2. On classification and related tasks

In this part of the course we take a concise ”bird’s-eye view” of different central tasks and concepts involved in machine learning and classification particularly. A task here refers to whatever it is that machine learning is intended to improve performance. When this is a classification task, we need to learn an appropriate classifier from training data. There are many types of classifiers, for example, Bayesian classifiers, distance-based classifiers, say, nearest neighbor searching, to name a few. Generally, we call them *models*.

In the following we define or list some useful concepts. To classify, *labels l* of data cases (instances or examples) are known so that computational models can be built on the basis of a training set and validated with a test set. When considering other tasks than classification, labels are typically unknown. However, labels can then be generated, for instance, by clustering.

Data cases are represented with variables (attributes or features) in addition to the label. Noise may distort variable values. It is formed by erroneous measurements or random influence like ”measuring inaccuracy”. In addition, missing values may appear. Notwithstanding these data quality lowering matters classification is still possible provided that the data set has not heavily deteriorated.

In principle, there could also be noisy class labels, i.e., somehow corrupted, but we do not treat this kind of special situations.

One consequence of noisy data is that it is not generally advisable to attempt to match the training data exactly, as this may lead to overfitting the noise.

2.1 Binary classification

Classification is the commonest task in machine learning. A *classifier* is a mapping from the space of all possible data cases *K* represented by training set *T* onto a finite and usually small set of class labels $\{c_1, c_2,\ldots, c_C\}$ (*C* classes altogether).

In the simplest case, there are only two classes which are usually referred to as positive and negative, + and -, or +1 and -1. Two-class classification is called *binary classification*.

Spam email filtering is a good example of binary classification, in which spam is conventionally seen as the positive class and non-spam as the negative class. Positive here has not been ”good”, but rather something important which is being attempted to be caught.

The tree in Fig. 2.1 (a) can be turned into a classifier by labeling each leaf with a class. The simplest way to do this is by assigning the *majority class* in each leaf, resulting in the decision tree in Fig. 2.1 (b). The classifier works as follows: if an email contains the word ’inheritance’, it is classified as Ĉ=spam (rightmost leaf). Otherwise, the occurrence of the word ’lottery’ decides whether it gets labeled spam or non-spam. From numbers in Fig. 2.1 we can get an idea how well the classifier does. The leftmost leaf correctly predicts 40 non-spam emails, but also misclassifies 20 spam emails that contain neither ’inheritance’ nor ’lottery’. The middle leaf correctly classifies 10 spam emails and incorrectly 5 non-spam emails as spam. The ’inheritance’ test correctly picks out 20 spam emails, but also 5 non-spam. Altogether, 30 out of 50 spam emails are classified correctly and 40 out of 50 non-spam emails.

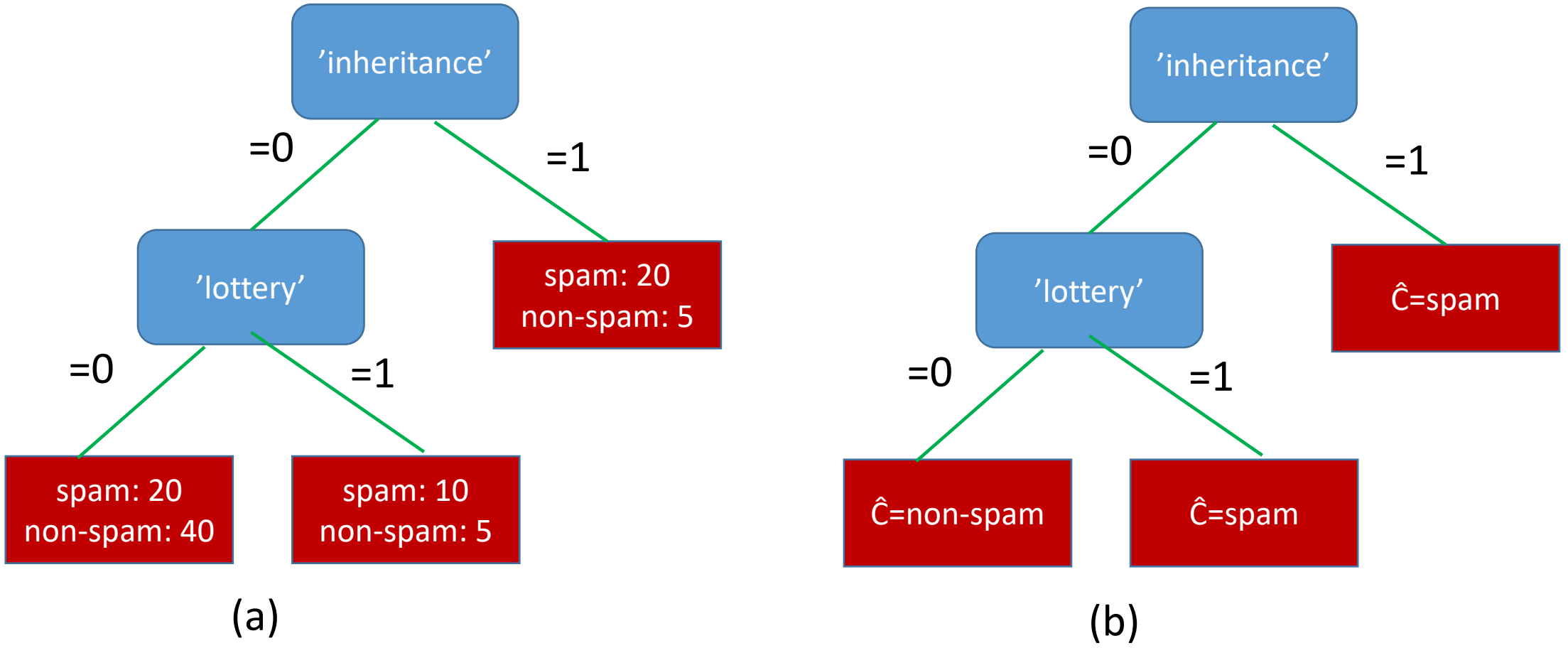

Fig. 2.1 (a) A tree with training set class distribution on the leaves. (b) A decision tree obtained using the majority class decision rule. Decision 0 equals 'no' and 1 'yes'.

Assessing classification performance

The performance of classifiers can be summarised by means of a table known as *contingency table* or *confusion matrix* (Table 2.1). In this table each row refers to actual classes as recorded in the test set, and each column to classes as predicted by the classifier. For example, the first row states that the test set consists of 50 positives, 30 of which were correctly predicted and 20 incorrectly. The last column and last row show the *marginals*, column and row sums. Marginals are important, because they allow us to assess statistical significance. For instance, the contingency Table 2.1 (b) has the same marginals, but the classifier clearly makes a random choice as to which predictions are positive and which are negative – as a result the distribution of actual positives and negatives in either predicted class is the same as the overall distribution (uniform here).

Table 2.1 (a) A two-class contingency table or confusion matrix depicting the performance of the decision tree in Fig. 2.1. Numbers on the descending diagonal indicate correct predictions, while the ascending diagonal concerns prediction errors. (b) A contingency table with the same marginals but independent rows and columns.

(a)	Predicted +	Predicted -	
Actual +	30	20	50
Actual -	10	40	50
	40	60	100

(b)	+	-	
+	20	30	50
-	20	30	50
	40	60	100

From a contingency table we can calculate a range of peformance indicators. The simplest and one of the most important is *accuracy*, which is the proportion of correctly classified test cases.

While classifying tests cases, the testing or validation program has to count how many cases of the class + of interest are classified correctly in +. These are called *true positive* (*TP*). Those tests cases from the opposite class - that are classified correctly are counted and called *true negative* (*TN*). Other cases from + classified incorrectly into - are called *false nagative* (*FN*) and those from - incorrectly classified into + are called *false positive* (*FP*) according to Table 2.2.

Accuracy measures

Table 2.2 Accuracy rates	Classification of the model: positive	Classification of the model: negative
Actual positive	*TP*	*FN*
Actual negative	*FP*	*TN*

We compute classification or prediction results, where *N* is the number of the test cases in a test set:

$$\text{true positive rate } TPR = \text{sensitivity} = \text{recall} = \frac{TP}{TP + FN} 100\%$$

$$\text{true negative rate } TNR = \text{specificity} = \text{negative recall} = \frac{TN}{TN + FP} 100\%$$

$$(1)\ (\text{total})\ \text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN} 100\% = \frac{TP + TN}{N} 100\%$$

$$\left(\quad (2)\ \ \text{accuracy} = \frac{TPR + TNR}{2} \quad \right)$$

$$\text{positive predictive value} = \text{precision} = \text{confidence} = \frac{TP}{TP + FP} 100\%$$

$$\text{negative predictive value} = \frac{TN}{TN + FN} 100\%$$

Usually, accuracy is computed according to (1), and (2) is rarely employed being appropriate when the numbers of positives and negatives are quite equal.

Sometimes error counts are calculated, e.g., in classification applications that concern finding error functions of devices, fault diagnostics. Then the following three measures are useful. Note that then typically it is assumed that errors are infrequent.

$$\text{error rate} = (1 - \text{accuracy})100\% = \left(1 - \frac{TP + TN}{TP + TN + FP + FN}\right)100\%$$

$$= \left(1 - \frac{TP + TN}{N}\right)100\% = \left(\frac{FP + FN}{N}\right)100\%$$

$$\text{false positive rate} = \text{false alarm rate} = \text{false acceptance rate} =$$

$$\frac{FP}{FP + TN}100\% = (1 - \text{true negative rate})100\%$$

$$\text{false negative rate} = \text{false rejection rate} = \frac{FN}{FN + TP}100\%$$

Sometimes these measures are expressed without per cent, i.e., as values from [0,1].

If we consider precision and recall, we can see that they are to some extent inversely related, in that if the number of false positives increases (meaning that the algorithm is using a broader definition of that class), then the number of false negatives often decreases, and vice versa. They can be combined to give a single measure, F_1 measure, which can be written in terms of precision and recall, or true positives and false negatives and positives:

$$F_1 = \frac{2 \cdot precision \cdot recall}{precision + recall} = \frac{TP}{TP + (FN + FP)/2}$$

ROC curve

We can use the preceding measures to evaluate a particular classifier with different learning parameter values or different classifiers. In this case, *Receiver Operator Characteristic* curve – almost always known just as the ROC curve – is useful.

ROC curve maps percentage of true positives on the *y* axis against false positives of the *x* axis. See Fig. 2.2.

A single run of a classifier yields a single point on the ROC plot. A perfect classifier would give a point at (0,100), that is, 100% true positives and 0% false positives. A fictional ”anti-classifier” classifying everything wrong would give (100,0).

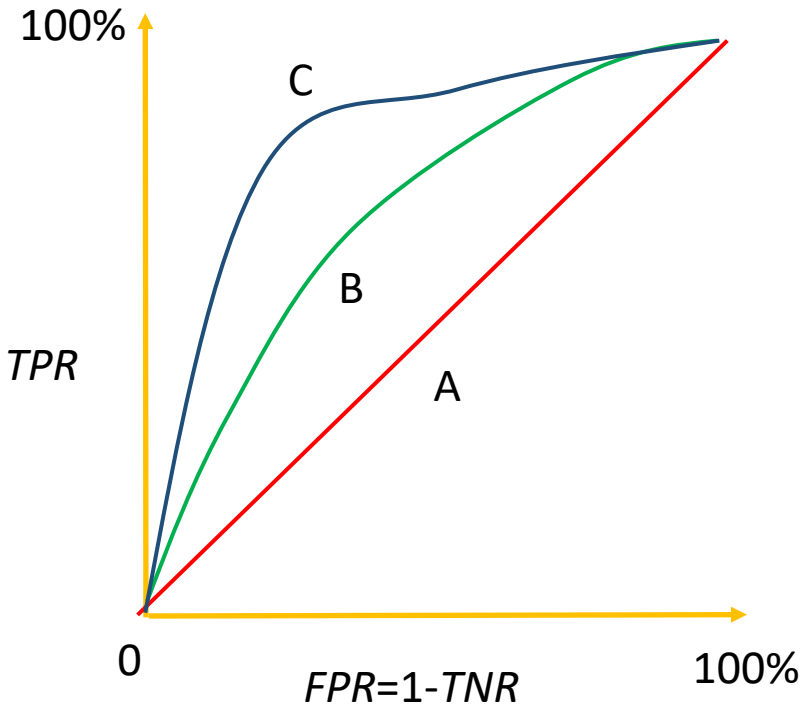

Fig. 2.2 The diagonal line A of represents exactly random, so anything above it is better than chance. The further from the line A, that is, B and C, the better.

(In Matlab ROC curve can be computed with ’perfcurve’.)

Points of ROC curve are computed by running the same classifier several times with different choices of a training set and test set, for instance, by crossvalidation.

In order to compare the same classifier with different control parameter settings or different classifiers, ROC curves are generated for them.

Additionally, AUC (Area Under Curve) can be computed to evaluate the performance. The scales are then typically [0,1] instead of per cent. For random classifications, AUC is equal to ½ and for better results it is from interval (1/2,1].

Fig. 2.3 presents an example.

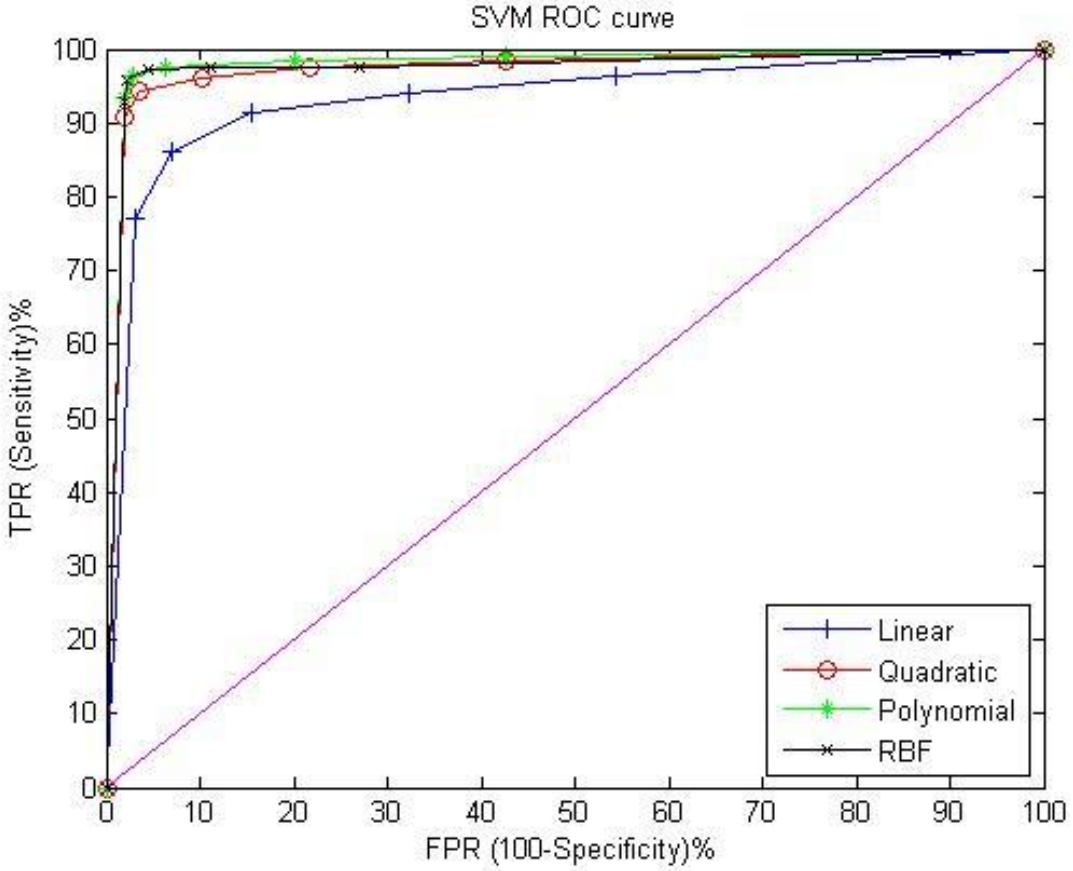

Fig. 2.3 ROC curves[1] presented with true positive rates (TPR) and false positive rates (FPR) in percent for support vector machines with respect to linear, quadratic, polynomial of degree 3 and radial basis function (RBF) kernel with σ=2.5.

[1] Y. Zhang and M. Juhola: On applying signals of saccade eye movements for biometric verification of a subject, 8th International Conference on Mass Data Analysis of Images and Signals (MDA 2013), Springer, pp. 78-92.

Unbalanced data sets

For the accuracy considered so far it assumes implicitly – in principle – that there are the same number of positive and negative cases in the (training) data set. This is known as a balanced data set. Nevertheless, this is rarely true. In practice, small differences do not affect. (If a class distribution is really skewed including a highly predominating majority class, such a situation may also be difficult to train in machine learning sense; the majority class is learnt easily, but small classes may be ”lost” even completely at their worst.)

A simple approach is to compute a balanced accuracy (2) (p. 56), but a more correct measure is the following Matthew’s Correlation Coefficient.

$$MCC = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} 100\%$$

If any sum of the parentheses in the denominator is 0, then the whole of the denominator is set to 1.

Class probability estimation

A *class probability estimator* is a scoring classifier that outputs probability vectors over classes, mapping $K \rightarrow [0,1]^C$ where $K = V_1 \times V_2 \times \ldots \times V_p$ (data, Cartesian product of the variables) and C is the number or classes. The mapping space includes a vector in which one component only equals 1 and the other equal 0. The components of the probability vector correspond to the probabilities of the classes as follows:

$$\hat{\mathbf{p}} = \left(\hat{p}_1(x), \ldots, \hat{p}_C(x)\right), \sum_{i=1}^{C} \hat{p}_i(x) = 1$$

If there are only two classes, the probability of one class is naturally 1 minus that of the other class.

In Fig. 2.4 there are probabilities derived from Fig. 2.1 (a).

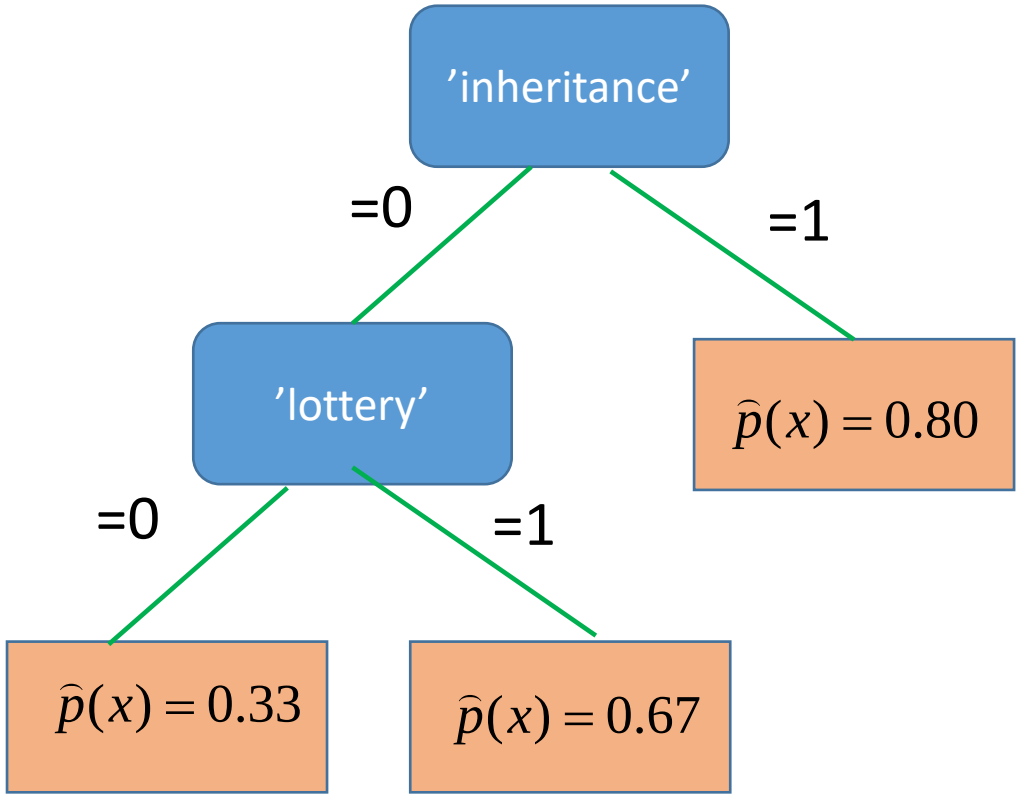

Fig. 2.4. A probability estimation tree derived from the tree in Fig. 2.1 (a).

If we threshold probabilities at 0.5, i.e., predict spam if the spam probability is 0.5 or more and predict non-spam otherwise, we get the same classifier as obtained by prediciting the majority class in each leaf of the tree in Fig. 2.1 (b).

Assessing class probability estimates

As with classifiers, we can ask how good these estimators are. A slight complication here is that we do not have access to the true probabilities. One trick that is often applied is to define a binary vector ($I[c(x)=c_1],\ldots, I[c(x)=c_C]$), which has the ith bit set to 1 if x's true class is c_i and all other bits are set to 0, and use these as the "true" probabilities. We can then define the *squared error* (*SE*) of the predicted probability vector

$$\hat{\mathbf{p}} = \left(\hat{p}_1(x),\ldots, \hat{p}_C(x)\right)$$

of C classes as follows:

$$SE(x) = \frac{1}{2}\sum_{i=1}^{C}\left(\hat{p}_i(x) - I[c(x)=c_i]\right)^2$$

The *mean squared error* (*MSE*) as the average squared error over all cases in the test set *T* is:

$$MSE(T) = \frac{1}{|T|} \sum_{x \in T} SE(x)$$

Sometimes such an error measure as root mean squared error (*RMSE*) is computed. *RMSE* comes from *MSE* by taking square root over *MSE*.

2.2 Beyond binary classification

Certain concepts are fundamentally binary. In the following, we will consider general issues related to having more than two classes in classification. The discussion will address two issues: how to evaluate multiclass performance, and how to build multiclass models out of binary models.

Handling more than two classes

Classification tasks with more than two classes are common. For instance, once a patient has been diagnosed as suffering from rheumatic disease, the physician will want to classify him or her further into one of several variants. If there are *C* classes, performance of a classifier can be assessed using a *C*-by-*C* contingency table, which is still the sum of the descending diagonal of the contingency table, divided by the number of test cases. See the following example.

Example: performance of multiclass classifiers

	Predicted			
Actual	**15**	2	3	20
	7	**15**	8	30
	2	3	**45**	50
	24	20	56	100

The accuracy of this classification is ((15 + 15 + 45)/100)·100% = 75%. We can calculate, for example, true positive rate or recall per class, which are, class by class, (15/20)·100% = 75%, (15/30)·100%=50% and (45/50)·100%=90%. We could average these numbers to obtain single true positive rate for the whole classifier or could calculate a weighted mean taking the proportion of each class into account.

Construction of multiclassification with a set of binary classifiers

Such binary classification methods as linear classifiers can be employed as a set in order to run multiclassification. In fact, any type of classifiers can be used in the following binary way. Besides, for some complicated (e.g. with considerably overlapping classes) data sets we may attain better results with these constructions than with the ”direct” use of multiclass classifiers.

The *one-versus-rest* or *one-versus-all* scheme is to train C binary classifiers, the first of which separates class C_1 from the other classes C_2 ,…, C_C, the second of which separates C_2 from C_1, C_3 ,…, C_C, and so on. When training the ith classifier we treat all cases of class C_i as positive cases and the remaining cases as negative cases. This means that C classifiers are trained and tested.

An alternative is *one-versus-one*. In this scheme, we train $C(C\text{-}1)/2$ binary classifiers, one for each pair of different classes.

A convenient way to use a set of these binary classifiers is voting. For one-versus-rest, we give 1 vote to either class C_i or its ”opponents”, rest classes, depending on how the classifier predicts the test case. If a test case really originates from C_i, it is more probable that a classifier classifies it into that than into its opponents. If it originates from the other classes, the opposite result is more probable.

The procedure is run for all classes $i=1,\ldots,C$. A *tie* can occur for the case which means that two (or more in principle) classes obtain 1. Then we can make a random decision between the two (or more) guessing the class from those obtained 1 vote, or apply another classifier type to determine the result between them.

A rejection is also possible for some classifiers such as decision trees that occasionally cannot make a decision in all of their leaves. In such a situation voting between one class versus the rest cannot be made (if not using other classifier type).

In principle, it is possible that an exceptional case is classified into no class C_i, but always to its opponents, other classes. This is improbable and exceptional, but possible for complicated data. See hypothetical Fig. 2.5. This is an infeasible situation for the one-versus-rest construction, but could be attempted to be classified with some other classification method.

The procedure is run for all cases of the test set and correct and incorrect results are computed and suitable previous accuracy measures applied to them.

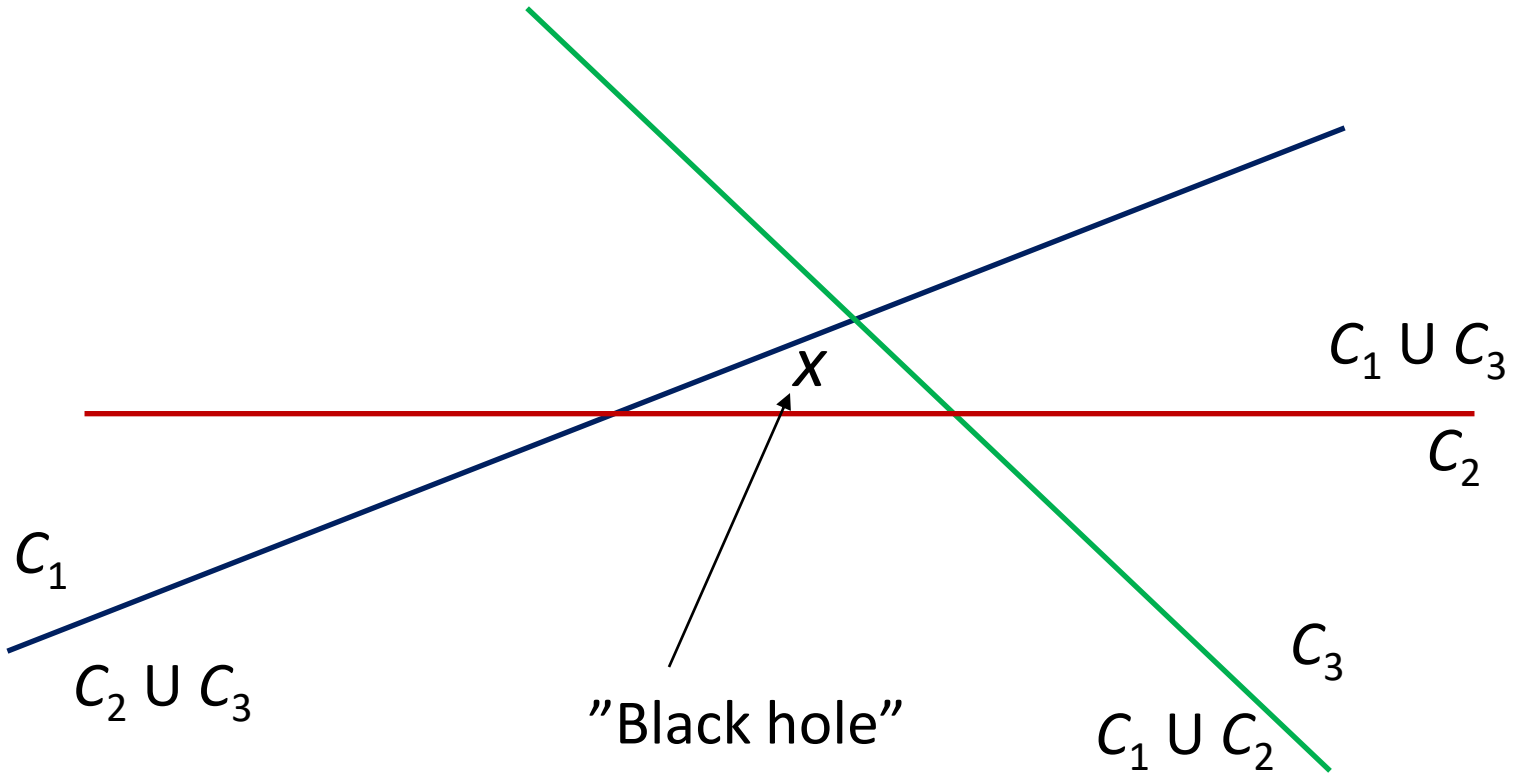

Fig. 2.5 If the first classification result of test case *x* is in area C_2 U C_3, the second result in C_1 U C_3, and the third in C_1 U C_2, *x* obtains no final decision since it is in the unknown area without any classification solution.

One-versus-one is another construction. Then, while running a classifier, 1 vote is given one of two classes. If the case is really from one of these two classes, the correct class is probably classified obtaining 1 vote. If the case is not from either, the result is rather random meaning that in probability of approximately 0.5 one or the other is decided. This way, while running more one-versus-one pairs, the correct class will probably collect more votes (maximum *C*-1) than other, ”random hits”. Finally, by the technique of majority voting the class is predicted. However, a tie is again possible if two (or more) classes obtain equally many votes.

All test cases are tested and, ultimately, accuracy measures computed.

One-versus-one requires more computation than one-versus-rest if looked at the number of classes, $O(C^2)$ and $O(C)$, bacause there are more pairs composed of two classes only. On the other hand, this is not actually so simple. Inasmuch as in one-versus-one a training set is considerably smaller including only the training cases of two classes, building a classifier can be executed much faster.

For real data sets, classification accuracy is hardly ever equal to 100%. This would be possible only if classes were entirely separable. In the ”incomplete nature”, such is very infrequent. If nothing else, some noise (random influence such as measurement error) may occur in a data set.

How to test a classifier?

At first, we divide our entire data set **randomly** to a training (learning) set and test set. A computational model is generated by using some machine training method on the basis of the current training set. Then the model is tested by using the current test set. The simplest division is the **hold-out** method: 50% of cases into both training and test set. If the number of cases is limited such as in Vertigo data[4] (815 patient cases in six disease classes), **crossvalidation** (CV) is often used: data are first divided into usually *k*=10 randomly chosen subsets of 10% and then one by one of them is a test set, but the rest with 90% is the corresponding training set (10 training set - test set pairs).

[4]M. Siermala et al., Evaluation and classification of otoneurological data with new data analysis methods based on machine learning, Information Sciences, 177, 1963-1976, 2007

A single test run of those 10 pairs does not yet give representative information about classification accuracy. Many test runs are made for all the data set and finally average classification accuracy is calculated as in Fig. 2.6. Repetitions can be made with new randomly chosen 10 subsets. Nevertheless, if the classification process contains some random choices, e.g., initial values, repetitions can also made for the same 10 subsets.

Leave-one-out is a technique in which a test set contains one case only and all other *n*-1 cases are included in its training set. This way *n* tests are run with *n* only slightly different models. This is applied when a data set is relatively small and as many training cases as possible are wanted to use for training.

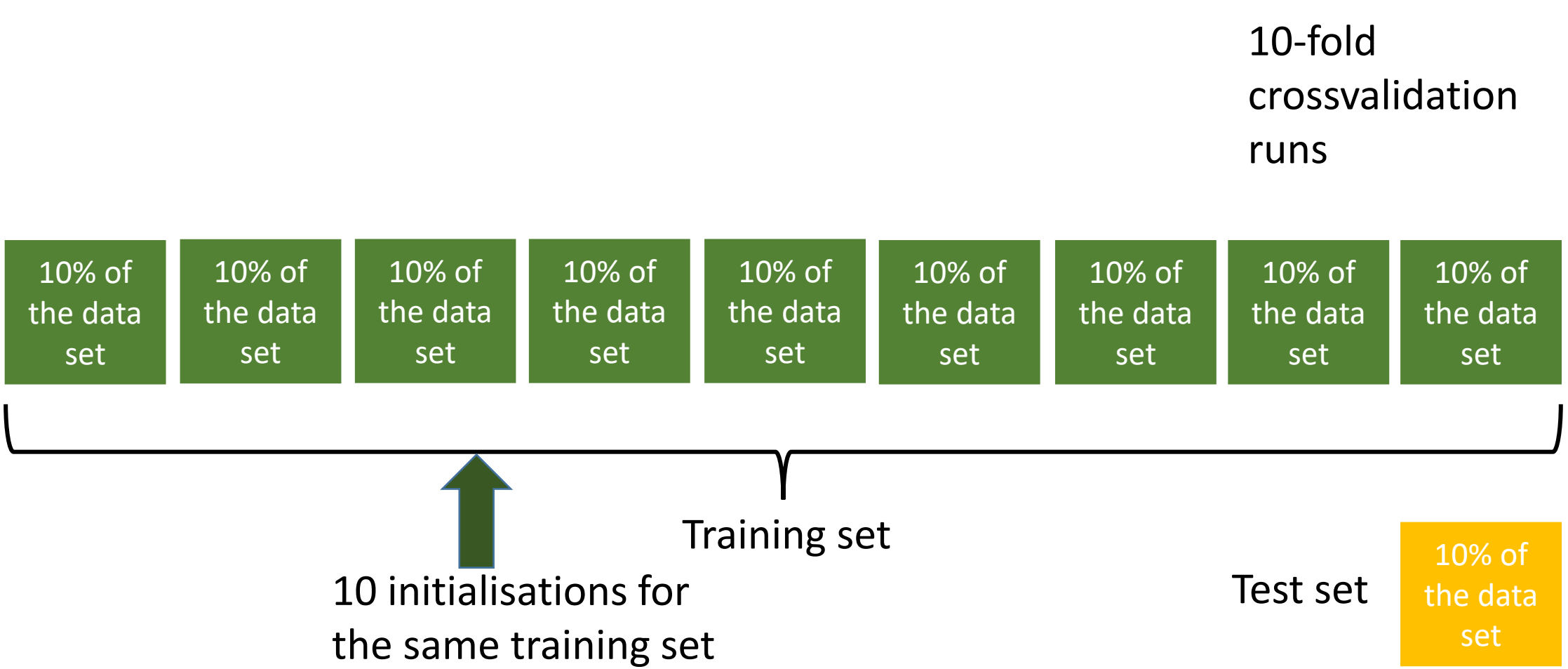

Fig. 2.6 10-fold crossvalidation gives 10 models. This random division can be repeated, e.g., 10 times producing 100 test runs altogether.