Chapter 13
Induction in Multi-Label Domains

All the techniques discussed in the previous chapters assumed that each example is labeled with one and only one class. In realistic applications, however, this is not always the case. Quite often, an example is known to belong to two or more classes at the same time, sometimes to *many* classes. For machine learning, this poses certain new problems. After a brief discussion of how to deal with this issue within the framework of classical paradigms, this chapter describes the currently most popular approach: *binary relevance*.

The idea is to induce a binary classifier separately for each class, and then to use all these classifiers in parallel. More advanced versions of this technique seek to improve classification performance by exploiting mutual interrelations between classes. As yet another alternative, the chapter discusses also the simple mechanism of *class aggregation*.

13.1 Classical Machine Learning in Multi-Label Domains

Let us begin with an informal definition of a multi-label domain. After this, we will take a look at how to address the problem within the classical paradigms introduced in earlier chapters.

What is a Multi-Label Domain? In many applications, the traditional requirement that an example should be labeled with one and only one class is hard to satisfy. Thus a text document may represent `nutrition`, `diet`, `athletics`, `popular science`, and perhaps quite a few other categories. Alternatively, an image may at the same time represent `summer`, `cloudy weather`, `beach`, `sea`, `seagulls`, and so on. Something similar is the case in many other domains.

The number of classes with which an average example is labeled differs from one application to another. In some of them, almost every example has a great many labels selected from perhaps thousands of different classes. At the other end of the

M. Kubat, *An Introduction to Machine Learning*,
DOI 10.1007/978-3-319-63913-0_13

spectrum, we find domains where only some examples belong to more than one class, the majority being labeled with only a single one.

Whatever the characteristics of the concrete data, the task for machine learning is to induce a classifier (or a set of classifiers) satisfying two basic requirements. First, the tool should for a given example return as many of its true classes as possible; missing any one of them would constitute a *false negative*. At the same time, the classifier should not label the example with a class to which the example does not belong—each such "wrong" class would constitute a *false positive*.

Neural Networks Chapter 5 explained the essence of a *multilayer perception*, MLP, a popular architecture of artificial neural networks. The reader will recall that the output layer consists of one neuron for each class, the number of inputs equals the number of attributes, and the ideal size of the hidden layer reflects the complexity of the classification problem at hand.

On the face of it, using an MLP in multi-label domains should not pose any major problems. For instance, suppose the network has been presented with a training example that is labeled with classes C_3, C_6, and C_7. In this event, the target values for training will be set to, say, $t_i = 0.8$, in the case of output neurons with indices $i \in \{3, 6, 7\}$, and to $t_i = 0.2$ for all the other output neurons.[1] The backpropagation-of-error technique can then be used in the same manner as in single-label domains.

A Word of Caution Multilayer perceptions may not necessarily be the best choice here. Indeed, multi-label domains have been less intensively studied, in the neural-networks literature, than other approaches, and not without reason. For one thing, the training of plain MLPs is known to be vulnerable to local minima, and there is always the architecture-related question: what is the best number of hidden neurons if we want to strike a reasonable compromise between overfitting the data if the network is too large, and suffering from insufficient flexibility if the network is too small?

Also the notoriously high computational costs can be a reason for concern. The fact that each training example can belong to more than one class certainly complicates the learning process. Sensing the difficulty of the task, the engineer is sometimes tempted to increase the number of hidden neurons. This, however, not only adds to the already high computational costs, but also increases the danger of overfitting.

It is always good to keep in mind that training neural networks is more art than science. While a lot can be achieved through ingenuity and experience, beginners are often disappointed. And in the case of a failure, the machine-learning expert should be prepared to resort to some alternative, less dangerous technique.

[1]The reader will recall that the target values 0.8 and 0.2 are more appropriate for the backpropagation-of-error algorithm than 1 and 0. See Chap. 5.

Nearest-Neighbor Classifiers Another possibility is the use of nearest-neighbor classifiers with which we got acquainted in Chap. 3. When example $\mathbf{x}$ is presented, the k-NN classifier first identifies the example's k nearest neighbors. Each of these may have been labeled with a set of classes, and the simplest classification attempt in a multi-label domain will label $\mathbf{x}$ with the union of these sets. For instance, suppose that $k = 3$, and suppose that the sets of class labels encountered in the three nearest neighbors are $\{C_1, C_2\}$, $\{C_2\}$, and $\{C_1, C_3\}$, respectively. In this event, the classifier will classify $\mathbf{x}$ as belonging to C_1, C_2, and C_3.

A Word of Caution This approach is practical only in domains where the average number of classes per example is moderate, say, less than three. Also the number of voting neighbors, k, should be small. Unless these two requirements are satisfied, too many class labels may be returned for $\mathbf{x}$, and this can give rise to too many false positives, which, in turn, leads to poor *precision*. At the same time, however, the multitude of returned labels also reduces the number of false negatives, which improves *recall*. In some domains, this is what we want. In others, *precision* is critical, and its low value may not be acceptable.

As so often in this paradigm, the engineer must resist the temptation to increase the number of the nearest neighbors in the hope that spreading the vote over more "participants" will give a chance to less frequent classes. The thing is, some of these "nearest neighbors" might then be too distant from $\mathbf{x}$, and thus inappropriate for classification purposes.

A Note on Other Approaches Machine learning scientists have developed quite a few other ways of modifying traditional machine-learning paradigms for the needs of multi-label domains. Among these, very interesting appear to be attempts to induce multi-label decision trees. But since they are somewhat too advanced for an introductory text, we will not present them here. After all, comparable classification performance can be achieved by simpler means—and these will be the subject of the rest of this chapter.

What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- Suggest an example of a multi-label domain. What is the essence of the underlying machine-learning task? In what way will one multi-label domain differ from another?
- Explain the simple method of multi-label training in multilayer perceptions. What practical difficulties might discourage you from using this paradigm?
- Describe the simple way of addressing a multi-label domain by a k-NN classifier. Discuss its potential pitfalls.

13.2 Treating Each Class Separately: Binary Relevance

Let us now proceed to the main topic of this section, the technique of *binary relevance*. We will begin by explaining the principle, and then discuss some of its shortcomings and limitations.

The Principle of Binary Relevance The most common approach to multi-label domains induces a separate binary classifier for each class: in a domain with N classes, N classifiers are induced. When classifying a future example, all these classifiers are used in parallel, and the example receives all classes for which the classifiers returned the positive label.

For the induction of these classifiers, the training data have to be modified accordingly. Here is how. For the i-th class ($i \in [1, N]$), we create a training set, T_i, that consists of the same examples as the original training set, T, the only difference being in labeling: in T_i, an example's class label is 1 if the list of class labels for this example in T contains C_i; otherwise, the label in T_i is 0.

Once the new training sets have been created, we apply to each of them a *baseline learner* that is responsible for the induction of the individual classifiers. Common practice applies the same baseline learner to each T_i. Typically, we use to this end some of the previously discussed machine-learning techniques such as perception learning, decision-tree induction, and so on.

Illustration of the Learning Principle Table 13.1 illustrates the mechanism with which the new training data are created. In the original training set, T, five different class labels can be found: $C_1, \ldots, C_5$. The *binary relevance* technique creates the five new training sets, $T_1, \ldots, T_5$, shown in the five tables below the original one.

Table 13.1 The original multi-label training set is converted into five new training sets, one for each class

	Classes
ex_1	C_1, C_2
ex_2	C_2
ex_3	C_1, C_3, C_5
ex_4	C_2, C_3
ex_5	C_2, C_4

T_1	
ex_1	1
ex_2	0
ex_3	1
ex_4	0
ex_5	0

T_2	
ex_1	1
ex_2	1
ex_3	0
ex_4	1
ex_5	1

T_3	
ex_1	0
ex_2	0
ex_3	1
ex_4	1
ex_5	0

T_4	
ex_1	0
ex_2	0
ex_3	0
ex_4	0
ex_5	1

T_5	
ex_1	0
ex_2	0
ex_3	1
ex_4	0
ex_5	0

Thus in the very first of them, T_1, examples $\mathtt{ex}_1$ and $\mathtt{ex}_3$ are labeled with 1 because these (and only these) two examples contain the label C_1 in the original T. The remaining examples are labeled with 0.

The baseline learner is applied separately to each of the five new sets, inducing from each T_i the corresponding classifier C_i.

An Easy-to-Overlook Pitfall In each of the training sets thus obtained, every example is labeled as a positive or negative representative of the given class. When the induced binary classifiers are used in parallel (to classify some $\mathbf{x}$), it may happen that none of them returns 1. This means that no label for $\mathbf{x}$ has been identified. When writing the machine-learning software, we must not forget to instruct the classifier what to do in this event. Usually, the programmer chooses from the following two alternatives: (1) return a default class, perhaps the one most frequently encountered in T, or (2) reject the example as too ambiguous to be classified.

Discussion The thing to remember is that the idea behind *binary relevance* is to transform the multi-label problem into a set of single-label tasks that are then addressed by classical machine learning. To avoid disappointment, however, the engineer needs to be aware of certain difficulties which, unless properly addressed, may lead to underperformance. Let us briefly address them.

Problem 1: Imbalanced Classes Some of the new training sets, T_i, are likely to suffer from the problem of *imbalanced class* representation which was discussed in Sect. 10.2. In Table 13.1, this occurs in the case of sets T_4 and T_5. In each of them, only one example out of five (20%) is labeled as positive, and all others are labeled as negative. In situations of this kind, we already know, machine-learning techniques tend to be biased toward the majority class—in this particular case, the class labeled as 0.

The solution is not difficult to find. The two most straightforward approaches are majority-class undersampling or minority-class oversampling. Which of them to choose will of course depend on the domain's concrete circumstances. As a rule of thumb, one can base the decision on the size of the training set. In very big domains, majority-class undersampling is better; but when the examples are scarce, the engineer cannot afford to "squander" them, and thus prefers minority-class oversampling.

Problem 2: Computational Costs Some multi-label domains are very large. Thus the training set in a text categorization domain may consist of hundreds of thousands of examples, each described by tens of thousands of attributes and labeled with a subset of thousands of different classes. It stands to reason that to induce thousands of decision trees from a training set of this size will be expensive, perhaps prohibitively so. We can see that, when considering candidates for the baseline learner, we may have to reject some of them because of computational costs.

Another possibility is to resort to the technique discussed in Sect. 9.5 in the context of boosting techniques: for each class, we create multiple subsets of the training examples, some of them perhaps described by different subsets of attributes. The idea is to induce for each class a group of subclassifiers that then vote. If (in

a given paradigm) learning from 50% of the examples takes only 5% of the time, considerable savings can be achieved.

Problem 3: Performance Evaluation Another question is how to measure the success or failure of the induced classifiers. Usually, each of them will exhibit different performance, some better than average, some worse than average, and some dismal. To get an idea of the big picture, some averaging of the results is needed. We will return to this issue in Sect. 13.7.

Problem 4: Neglecting Mutual Interdependence of Classes The baseline version of *binary relevance* treats all classes as if they were independent of each other. Quite often, this assumption is justified. In other domains, the classes *are* to some degree interdependent, but not much harm is done when this fact is ignored. But in some applications, the overall performance of the induced classifiers considerably improves if we find a way to exploit the class interdependence. To introduce methods of doing so will be the task for the next three sections.

What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- Describe the principle of *binary relevance*. How does it organize the learning process, and how are the induced classifiers used for the classification of future examples?
- What can render the computational costs of this approach prohibitively high? How will the engineer handle the situation?
- Why does *binary relevance* often lead to the problem of imbalanced classes? What remedies would you recommend?

13.3 Classifier Chains

In many applications, the classes are interrelated. The fact that a text document has been labeled as `nutrition` is sure to increase its chances of belonging also to `diet`—and decrease the probability that has something to do with `quantum mechanics`. In the context of binary relevance, this means that methods of exploiting class interdependence are likely to improve classification performance. One possibility of doing so is known as a *classifier chain*.

The Idea A very simple approach relies on a chain of classifiers such as the one in Fig. 13.1. Here, the classifiers are created one at a time, starting from the left. To begin with, the leftmost classifier is induced from the original examples labeled as positive or negative instances of class C_1 (recall the training set T_1 from Table 13.1).

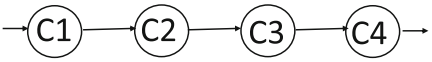

Fig. 13.1 With the exception of $C1$, the input of each classifier consists of the original attribute vector *plus* the label returned by the previous classifier

The second classifier is then induced from examples labeled as positive or negative instances of class C_2. To describe these latter examples, however, one extra attribute is added to the original attribute vector: the output of C_1. The same principle is then repeated in the course of the induction of all the remaining classifiers: for each, the training examples are described by the original attribute vector *plus* the class label returned by the previous classifier.

When using the classifier chain for the classification of some future example, **x**, the same pattern is followed. The leftmost classifier receives **x** described by the original attributes. To all other classifiers, the system presents **x** described by the original attribute vector plus the label delivered by the previous classifier. Ultimately, **x** is labeled with those classes whose classifiers returned 1.

An Important Assumption (Rarely Satisfied) In the classifier-chain technique, the ordering of the classes from left to right is the responsibility of the engineer. In some applications, this is easy because the classes form a logical sequence. Thus in document classification, `science` subsumes `physics`, which in turn subsumes `quantum mechanics`, and so on. If a document does not belong to `science`, it is unlikely to belong to `physics`, either; it thus makes sense to choose `science` as the leftmost node in the graph in Fig. 13.1, and to place `physics` next to it.

In other applications, class subsumption is not so obvious, but the sequence can still be used without impairing the overall performance. Even when the subsumptions are only intuitive, the engineer may always resort to a sequence backed by experiments: she can suggest a few alternative versions, test them, and then choose the one with the best results. Another possibility is to apply the classifier chain only to some of the classes (where the interrelations are known), treating the others as if only plain *binary relevance* was to be employed.

Hierarchically Ordered Classes Class interrelation does not have to be linear. It can acquire forms that can only be reflected by a more sophisticated data structure, perhaps a graph. In that case, we will need more advanced techniques such as the one described in Sect. 13.5.

One Shortcoming of the Classifier-Chain Approach More often than not, the engineer lacks any a priori knowledge about class interrelations. If she then still wants to employ classifier chains, the best she can do is to create the classifier sequence randomly. Of course, such ad hoc method cannot be guaranteed to work; to insist on an inappropriate classifier sequence may be harmful to the point where the classification performance of the induced system may fail to reach even that of plain *binary relevance*.

Sometimes, however, there is a way out. If the number of classes is manageable (say, five), the engineer may choose to experiment with several alternative sequences, and then choose the best one. But if the number of classes is greater, the necessity to try many alternatives will be impractical.

Error Propagation The fact that the classifiers are forced into a linear sequence makes them vulnerable to a phenomenon known as *error propagation*. Here is what it means. When a classifier misclassifies an example, the incorrect class label is passed on to the next classifier that uses this label as an additional attribute. An incorrect value of this additional attribute may then sway the next classifier to a wrong decision, which, too, is passed on, down the chain. In other words, an error of a single class may result in additional errors being made by subsequent classifiers. In this event, the *classifier chain* is likely to underperform. The thing to remember is that the overall error rate strongly depends on the quality of the earlier classifiers in the sequence.

One Last Comment The error-propagation phenomenon is less damaging if the classifiers do not strictly return either 0 or 1. Thus a Bayesian classifier calculates for each class its probability, a number from the interval $[0, 1]$. Propagating this probability through the chain is less harmful than the strict **pos** or **neg**. Similar considerations apply to some other classifiers such as those from the paradigm of neural networks (for instance, multilayer perceptions and radial-basis function networks).

What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- Discuss the typical impact that class interdependence can have on the performance of the *binary relevance* technique.
- Explain the principle of *classifier chain*. What can you say about the need to find a proper sequence of classifiers?
- Explain the problem of *error propagation* in classifier chains. Is there anything else to criticize about this approach?

13.4 Another Possibility: Stacking

In the light of the aforementioned drawbacks of classifier chains, some less risky alternative is needed. One possibility is to rely on *stacking*.

Architecture and Induction The essence is illustrated in Fig. 13.2. Here, the classifiers are arranged in two layers. The upper one represents plain *binary*

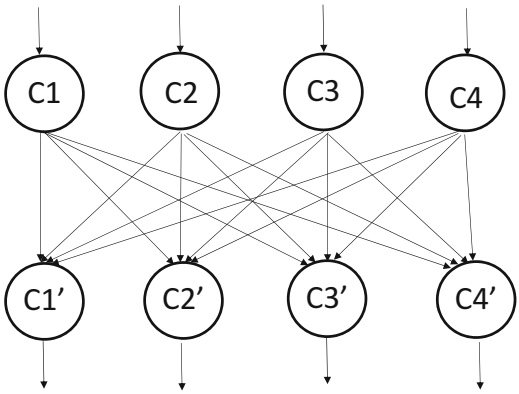

Fig. 13.2 The *stacking* technique. The upper-layer classifiers use as input the original attribute vector. For the lower-layer classifiers, this vector is extended by the vector of the class labels returned by the upper layer

relevance (independently induced binary classifiers, one for each class). More interesting is the bottom layer. Here, the classifiers are induced from the training sets where the original attribute vectors are extended by the list of the class labels returned by the upper-layer classifiers. In the concrete case depicted in Fig. 13.2, each attribute vector is preceded by four new binary attributes (because there are four classes): the i-th attribute has value 1 if the i-th classifier in the upper layer has labeled the example as belonging to class i; otherwise, this attribute's value is 0.

Classification When the class labels of some future example $\mathbf{x}$ are needed, $\mathbf{x}$ is presented first to the upper-layer classifiers. After this, the obtained classes labels are added at the front of $\mathbf{x}$'s original attribute vector as N new binary attributes (assuming there are N classes), and the newly described example is presented in parallel to the lower-layer classifiers. Finally, $\mathbf{x}$ is labeled with the classes whose lower-layer classifiers have returned 1.

The underlying philosophy rests on the intuition that the performance of classifier C_i may improve if this classifier is informed about the "opinions" of the other classifiers—about the other classes to which $\mathbf{x}$ belongs.

An Example Consider an example that is described by a vector of four attributes with these values: $\mathbf{x} = \{a, f, r, z\}$. Suppose that the upper-layer classifiers return the following labels: $C_1 = 1, C_2 = 0, C_3 = 1, c_4 = 0$. In this event, the lower-layer classifiers are all presented with the following example description: $\mathbf{x} = \{1, 0, 1, 0, a, f, r, z\}$.

The classification behaviors of the lower-level classifiers can differ from those in the upper layer. For example, if the lower-layer classifiers return 1, 1, 1, 0, the overall system will label $\mathbf{x}$ with C_1, C_2, and C_3, ignoring the original recommendations of the upper layer.

Some Comments Intuitively, this approach is more flexible than classifier chains because *stacking* makes it possible for any class to influence the recognition of any other class. The engineer does not provide any a priori information about class

interdependence, assuming that the class interdependence (or the lack thereof) is likely to be discovered in the course of the learning process.

When treated dogmatically, however, this principle may do more harm than good. The fact that $\mathbf{x}$ belongs to C_i often has nothing to do with $\mathbf{x}$ belonging to C_j. If this is the case, forcing the dependence link between the two (as in Fig. 13.2) will be counterproductive. If most classes are mutually independent, the upper layer may actually exhibit better classification performance than the lower layer, simply because the newly added attributes (the classes obtained from the upper layer) are *irrelevant*—and we know that irrelevant attributes can impair the results of induction.

Proper understanding of this issue will guide the choice of the baseline learner. Some approaches, such as induction of decision trees, or WINNOW are capable of eliminating irrelevant attributes, thus mitigating the problem.

What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- How are interdependent classes addressed by the stacking approach? Discuss both the induction phase and the classification phase.
- In what situations will stacking outperform *binary relevance* and/or a *classifier chain*?
- Under what circumstances will you prefer *binary relevance* to stacking?

13.5 A Note on Hierarchically Ordered Classes

In some domains, the known class interdependence is more complicated than in the cases we have considered so far. Our machine-learning techniques then have to be modified accordingly.

An Example Figure 13.3 shows a small part of a class hierarchy that could have been suggested by a specialist preparing the data for machine learning. Each node in the graph represents one class.

The domain deals with the classification of text documents. The hierarchy is interpreted in a way reminiscent of decision trees. To begin with, some documents may belong to the class `machine learning`. A solid line emanating from the corresponding node represents "yes," and the dashed line represents "no." Among those documents that do belong to `machine-learning`, some deal with the topic `decision tree`, others with *k*-`NN classifiers`, and so on (for simplicity, most subclasses are omitted here).

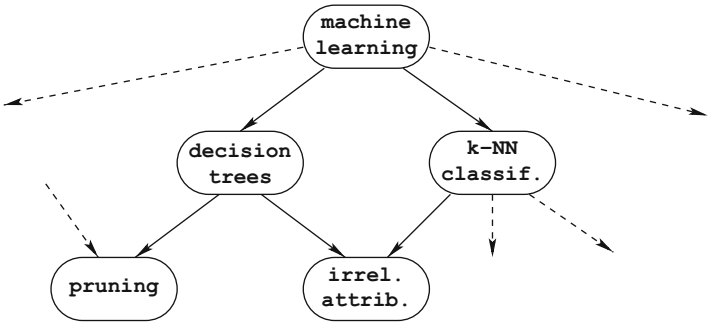

Fig. 13.3 Sometimes, the classes are hierarchically organized in a way known to the engineer in advance

In the picture, the relations are represented by arrows that point from parent nodes to child nodes. A node can have more than one parent, but a well-defined class hierarchy must avoid loops. The data structure defining class relations of this kind is known as a *directed acyclic graph*. In some applications, each node (except for the root node) has one and only one parent. This more constrained structure is known as a *generalization tree*.

Induction in Domains of This Kind Induction of hierarchically ordered classes is organized in a way similar to *binary relevance*. For each node, the corresponding training set is constructed, and from this training set, the baseline learner induces a classifier. By doing so, the most common approach proceeds in a top-down manner where the output of the parent class instructs the choice of the examples for the induction of a child class.

Here is a way to carry this out in a domain where the classes are organized by a generalization tree. First, the entire original training set is used for the induction of the class located at the root of the tree. Next, the training set is divided into two parts, one containing training examples that belong to the root class, the other containing training examples that do *not* belong to this class. The lower-level classes are then induced only from the relevant training sets.

A Concrete Example In the problem from Fig. 13.3, the first step is to induce a classifier for the class `machine learning`. Suppose that the original training set consists of the seven examples shown in Table 13.2. The labels of those examples are then used to decide which examples to include in the training sets for the induction of the child classes. For instance, note that only positive examples of `machine learning` are included in the training sets for `decision trees` and `k-NN classifiers`. Conversely, only negative examples of `machine learning` are included in the training set for the induction of `programming`.

Two Major Difficulties to Be Aware Of The induction process is not as simple as it looks. The first problem complicating the task is, again, the phenomenon of *error propagation*. Suppose an example represents a text document from the field

Table 13.2 Illustration of a domain with hierarchically ordered classes

	Machine learning
ex_1	**pos**
ex_2	**pos**
ex_3	**pos**
ex_4	**pos**
ex_5	**neg**
ex_6	**neg**
ex_7	**neg**

Decision trees	
ex_1	1
ex_2	1
ex_3	0
ex_4	0

k-NN	
ex_1	0
ex_2	0
ex_3	1
ex_4	1

Programming	
ex_5	1
ex_6	0
ex_7	0

In some lower-level classes, the training sets contain only those training examples for which the parent classifier returned **pos**; in others, only those for which the parent classifier returned **neg**

`circuit analysis`. If this example is mistakenly classified as belonging to `machine-learning`, the classifier, misled by this information, will pass it on to the next classifiers, such as `decision trees`, thus potentially propagating the error down to lower levels.[2]

Another complication is that the training sets associated with the individual nodes in the hierarchy are almost always heavily *imbalanced*. Again, appropriate measures have to be taken—usually undersampling or oversampling.

Where Does the Class Hierarchy Come From? In some rare applications, the complete class hierarchy is available right from the start, having been created manually by the customer who has the requisite background knowledge about the concrete domain. This is the case of some well-known applications from the field of text categorization.

Caution is needed, though. Customers are not infallible, and the hierarchies they develop often miss important details. They may suffer from subjectivity—with consequences similar to those explained when we discussed classifier chains. In some domains, only parts of the hierarchy are known. In this event, the engineer has to find a way of incorporating this partial knowledge in the *binary relevance* framework discussed earlier.

[2]The reader has noticed that the issue is similar to the one we have encountered in the section dealing with classifier chains.

What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- Give an example of a domain where the individual classes are hierarchically ordered.
- Explain the training-set dividing principle for the induction of hierarchically ordered classifiers.
- What are the most commonly encountered difficulties in the induction of hierarchically ordered classes?

13.6 Aggregating the Classes

The situation is simpler if there are only a few classes. For instance, the number of all class-label combinations in a domain with three classes cannot exceed seven, assuming that each example is labeled with at least one class.

Creating New Training Sets In a domain of this kind, a sufficiently large training set is likely to contain a sufficient number of representatives for each class combination, and this makes it reasonable to treat each such combination as a separate class. The general approach is similar to those we have already seen: from the original training set, T, new training sets, T_i, are created, and from each, a classifier is induced by the baseline learner. However, do not forget that, in *class aggregation*, each T_i represents one combination of class labels, say, C_2 AND C_4.

When, in the future, some example $\mathbf{x}$ is to be classified, it is presented to all of these classifiers in parallel.

Illustration of the Principle The principle is illustrated in Table 13.3. Here, the total number of classes in the original training set, T, is three. Theoretically, the total number of class combinations should be seven. In reality, only five of these combinations are encountered in T because no example is labeled with C_3 alone, and no example is labeled with all three classes simultaneously. We therefore create five tables, each defining the training set for one class combination. Note that this approach deals only with those class combinations that have been found in the original training set. For instance, no future example will be labeled as belonging to C_3 alone. This may be seen as a limitation, and the engineer will have to find a way to address this issue in a concrete application.

Classification The programmer must not forget to specify what exactly is to be done in a situation where more than one of these "aggregated" classifiers returns 1. In some machine-learning paradigms, say, a Bayesian classifier, this is easy because the classifiers are capable of quantifying their confidence in the class

Table 13.3 In a domain with a manageable number of class-label combinations, it is often possible to treat each combination as a separate class

	Classes
$\mathtt{ex}_1$	C_1, C_2
$\mathtt{ex}_2$	C_2
$\mathtt{ex}_3$	C_1, C_3
$\mathtt{ex}_4$	C_2, C_3
$\mathtt{ex}_5$	C_1

C_1	
$\mathtt{ex}_1$	0
$\mathtt{ex}_2$	0
$\mathtt{ex}_3$	0
$\mathtt{ex}_4$	0
$\mathtt{ex}_5$	1

C_2	
$\mathtt{ex}_1$	0
$\mathtt{ex}_2$	1
$\mathtt{ex}_3$	0
$\mathtt{ex}_4$	0
$\mathtt{ex}_5$	0

C_1 AND C_2	
$\mathtt{ex}_1$	1
$\mathtt{ex}_2$	0
$\mathtt{ex}_3$	0
$\mathtt{ex}_4$	0
$\mathtt{ex}_5$	0

C_1 AND C_3	
$\mathtt{ex}_1$	0
$\mathtt{ex}_2$	0
$\mathtt{ex}_3$	1
$\mathtt{ex}_4$	0
$\mathtt{ex}_5$	0

C_2 AND C_3	
$\mathtt{ex}_1$	0
$\mathtt{ex}_2$	0
$\mathtt{ex}_3$	0
$\mathtt{ex}_4$	1
$\mathtt{ex}_5$	0

they recommend. If two or more classifiers return 1, the master classifier simply chooses the one with the highest confidence.

The choice is more complicated in the case of classifiers that only return 1 or 0 without offering any information about their confidence in the given decision. In principle, one may consider merging the sets of classes. For example, suppose that, for some example **x**, two classifiers return 1, and that one of the classifiers is associated with classes C_1, C_3, and C_4, and the other is associated with classes C_3 and C_5. In this event, **x** will be labeled with C_1, C_3, C_4, and C_5.

Note, however, that this may easily result in **x** being labeled with "too many" classes. The reader already knows that this may give rise to many false positives, and thus lead to low *precision*.

Alternative Ways of Aggregation In the approach illustrated in Table 13.3, the leftmost table (the one headed by C_1) contains only one positive label because there is only one training example in T labeled solely with this class. If we want to avoid having to deal with training sets that are so extremely imbalanced, we need a "trick" that would improve the class representations in T_i's.

Here is one possibility. In T_i, we will label with 1 each example whose set of class labels in the original T contains C_1. By doing so, we must not forget that $\mathtt{ex}_1$ will thus be labeled as positive also in the table headed with (C_1 AND C_2).

Similarly, we will label with 1 all subsets of the set of classes found in a given training set. For instance, if an example is labeled with C_1, C_3, and C_4, we will label it with 1 in all training sets that represent nonempty subsets of $\{C_1, C_3, C_4\}$. This, of course, improves only training sets for relatively "small" combinations (combining, say, only one or two classes). For larger combinations, the problem persists.

A solution of the last resort will aggregate the classes only if the given combination is found in a sufficient percentage of the training examples. If the combination is rare, the corresponding T_i is not created. Although this means that the induced classifiers will not recognize a certain combination, this may not be such big loss if the combination is rare.

Some Criticism Class aggregation is not a good idea in domains where the number of class combinations is high, and the training set size is limited. If these two conditions are not satisfied, some of the newly created sets, T_i, are likely to contain no more than just a few positive examples, and as such will be ill-suited for machine learning: the training sets will be so *imbalanced* that all attempts to improve the situation by minority-class oversampling or majority-class undersampling are bound to fail—for instance, this will happen when a class combination is represented by just a single example.

As a rule of thumb, in domains with a great number of different class labels, where many combinations occur only rarely and some do not occur at all, the engineer will prefer plain *binary relevance* or some of its variations (chaining or stacking). Class aggregation is then to be avoided.

What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- Describe the principle of *class aggregation*. Explain separately the induction process and the way the induced classifiers are used to classify future examples.
- What possible variations on the class-aggregation theme do you know?
- What main shortcoming can render this approach impractical in many realistic applications?

13.7 Criteria for Performance Evaluation

We have mentioned earlier that performance evaluation in multi-label domains depends on averaging the results across classes. Let us introduce and briefly discuss some commonly used ways of doing so.

Macro-Averaging The simplest approach, *macro*-averaging, finds the values of the given criterion for each class separately, and then calculates their arithmetic average. Let L be the total number of classes. Here are the formulas that calculate macro-*precision*, macro-*recall*, and macro-F_1 from the values of these quantities for the individual classes:

$$Pr^M = \frac{1}{L}\sum_{i=1}^{L} pr_i$$
$$Re^M = \frac{1}{L}\sum_{i=1}^{L} re_i \tag{13.1}$$
$$F_1^M = \frac{1}{L}\sum_{i=1}^{L} F_{1i}$$

Macro-averaging is suitable in domains where each class has approximately the same number of representatives. In some applications, this requirement is not satisfied, but the engineer may still prefer macro-averaging if he or she considers each class to be equally important, regardless of its proportional representation in the training set.

Micro-Averaging In the other approach, *micro*-averaging, each class is weighed according to its frequency in the given set of examples. In other words, the performance is averaged over all examples. Let L be the total number of classes. Here are the formulas for micro-*precision*, micro-*recall*, and micro-F_1:

$$Pr^\mu = \frac{\sum_{i=1}^{L} N_{TP_i}}{\sum_{i=1}^{L}(N_{TP_i} + N_{FP_i})}$$
$$Re^\mu = \frac{\sum_{i=1}^{L} N_{TP_i}}{\sum_{i=1}^{L}(N_{TP_i} + N_{FN_i})} \tag{13.2}$$
$$F_1^\mu = \frac{2 \times Pr^\mu \times Re^\mu}{Pr^\mu + Re^\mu}$$

Note that F_1^μ is calculated from micro-*precision* and micro-*recall* and not from the observed classifications of the individual examples.

Micro-averaging is preferred in applications where the individual classes cannot be treated equally. For instance, the engineer may reason that good performance on dominant classes is not really compromised by poor performance on classes that are too rare to be of any importance.

A Numeric Example Let us illustrate these formulas using Table 13.4. Here, we can see five examples. For each of them, the middle column lists the correct class labels, and the rightmost column gives the labels returned by the classifier. The reader can see minor discrepancies in the sense that the classifier has missed some classes (causing false negatives). For instance, this is the case of class C_3 being missed in example $\mathtt{ex}_3$. At the same time, the classifier labels some examples with incorrect class labels, which constitutes false positives. For instance, this is the case of example $\mathtt{ex}_1$ being labeled with class C_3.

Table 13.4 Illustration of performance evaluation in multi-label domains

The following table gives, for five testing examples, the known class labels versus the class labels returned by the classifier.

	True classes	Classifier's classes
$\texttt{ex}_1$	C_1, C_2	$C_1, C_2, C_3,$
$\texttt{ex}_2$	C_2	$C_2, C_4,$
$\texttt{ex}_3$	C_1, C_3, C_5	$C_1, C_5,$
$\texttt{ex}_4$	C_2, C_3	$C_2, C_3,$
$\texttt{ex}_5$	C_2, C_4	$C_2, C_5,$

Separately for each class, here are the values of true positives, false positives, and false negatives. Next to them are the corresponding values for precision and recall, again separately for each class.

$$
\begin{array}{lllll}
N_{TP_1} = 2 & N_{FP_1} = 0 & N_{FN_1} = 0 & Pr_1 = \frac{2}{2+0} = 1 & Re_1 = \frac{2}{2+0} = 1 \\
N_{TP_2} = 4 & N_{FP_2} = 0 & N_{FN_2} = 0 & Pr_2 = \frac{4}{4+0} = 1 & Re_2 = \frac{4}{4+0} = 1 \\
N_{TP_3} = 1 & N_{FP_3} = 1 & N_{FN_3} = 1 & Pr_3 = \frac{1}{1+1} = 0.5 & Re_3 = \frac{1}{1+1} = 0.5 \\
N_{TP_4} = 0 & N_{FP_4} = 1 & N_{FN_4} = 1 & Pr_4 = \frac{0}{0+1} = 0 & Re_4 = \frac{0}{0+1} = 0 \\
N_{TP_5} = 1 & N_{FP_5} = 1 & N_{FN_5} = 0 & Pr_5 = \frac{1}{1+1} = 0.5 & Re_5 = \frac{1}{1+0} = 1
\end{array}
$$

This is how the macro-averages are calculated:

$$Pr^M = \frac{1+1+0.5+0+0.5}{5} = 0.6$$

$$Re^M = \frac{1+1+0.5+0+1}{5} = 0.7$$

Here is how the micro-averages are calculated:

$$Pr^\mu = \frac{2+4+1+0+1}{(2+0)+(4+0)+(1+1)+(0+1)+(1+1)} = 0.73$$

$$Re^\mu = \frac{2+4+1+0+1}{(2+0)+(4+0)+(1+1)+(0+1)+(1+0)} = 0.8$$

These discrepancies are then reflected in the numbers of true positives, false positives, and false negatives. These, in turn, make it possible to calculate for each class its *precision* and *recall*. After this, the table shows the calculations of the macro- and micro-averages of these two criteria.

Averaging the Performance over Examples So far, the true and false positive and negative examples were counted across individual classes. However, in domains where an average example belongs to a great many classes, it can make sense to average over the individual examples.

The procedure is in principle the same as before. When comparing the true class labels with those returned for each example by the classifier, we obtain the numbers of true positives, false positives, and false negatives. From these, we easily obtain the macro-averages and micro-averages. The only thing we must keep in mind is that the average is not taken over the classes, but over examples—thus in macro-averages, we divide the sum by the number of examples, not by the number of classes.

What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Give the formulas for *macro*-averaging of *precision*, *recall*, and F_1.
- Give the formulas for *micro*-averaging of *precision*, *recall*, and F_1. Discuss the difference between *macro*-averaging and *micro*-averaging.
- What is meant by "averaging the performance over examples"?

13.8 Summary and Historical Remarks

- In some domains (such as text categorization), each example can be labeled with more than one class at the same time. These are so-called *multi-label* domains.
- In domains of this kind, classical machine-learning paradigms can sometimes be used. Unless special precautions have been taken, however, the results are rarely encouraging. For some paradigms, multi-label versions exists, but these are too advanced for an introductory text, especially in view of the fact that good results can be achieved with simpler means.
- The most common approach to *multi-label domains* induces a binary classifier for each class separately, and then submits the example to all these classifiers in parallel. This is called the *binary relevance* technique.
- What the basic version of *binary relevance* seems to neglect is the fact that the individual classes may not be independent of each other. The fact that an example has been identified as a representative of class C_A may strengthen or weaken its chances of belonging also to class C_B.
- The simplest mechanism for dealing with class interdependence in multi-label domains is the *classifier chain*. Here, the output of one binary classifier is used as an additional attribute describing the example to be presented to the next classifier in line.

- One weakness of classifier chains is that the user is expected to specify the sequence of classes (perhaps according to class subsumption). If the sequence is poorly designed, the results are disappointing.
- Another shortcoming is known as *error propagation*: an incorrect label given to an example by one classifier is passed on to the next classifier in the chain, potentially misleading it.
- A safer approach relies on the two-layered *stacking* principle. The upper-layer classifiers are induced from examples described by the original attribute vectors, and the lower-layer classifiers are induced from examples described by attribute vectors to which the class labels obtained in the upper layer have been added. When classifying an example, the outcomes of the lower-layer classifiers are used.
- Sometimes, it is possible to take advantage of known hierarchical order among the classes. Here, too, induction is carried out based on specially designed training sets. Again, the user has to be aware of the dangers of error propagation.
- Yet another possibility is to resort to *class aggregation* where each combination of classes is treated as a separate higher-level class. A special auxiliary training set is created for each of these higher-level classes.
- The engineer has to pay attention to ways of measuring the quality of the induced classifiers. Observing that each class may experience different classification performance, we need mechanisms for averaging over the classes (or examples). Two of them are currently popular: micro-averaging and macro-averaging.

Historical Remarks The problem of multi-label classification is relatively new. The first time it was encountered was in the field of text categorization—see McCallum [57]. The simplest approach, the *binary relevance* principle, was employed by Boutell et al. [7]. A successful application of classifier chains was reported by Read et al. [79], whereas Goldpole and Sarawagi [32] are credited with having developed the *stacking* approach. Apart from the approaches related to *binary relevance*, some authors have studied ways of modifying classical single-label paradigms. The ideas on nearest-neighbor classifiers in multi-label domains are borrowed from Zhang and Zhou [101] (whose technique, however, is much more sophisticated than the one described in this chapter). Induction of hierarchically ordered classes was first addressed by Koller and Sahami [47]. Multi-label decision trees were developed by Clare and King [15].

13.9 Solidify Your Knowledge

The exercises are to solidify the acquired knowledge. The suggested thought experiments will help the reader see this chapter's ideas in a different light and provoke independent thinking. Computer assignments will force the readers to pay attention to seemingly insignificant details they might otherwise overlook.

Table 13.5 An example of a multi-label domain

	True classes	Classifier's classes
$\texttt{ex}_1$	C_1	C_1, C_2
$\texttt{ex}_2$	C_1, C_2	C_1, C_2
$\texttt{ex}_3$	C_1, C_3	C_1
$\texttt{ex}_4$	C_2, C_3	C_2, C_3
$\texttt{ex}_5$	C_2	C_2
$\texttt{ex}_6$	C_1	C_1, C_2

Exercises

1. Consider the multi-label training set shown in the left part of Table 13.5. Show how the auxiliary training sets will be created when the principle of *binary relevance* is to be used.
2. For the same training set, create the auxiliary training sets for the approach known as *class aggregation*. How many such sets will we need?
3. Draw the schema showing how the problem from Table 13.5 would be addressed by *stacking*. Suppose the examples in the original training set are described by ten attributes. How many attributes will the lower-level classifiers have to use?
4. Suggest the classifier-chain schema for a domain with the following four classes: `decision trees`, `machine learning`, `classification`, `pruning`.
5. Returning to the set of examples from Table 13.5, suppose that a classifier has labeled them as indicated in the rightmost column. Calculate the macro- and micro-averages of *precision* and *recall*.

Give It Some Thought

1. Suggest a multi-label domain where the principle of *classifier chain* can be a reasonable strategy to follow. What would be the main requirement for such data?
2. Consider a domain where the majority of training examples are labeled each with only a single class, and only a small subset of the examples (say, 5%) are labeled with more than one class. Suggest a machine learning approach to induce reliable classifiers from such data.
3. Suppose that you have a reason to assume that a few classes are marked by strong interdependence while most of the remaining classes are mutually independent. You are thinking of using the *stacking* approach. What is the main problem that might compromise the performance of the induced classifiers? Can you suggest a mechanism that overcomes this pitfall?

4. Suppose you have been asked to develop machine-learning software for induction from multi-label examples. This chapter has described at least four approaches that you can choose from. Write down the main thoughts that would guide your decision.
5. Suggest a mechanism that would mitigate the problem of *error propagation* during multi-label induction with hierarchically ordered classes. Hint: after a testing run, consider "enriching" the training sets by "problematic" examples.

Computer Assignments

1. Write a program that accepts as input a training set of multi-label examples, and returns as output the set of auxiliary training sets needed for the *binary relevance* approach.
2. Write a program that converts the training set from the previous question into auxiliary training sets, following the principle of class aggregation.
3. Search the web for machine-learning benchmark domains that contain multi-label examples. Convert them using the data-processing program from the previous question, and then induce the classifiers by the *binary relevance* approach.
4. Write a program that first induces the classifiers using *binary relevance* as in the previous question. In the next step, the program redescribes the training examples by adding to their attribute vectors the class labels as required by the lower layer in the *classifier stacking* technique.
5. What data structures would you use for the input and output data when implementing the *classifier stacking* technique?
6. Write a program that takes as input the values of $N_{TP}, N_{TN}, N_{FP}, N_{FN}$ for each class, and returns micro- and macro-averaged *precision*, *recall*, and F_1.