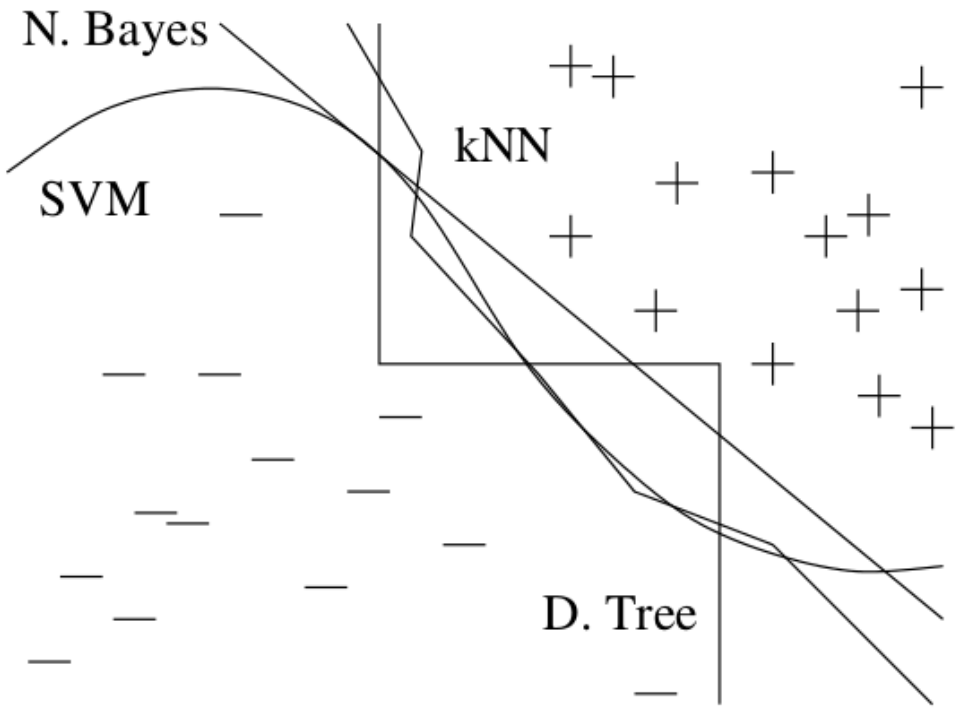
There are two main choices: design a better learning algorithm, or gather more data: more examples, and possibly more raw features, subject to the curse of dimensionality. Machine learning researchers are mainly concerned with the former, but pragmatically the quickest path to success is often to just get more data. As a rule of thumb, a dumb algorithm with lots and lots of data beats a clever one with modest amounts of it. After all, this is what machine learning is all about — letting data do the heavy lifting.

More data beats a cleverer algorithm (2)

Part of the reason why using cleverer algorithms has a smaller payoff than we might expect is that, to a first approximation, they all do the same. All learners essentially work by grouping nearby examples into the same class; the key difference is in the meaning of "nearby."

With non-uniformly distributed data, learners can produce widely different frontiers while still making the same predictions in the regions that matter (those with a substantial number of training examples, and therefore also where most test examples are likely to appear).

The effect is much stronger in high dimensions.



A stronger algorithm is not always better

Trying to start machine learning with the "strongest" methods is usually not the best strategy. There is no substitute for independently conducting initial experiments, and looking for ways to improve the result.

As a rule, it pays to try the simplest learners first, eg.:

- naive Bayes before logistic regression
- k-nearest neighbors before support vector machines
- ...

More sophisticated learners are seductive, but they are usually harder to use, because they have more knobs you need to turn to get good results, and because their internals are more opaque.

We should turn to them when we are satisfied that we explored all options with the simpler ones, and have the resources necessary for the deeper exploration: time and skill.

Useful Resources

Parts of the following resources have been used in this presentation:

1. Stuart J. Russell, Peter Norvig: Artificial Intelligence A Modern Approach (Fourth Edition), Pearson, 2021

2. Ian H. Witten, Eibe Frank, Mark A. Hall: Data Mining Practical Machine Learning Tools and Techniques, Third Edition, Morgan Kaufman, 2011

3. Kevin P. Murphy: Machine Learning A Probabilistic Perspective, MIT Press, 2012 (images made available by the MIT Press)

4. Leslie Kaelbling, Tomás Lozano-Pérez: M.I.T. 6.034 Artificial Intelligence Spring 2005 `http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-034-artificial-intelligence-spring-2005/lecture-notes/`

5. Tom M. Mitchell: Generative and Discriminative Classifiers: Naive Bayes and Logistic Regression, draft chapter intended for inclusion in the upcoming second edition of the textbook Machine Learning, 2017 `http://www.cs.cmu.edu/~tom/mlbook/NBayesLogReg.pdf`

6. Pedro Domingos: A Few Useful Things to Know about Machine Learning, Communications of the ACM, 2012