9

Kernels and Distances for Structured Data

Many contemporary machine learning approaches employ kernels or distance metrics, and in recent years there has been a significant interest in developing kernels and distance metrics for structured and relational data. This chapter provides an introduction to these developments, starting with a brief review of basic kernel- and distance-based machine learning algorithms, and then providing an introduction to kernels and distance measures for structured data, such as vectors, sets, strings, trees, atoms and graphs. While doing so, we focus on the principles used to develop kernels and distance measures rather than on the many variants and variations that exist today. At the same time, an attempt is made to draw parallels between the way kernels and distance measures for structured data are designed. Convolution and decomposition can be used to upgrade kernels for simpler data structures to more complex ones; cf. [Haussler, 1999]. For distances, decomposition is important, as is the relationship between the generality relation and the size of the data; cf. [De Raedt and Ramon, 2008].

9.1 A Simple Kernel and Distance

Distance- and kernel-based learning is typically addressed in the traditional function approximation settings of classification and regression introduced in Sect. 3.3. Both distances and kernels provide information about the similarity amongst pairs of instances. Indeed, a kernel function can directly be interpreted as measuring some kind of similarity, and the distance amongst two objects is inversely related to their similarity.

To introduce distances and kernels, we will in this section focus on the familiar Euclidean space, that is $\mathcal{L}_e = \mathbb{R}^d$. In this space, the instances correspond to vectors, for which we use the notation $\mathbf{x}$ or $(x_1, \cdots, x_d)$, with components x_i.

In the next section, we will then discuss the use of kernels and distances in machine learning using standard distance- and kernel-based approaches, such

as the k-nearest neighbor and support vector machine algorithms. Afterwards, we shall investigate how distances and kernels for structured data can be constructed, in particular for strings, sets and graph-structured data. While doing so, we shall focus on the underlying principles rather than on the many specific distance measures and kernels that exist today.

A natural distance measure for $\mathbb{R}^d$ is the *Euclidean distance*:

$$d_e(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_i (x_i - y_i)^2} \tag{9.1}$$

The simplest kernel for $\mathbb{R}^n$ is the *inner product*:

$$< \mathbf{x}, \mathbf{y} >= \sum_i x_i y_i \tag{9.2}$$

The *norm* $\|\mathbf{x}\|$ of a vector $\mathbf{x}$ denotes its size. It can be computed from the inner product as follows:

$$\|\mathbf{x}\| = \sqrt{< \mathbf{x}, \mathbf{x} >} \tag{9.3}$$

Furthermore, the inner product $< \mathbf{x}, \mathbf{y} >$ of two vectors in $\mathbb{R}^d$ can be written as

$$< \mathbf{x}, \mathbf{y} >= \|\mathbf{x}\| \ \|\mathbf{y}\| cos(\alpha) \tag{9.4}$$

where α is the angle between the two vectors $\mathbf{x}$ and $\mathbf{y}$, which allows us to interpret the inner product in geometric terms. The equation states that when the two vectors are normalized (that is, have length 1), the inner product computes the cosine of the angle between the two vectors. It then follows that when $\alpha = 0$, $< \mathbf{x}, \mathbf{y} > = 1$, and when the two vectors are orthogonal, the inner product yields the value 0. This indicates that the inner product measures a kind of similarity.

Even though we will formally introduce kernels and distance measures later an interesting relationship amongst them can already be stated. Any (positive definite) kernel K induces a distance metric d_K as follows:

$$d_K(\mathbf{x}, \mathbf{y}) = \sqrt{K(\mathbf{x}, \mathbf{x}) - 2K(\mathbf{x}, \mathbf{y}) + K(\mathbf{y}, \mathbf{y})} \tag{9.5}$$

It is instructive to verify this statement using the kernel $K(\mathbf{x}, \mathbf{y}) =< \mathbf{x}, \mathbf{y} >$.

Exercise 9.1. Show that the distance d_e can be obtained from Eq. 9.5 using the inner product as a kernel, that is:

$$d_e(\mathbf{x}, \mathbf{y}) = \sqrt{< \mathbf{x}, \mathbf{x} > - 2 < \mathbf{x}, \mathbf{y} > + < \mathbf{y}, \mathbf{y} >}. \tag{9.6}$$

A key difference between a kernel and a distance measure is that kernels measure the similarity between instances whereas distances measure the dissimilarity.

Example 9.2. Assume that the instances are vectors over $\{-1, +1\}^d$ and that we are using, as before, the inner product kernel. The maximum value of the inner product over all possible pairs of instances will be d, and this value will be obtained only when the two instances are identical. The minimum value will be $-d$, which will be obtained when the two vectors contain opposite values at all positions. So, the maximum is reached when the vectors are identical and the minimum is reached when they are opposite. It is easy to see that using a distance measure this is just the other way around. The minimum of 0 will always be reached when the objects are identical.

9.2 Kernel Methods

In this section, we briefly review the key concepts underlying kernel methods in machine learning. More specifically, we first introduce the max margin approach, continue with support vector machines, and then conclude with the definitions of some simple kernels. This section closely follows the introductory chapter of the book by Schölkopf and Smola [2002]. The next section then provides an introduction to distance-based methods in machine learning.

9.2.1 The Max Margin Approach

The basic max margin approach tackles the binary classification task, in which the instances are vectors and the classes are $\{+1, -1\}$. The goal is then to find a hypothesis h in the form of a hyperplane that separates the two classes of examples. Hyperplanes can conveniently be defined using an inner product and the equation

$$< \mathbf{w}, \mathbf{x} > +b = 0 \tag{9.7}$$

where $\mathbf{w}$ is the weight vector and b is a constant. The hyperplane then consists of all vectors $\mathbf{x}$ that are a solution to this equation. Such a hyperplane h can then be used to classify an example e as positive or negative by computing

$$h(e) = sgn(< \mathbf{w}, \mathbf{e} > +b) \text{ where} \tag{9.8}$$

$$sgn(val) = \begin{cases} +1 & \text{if } val > 0 \\ -1 & \text{if } val \leqslant 0 \end{cases} \tag{9.9}$$

The idea is not just to compute any hyperplane h that separates the two classes, but the optimal one, which maximizes the margin. The *margin* of a hyperplane $h = (\mathbf{w}, b)$ w.r.t. a set of examples E is given by

$$margin(h, E) = \min\{\|\mathbf{x} - \mathbf{e}_i\| \mid \mathbf{e}_i \in E, < \mathbf{w}, \mathbf{x} > + \, b = 0\} \tag{9.10}$$

The margin can be interpreted as the minimum distance between an example and the hyperplane. The theory of statistical learning shows that maximizing the margin provides optimal generalization behavior. Thus the maximum

margin approach computes the *optimal* hyperplane, that is, the hyperplane h^* that maximizes the margin:

$$h^* = \arg\max_h margin(h, E) \tag{9.11}$$

9.2.2 Support Vector Machines

Computing the maximum margin hyperplane is an optimization problem that is usually formalized in the following way. The first requirement is that the examples $(\mathbf{e}_i, f(\mathbf{e}_i))$ are correctly classified. Mathematically, this results in the constraints:

$$\forall \mathbf{e} \in E : f(\mathbf{e})\big(< \mathbf{w}, \mathbf{e} > + b \ \big) \ > 0 \tag{9.12}$$

Secondly, because any solution $(\mathbf{w}, b)$ to this set of equations can be scaled by multiplying with a constant, the convention is that those examples closest to the hyperplane satisfy

$$| < \mathbf{w}, \mathbf{e} > + b \ | = 1 \tag{9.13}$$

This results in the following optimization problem:

$$\begin{aligned} \min_{\mathbf{w},b} \quad & < \mathbf{w}, \mathbf{w} > \\ \text{subject to} \quad & \forall \mathbf{e} \in E : f(\mathbf{e})\big(< \mathbf{w}, \mathbf{e} > + b \ \big) \ \geqslant 1 \end{aligned} \tag{9.14}$$

It can be shown that

$$margin(h, E) = \frac{1}{\|\mathbf{w}\|} \tag{9.15}$$

(see [Schölkopf and Smola, 2002] for the proof). Therefore, minimizing $\|\mathbf{w}\|$ corresponds to maximizing the $margin(h, E)$. This is illustrated in Fig. 9.1.

The optimization problem is usually not solved in the form listed above. It is rather turned into the *dual* problem using the theory of Karush, Kuhn and Tucker and the Langragian; cf. [Cristianini and Shawe-Taylor, 2000]. This dual formulation is typically an easier optimization problem to solve:

$$\begin{aligned} \max_{(\alpha_1,\cdots,\alpha_n)} & \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} f(\mathbf{e}_i) f(\mathbf{e}_j) \alpha_i \alpha_j < \mathbf{e}_i, \mathbf{e}_j > \\ \text{subject to} \quad & \forall i : \alpha_i \geqslant 0 \text{ and } \sum_i f(\mathbf{e}_i) \alpha_i = 0 \end{aligned} \tag{9.16}$$

Several public domain quadratic programming solvers exist that solve this optimization problem. Using this formulation, the weight vector $\mathbf{w}$ is a linear combination of the data vectors:

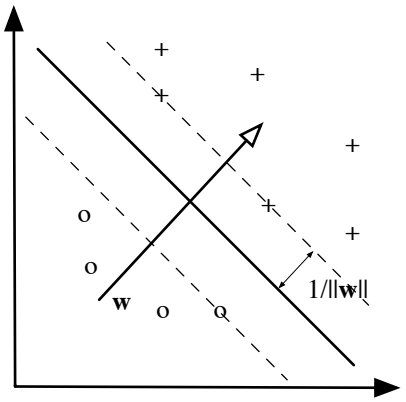


Fig. 9.1. The max margin approach and the support vectors. The positive examples are marked with a "+" and the negative ones with an "o". The maximum margin hyperplane is the middle line. The support vectors lying on the dotted lines, and the vector normal to the margin is indicated by **w**

$$\mathbf{w} = \sum_i \alpha_i f(\mathbf{e}_i)\ \mathbf{e}_i \tag{9.17}$$

Typically, many α_i will have the value 0. Those vectors $\mathbf{e}_i$ with non-zero α_i are the so-called *support vectors*. These are also the vectors that lie on the margin, that is, those for which

$$f(\mathbf{e}_i)\big(< \mathbf{w}, \mathbf{e}_i > + b \big) = 1 \tag{9.18}$$

These are the only ones that influence the position of the optimal hyperplane. The constant b can be obtained by substituting two support vectors, one for each class, in Eq. 9.18 and solving for b. The value $h(\mathbf{e})$ of a new data point e can then be computed as:

$$h(\mathbf{e}) = sgn\Big(\sum_i \alpha_i f(\mathbf{e}_i) < \mathbf{e}_i, \mathbf{e} > + b \Big) \tag{9.19}$$

Given that the α_i are non-zero only for the support-vectors, only those need to be taken into account in the above sum.

A central property of the formulation of the dual problem in Eq. 9.16 as well as the use of the resulting hyperplane for prediction in Eq. 9.19 is that they are entirely formulated in terms of inner products. To formulate the dual problem, one only needs access to the data points and to the so-called *Gram* matrix, which is the matrix containing the pairwise inner products $< \mathbf{e}_i, \mathbf{e}_j >$ of the examples.

There are numerous extensions to the basic support vector machine setting. The most important ones include an extension to cope with data sets that are not linearly separable (by introducing slack variables ξ_i for each of the instances) and several loss functions designed for regression.

9.2.3 The Kernel Trick

The support vector machine approach introduced above worked within the space $\mathbb{R}^d$, used the inner product as the kernel function and worked under the assumption that the examples were linearly separable. In this subsection, we will lift these restrictions and introduce a more formal definition of kernel functions and some of its properties.

Consider the classification problem sketched in Fig. 9.2. Using the max margin approach with the inner product kernel, there is clearly no solution to the learning task. One natural and elegant solution to this problem is to first transform the instances $\mathbf{e}$ to some vector $\Phi(\mathbf{e})$ in some other feature space and to then apply the inner product.

Example 9.3. Using the function

$$\Phi : \mathbb{R}^2 \rightarrow \mathbb{R}^3 : (x_1, y_1) \rightarrow (x_1^2, \sqrt{2}x_1x_2, x_2^2)$$

the examples are separable using a hyperplane in the transformed space $\mathbb{R}^3$.

This motivates the use of the kernel

$$K(\mathbf{x}, \mathbf{y}) = < \Phi(\mathbf{x}), \Phi(\mathbf{y}) > \tag{9.20}$$

instead of the inner product in $\mathbb{R}^2$. The interesting point about this kernel K is that Φ does not have to be computed explicitly at the vectors, because

$$K(\mathbf{x}, \mathbf{y}) = < \mathbf{x}, \mathbf{y} >^2 \tag{9.21}$$

This is called the *kernel trick*. It often results in enormous computational savings as the dimension of the transformed space can be much larger than that of the input space $\mathcal{L}_e$. The example also illustrates that kernels are essentially inner products in *some* feature space that does not necessarily coincide with the input space $\mathcal{L}_e$.

Formally, functions have to satisfy some conditions for being a kernel and for the optimization approaches sketched above to work. In particular, one typically uses so-called Mercer kernels, which are symmetric functions that are positive definite.[1] For convenience, we will continue to talk about *kernels* instead of Mercer or positive definite kernels.

[1] A symmetric function is *positive definite* if and only if $\forall m \in \mathbb{N} : \forall \mathbf{x}_1, \cdots, \mathbf{x}_m \in \mathcal{L}_e : \forall a_1, \cdots, a_m \in \mathbb{R} : \sum_{i,j=1}^{m} a_i a_j K(\mathbf{x}_i, \mathbf{x}_j) \geqslant 0$, or, equivalently, if the eigenvalues of the possible Gram matrices are non-negative.

Many interesting kernels can be constructed using the following properties and the observation that the inner product in Euclidean space is a kernel. Basically, if $H(\mathbf{x}, \mathbf{y})$ and $G(\mathbf{x}, \mathbf{y})$ are kernels on $\mathcal{L}_e \times \mathcal{L}_e$ then the functions

$$\begin{aligned} K_s(\mathbf{x}, \mathbf{y}) &= H(\mathbf{x}, \mathbf{y}) + G(\mathbf{x}, \mathbf{y}) \\ K_p(\mathbf{x}, \mathbf{y}) &= H(\mathbf{x}, \mathbf{y})G(\mathbf{x}, \mathbf{y}) \\ K_d(\mathbf{x}, \mathbf{y}) &= (H(\mathbf{x}, \mathbf{y}) + l)^d \\ K_g(\mathbf{x}, \mathbf{y}) &= exp(-\gamma(H(\mathbf{x}, \mathbf{x}) - 2H(\mathbf{x}, \mathbf{y}) + H(\mathbf{y}, \mathbf{y}))) \\ K_n(\mathbf{x}, \mathbf{y}) &= \frac{H(\mathbf{x}, \mathbf{y})}{\sqrt{H(\mathbf{x}, \mathbf{x}).H(\mathbf{y}, \mathbf{y})}} \end{aligned} \tag{9.22}$$

are also kernels,[2] where s stands for sum, p for product, d for polynomial, g for Gaussian and n for normalized.

Exercise 9.4. Show how the kernel Eq. 9.21 can be formulated as a polynomial kernel. Show also that this kernel corresponds to $< \Phi(\mathbf{x}), \Phi(\mathbf{y}) >$ with Φ defined as in Eq. 9.20.

In Sect. 9.4, we shall expand these results to obtain kernels for structured data. Before doing so, we turn our attention to distance-based learning.

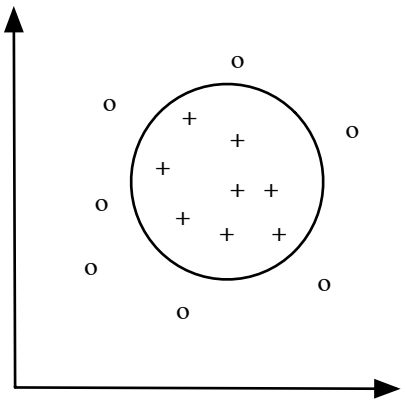


Fig. 9.2. A circular concept. The examples are marked with a "+" and the negative ones with an "o". Whereas there is no hyperplane in the original space that separates the positives from the negatives, there is one in the feature space. Mapping that hyperplane back to the original space yields the circular concept indicated

[2] For the normalized kernel, it is assumed that $K(x, x) > 0$ for all x.

9.3 Distance-Based Learning

Two of the most popular distance-based learning algorithms are the k-nearest neighbor algorithm for classification and regression and the k-means clustering algorithm. Both types of algorithms employ distance measures or metrics, which we introduce before presenting these algorithms.

9.3.1 Distance Functions

Distance measures and metrics are mathematically more easy to define than kernels, but often harder to obtain.

A distance *measure* is a function $d : \mathcal{L}_e \times \mathcal{L}_e \to \mathbb{R}$ that measures the distance between instances in e. *Distance measures* must satisfy the following requirements:

$$\begin{aligned}
\text{non-negativeness:}\quad & \forall x, y : d(x, y) \geqslant 0 && (9.23)\\
\text{reflexivity:}\quad & \forall x, y : d(x, y) = 0 && (9.24)\\
\text{strictness:}\quad & \forall x, y : d(x, y) = 0 \text{ only if } x = y && (9.25)\\
\text{symmetry:}\quad & \forall x, y : d(x, y) = d(y, x) && (9.26)
\end{aligned}$$

If a distance measure also satisfies the triangle inequality

$$\forall x, y, z : d(x, y) + d(y, z) \geqslant d(x, z) \tag{9.27}$$

then the distance measure is called a *metric*.

Exercise 9.5. Show that d_e as defined in Eq. 9.1 is a metric.

Using Eq. 9.22 one can define new kernels in terms of existing ones. This is also possible for distance metrics. For instance, if d_1 and d_2 are metrics defined on $\mathcal{L}_e \times \mathcal{L}_e$ and $c, k \in \mathbb{R}$, then

$$\begin{aligned}
d_c(x, y) &= c d_1(x, y)\\
d_{1+2}(x, y) &= d_1(x, y) + d_2(x, y)\\
d_k(x, y) &= d_1(x, y)^k \text{ with } 0 < k \leqslant 1\\
d_n(x, y) &= \frac{d_1(x, y)}{d_1(x, y) + 1}
\end{aligned} \tag{9.28}$$

are metrics, but functions such as

$$d_p(x, y) = d_1(x, y)\ d_2(x, y) \tag{9.29}$$

are not.

Exercise 9.6. Show that d_p is not a metric.

9.3.2 The *k*-Nearest Neighbor Algorithm

The k-nearest neighbor algorithm applies to both classification and regression tasks, where one is given a set of examples $E = \{(e_1, f(e_1)), \cdots, (e_n, f(e_n))\}$ of an unknown target function f, a small positive integer k, and a distance measure (or metric) $d : \mathcal{L}_e \times \mathcal{L}_e \rightarrow \mathbb{R}$. The idea is to simply store all the examples, to keep the hypothesis h implicit, and to compute the predicted value $h(e)$ for an instance e by selecting the set of k nearest examples E_k to e and computing the average value for f on E_k. This prediction process is sketched in Algo. 9.1.

Algorithm 9.1 The k-nearest neighbor algorithm

$E_k := \{(x, f(x)) \in E | d(e, x)$ is amongst the k smallest in $E\}$
Predict $h(e) := avg_{x \in E_k} f(x)$

In this algorithm, the function avg computes the average of the values of the function f in E_k. In a classification settting, avg corresponds to the mode, which is the most frequently occurring class in E_k, while in a regression setting, avg corresponds to the arithmetic average. The k-nearest neighbor algorithm is very simple to implement; it often also performs well in practice provided that the different attributes or dimensions are normalized, and that the attributes are independent of one another and relevant to the prediction task at hand; cf. standard textbooks on machine learning such as [Mitchell, 1997, Langley, 1996].

Exercise 9.7. Argue why violating one of these assumptions can give problems with the k-nearest neighbor algorithm.

Many variants of the basic k-nearest neighbor algorithm exist. One involves using all the available examples x in E but then weighting their influence in $h(e)$ using the inverse distance $1/d(e, x)$ to the example e to be classified. Other approaches compute weights that reflect the importance of the different dimensions or attributes. Further approaches devise schemes to forget some of the examples in E [Aha et al., 1991].

9.3.3 The *k*-Means Algorithm

Clustering is a machine learning task that has, so far, not been discussed in this book. When clustering, the learner is given a set of unclassified examples E, and the goal is to partition them into meaningful subsets called *clusters*. Examples falling into the same cluster should be similar to one another, whereas those falling into different classes should be dissimilar. This requirement is sometimes stated using the terms high *intra-cluster similarity* and low *inter-cluster similarity*. There are many approaches to clustering,

but given that this chapter focuses on distances, we only present a simple clustering algorithm using distance measures.

The k-means algorithm generates k clusters in an iterative way. It starts initializing these clusters by selecting k examples at random, the initial *centroids*, and then computing the distance between each example and the k centroids. The examples are then assigned to that cluster for which the distance between the example and its centroid is minimal. In the next phase, for each cluster, the example with minimum distance to the other examples in the same cluster is taken as the new centroid, and the process iterates until it converges to a stable assignment, or the number of iterations exceeds a threshold. The algorithm is summarized in Algo. 9.2.

Algorithm 9.2 The k-means algorithm

```
Select k centroids c_k from E at random
repeat
  Initialize all clusters C_k to their centroids {c_k}
  for all examples e ∈ E − {c_1, ..., c_k} do
    Let j := arg min_j d(e, c_j)
    Add e to C_j
  end for
  for all clusters C_j do
    Let i := arg min_i Σ_{e∈C_j} d(e, e_i)
    c_j := e_i
  end for
until convergence, or, max number of iterations reached
```

Several variants of the k-means algorithm exist. Analogously to the c-nearest neighbor algorithm there is the c-means algorithm for *soft* clustering. In soft clustering, all examples belong to all the clusters to a *certain degree*. The degree is, as in the c-nearest neighbor algorithm, inversely related to the distance to the centroid of the cluster. There is also the k-medoid algorithm, where the centroids are replaced by *medoids*, which correspond to the average of the instances in the cluster.

9.4 Kernels for Structured Data

In this section, we first introduce a general framework for devising kernels for structured data, the *convolution kernels* due to Haussler [1999], and we then show how it can be applied to different types of data: vectors and tuples, sets and multi-sets, strings, trees and atoms, and graphs. In the next section, we will take a similar approach for defining distance measures and metrics.

9.4.1 Convolution and Decomposition

In Haussler's framework for convolution kernels, $\mathcal{L}_e$ consists of structured instances e, and it is assumed that there is a *decomposition* relation R defined on $\mathcal{L}_e \times \mathcal{E}_1 \times \cdots \times \mathcal{E}_D$ such that the tuple

$$(e, e_1, \cdots, e_D) \in R \text{ if and only if } e_1, \cdots, e_D \text{ are parts of } e \tag{9.30}$$

So, the $\mathcal{E}_i$ denote spaces of parts of objects. We shall also write $(e_1, \cdots, e_D) \in R^{-1}(e)$.

The type of decomposition relation depends very much on the nature of the structured instances in $\mathcal{L}_e$. For instance, when dealing with sets, it is natural to define $R(e, x) = x \in e$; when dealing with strings, one can define $R(e, x_1, x_2)$ as concatenation, that is, $R(e, x_1, x_2)$ is true if and only if e is x_1 concatened with x_2; and when dealing with vectors in $\mathbb{R}^n$, $R(e, x_1, \cdots, x_n)$ is true if e denotes the vector $(x_1, \cdots, x_n)$. Observe that there may be multiple ways to decompose an object. For instance, when working with strings of length n, they can be decomposed in n different ways using the concatenation relation.

The basic result by Haussler employs kernels $K_d : \mathcal{E}_d \times \mathcal{E}_d \to \mathbb{R}$ defined at the part level to obtain kernels at the level of the instances $\mathcal{L}_e$ using a finite decomposition relation R. More formally, under the assumption that the K_d are kernels, Haussler shows that

$$K_{R,\oplus}(x, z) = \sum_{(x_1,\cdots,x_D)\in R^{-1}(x)} \sum_{(z_1,\cdots,z_D)\in R^{-1}(z)} \sum_{d=1}^{D} K_d(x_d, z_d) \tag{9.31}$$

$$K_{R,\otimes}(x, z) = \sum_{(x_1,\cdots,x_D)\in R^{-1}(x)} \sum_{(z_1,\cdots,z_D)\in R^{-1}(z)} \prod_{d=1}^{D} K_d(x_d, z_d) \tag{9.32}$$

are also kernels.

This is a very powerful result that can be applied to obtain kernels for sets, vectors, strings, trees and graphs, as we will show below.

The idea of defining a function on pairs of structured objects by first decomposing the objects into parts and then computing an aggregate on the pairwise parts is also applicable to obtaining distance measures. It is only that the aggregation function used will often differ, as will become clear throughout the remainder of this chapter.

9.4.2 Vectors and Tuples

The simplest data type to which decomposition applies is that of vectors and tuples. They form the basis for attribute-value learning and for the more complex relational learning settings.

Let us first reinvestigate the inner product over $\mathbb{R}^d$. Viewing the product in $\mathbb{R}$ as a kernel $K_\times$, the inner product in $\mathbb{R}^d$ can be viewed as a decomposition kernel (using Eq. 9.32):

$$< \mathbf{x}, \mathbf{y} >= \sum_i K_\times(x_i, y_i) \tag{9.33}$$

This definition can easily be adapted to define kernels over tuples in attribute-value form. The key difference is that, generally speaking, not all attributes will be numeric. To accomodate this difference, one only needs to define a kernel over a discrete attribute. This could be, for instance,

$$\delta(x, y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases} \tag{9.34}$$

Example 9.8. Reconsider the playtennis example of Table 4.1. Using δ in combination with the dot product yields:

$$< (\mathsf{sunny}, \mathsf{hot}, \mathsf{high}, \mathsf{no}), (\mathsf{sunny}, \mathsf{hot}, \mathsf{high}, \mathsf{yes}) >= 3$$

9.4.3 Sets and Multi-sets

When using sets, and membership as the decomposition relation, and a kernel K_{el} defined on elements, we obtain the convolution kernel K_{Set}:

$$K_{Set}(X, Y) = \sum_{x_i \in X} \sum_{y_j \in Y} K_{el}(x_i, y_j) \tag{9.35}$$

For instance, if $K_{el}(x, y) = \delta(x, y)$ (cf. Eq. 9.34), then

$$K_{Set-\delta}(X, Y) = |X \cap Y| \tag{9.36}$$

Of course, other kernels at the element level can be choosen as well. For sets, it is also convenient to *normalize* the kernel. This can be realized using Eq. 9.22, which yields

$$K_{SetNorm}(X, Y) = \frac{K_{Set}(X, Y)}{\sqrt{K_{Set}(X, X) K_{Set}(Y, Y)}} \tag{9.37}$$

Using the δ kernel, we obtain

$$K_{SetNorm-\delta}(X, Y) = \frac{|X \cap Y|}{\sqrt{|X|\ |Y|}} \tag{9.38}$$

where it is assumed that the sets are not empty.

The K_{Set} kernel naturally generalizes to multi-sets. The only required change is that in the summations the membership relation succeed multiple times for those objects that occur multiple times.

Exercise 9.9. Compute $K_{MultiSet-\delta}(\{a, a, b, b, c\}, \{a, a, b, b, b, d\})$ using $K_{el} = \delta$.

9.4.4 Strings

Let us now apply convolution kernels to strings. There are two natural ways of decomposing a string into smaller strings. The first employs the substring relation, the second the subsequence relation.

A string $S : s_0 \cdots s_n$ is a *substring* of $T : t_0 \cdots t_k$ if and only if

$$\exists j : s_0 = t_j \wedge \cdots \wedge s_n = t_{j+n} \tag{9.39}$$

that is, all symbols in the string S occur in consecutive positions in the string T.

On the other hand, a string $S : s_0 \cdots s_n$ is a *subsequence* of $T : t_0 \cdots t_k$ if and only if

$$\exists j_0 < j_1 < \cdots < j_n : s_0 = t_{j_0} \wedge \cdots \wedge s_n = t_{j_n} \tag{9.40}$$

that is all symbols in the string S occur in positions in the string T in the same order, though not necessarily consecutively. $j_n - j_0 + 1$ is called the *occurring length* of the subsequence. For instance, the string *achi* is a substring of *machine* and a subsequence of *Tai-chi* of length 5.

These two notions result in alternative relationships R. Let us first consider the substring case, where the string can be decomposed into its substrings. It is convenient to consider only substrings from Σ^n, the set of all strings over Σ containing exactly n symbols, yielding:

$$R^{-1}_{substring}(s) = \{t \in \Sigma^n | t \text{ is a substring of } s\} \tag{9.41}$$

Note that it is assumed that $R^{-1}_{substring}$ returns a multi-set. In combination with a kernel such as δ, it is easy to obtain a substring kernel.

Example 9.10. Consider the strings *john* and *jon*. When choosing $n = 2$:

$$R^{-1}_{substring}(john) = \{jo, oh, hn\}$$
$$R^{-1}_{substring}(jon) = \{jo, on\}$$

Therefore, $K_{substring}(john, jon) = 1$ because there is one common substring of length 2.

It is also interesting to view this kernel as performing a kind of feature construction or propositionalization. Essentially,

$$K_{substring}(s,t) =< \Psi_{substring}(s), \Psi_{substring}(t) >, \tag{9.42}$$

which corresponds to the inner product using the feature mapping $\Psi_{substring}(s)=$ $(\psi_{u_1}(s), \cdots, \psi_{u_k}(s))$ with $\Sigma^n = \{u_1, \cdots, u_k\}$ and $\psi_{u_i}(s) =$ number of occurrences of u_i as a substring in s. This kernel is sometimes called the *n-gram* kernel.

An interesting alternative uses the subsequence relation:

$$R^{-1}_{subseq}(s) = \{t/l | t \in \Sigma^n, t \text{ is a subsequence of } s \text{ of occurring length } l\} \tag{9.43}$$

Again, it is assumed that a multi-set is returned. The idea of the subsequence kernel is that subsequences with longer occurring lengths are penalized. This is realized using the following kernel in the convolution in Eq. 9.32:

$$K_{pen}(s/l, t/k) = \begin{cases} \lambda^{l+k} & \text{if } s = t \\ 0 & \text{otherwise} \end{cases} \tag{9.44}$$

where $0 < \lambda < 1$ is a decay factor that determines the influence of occurring length. The larger λ, the less the influence of gaps in the subsequences.

Example 9.11. Reconsider the strings *john* and *jon*. When choosing $n = 2$:

$$R^{-1}_{subseq}(john) = \{jo/2, jh/3, jn/4, oh/2, on/3, hn/2\}$$
$$R^{-1}_{subseq}(jon) = \{jo/2, jn/3, on/2\}$$

Hence, $K_{subseq,pen}(john, jon) = \lambda^{2+2} + \lambda^{3+2} + \lambda^{4+3}$ for the subsequences jo, on and jn.

This kernel corresponds to the feature mapping $\Psi_{subseq}(s) = (\phi_{u_1}(s), \cdots, \phi_{u_k}(s))$ where $\phi_{u_i}(s) = \sum_{u_i/l \in R^{-1}_{subseq}(s)} \lambda^l$.

Exercise 9.12. If one takes into account all strings of length less than or equal n instead of only those of length n, does one still obtain valid substring and subsequence kernels? Why? If so, repeat the two previous exercises for this case.

The two kernels K_{subseq} and $K_{substring}$ can be computed efficiently, that is, in time $O(n|s||t|)$, where $|s|$ denotes the length of the string s; cf. Gärtner [2003].

9.4.5 Trees and Atoms

As this book is largely concerned with logical representations, which are centered around terms and atoms, this subsection concentrates on defining kernels and distances for these structures, in particular, ground atoms. Nevertheless, due to their intimate relationship to ordered trees, the resulting kernels and distances can easily be adapted for use with ordered and unordered trees.

Because ground terms are hierachically structured, there is a natural way of decomposing them. We assume that two ground terms and the type of each of the arguments are given. One can then inductively define a kernel on ground terms $K_{Term}(t, s)$:

$$= \begin{cases} k_c(t,s) & \text{if } s \text{ and } t \text{ are constants} \\ k_f(g,h) & \text{if } t = g(t_1,\cdots,t_k) \text{ and } s = h(s_1,\cdots,s_m) \\ & \quad \text{and } g \neq h \\ k_f(g,g) + \sum_i K_{Term}(t_i,s_i) & \text{if } t = g(t_1,\cdots,t_k) \text{ and } t = g(s_1,\cdots,s_k) \end{cases} \tag{9.45}$$

This definition assumes that k_c is a kernel defined on constants and k_f a kernel defined on functor names. The kernel K_{Term} is an instance of the convolution kernel defined in Eq. 9.32. It uses the decomposition function $R^{-1}(f(t_1,t_2,\cdots,t_n)) = (f,n,t_1,\cdots,t_n)$.Notice that it is also possible to employ the product convolution kernel of Eq. 9.32, which can be obtained by replacing all sums in the recursive case by products; cf. [Menchetti et al., 2005].

Example 9.13. If $k_c = k_f = \delta$, then

$$\begin{aligned} K_{term}(r(a,b,c),r(a,c,c)) &= k_f(r,r) + [K_{term}(a,a) + K_{term}(b,c) \\ &\quad + K_{term}(c,c)] \\ &= 1 + [\delta(a,a) + \delta(b,c) + \delta(c,c)] \\ &= 1 + [1 + 0 + 1] \\ &= 3 \end{aligned}$$

It is clear that K_{term} can be computed efficiently, that is, in time linearly in the size of the two terms.

9.4.6 Graph Kernels*

Throughout this chapter, we shall focus on directed graphs in which the nodes are labeled and the edges are not, though the reader should keep in mind that it is often straightforward to adapt the framework to other types of graphs. Formally, the type of graph G considered is a triple (V,E,l), where V is the set of vertices, $E \subseteq V \times V$ is the set of edges, and the labeling function $l : V \to \Sigma$ maps vertices onto their labels from an alphabet Σ. Fig. 9.3 displays two graphs of this type, where the vertices are numbered and the labels are from the alphabet $\Sigma = \{A,B,C,\cdots,Z\}$. When the context is clear, we may omit the names of the vertices.

Subgraph Isomorphism

When working with graphs, the generality relation is specified using the notion of subgraph morphism. For logical formulae, various notions of subsumption exist that posses different properties, which is also the case for subgraph morphisms. Best known is the subgraph isomorphism, which closely corresponds to OI-subsumption, introduced in Sect. 5.5.1. In a subgraph ismorphism each node of the more general graph is mapped onto a different node in the more

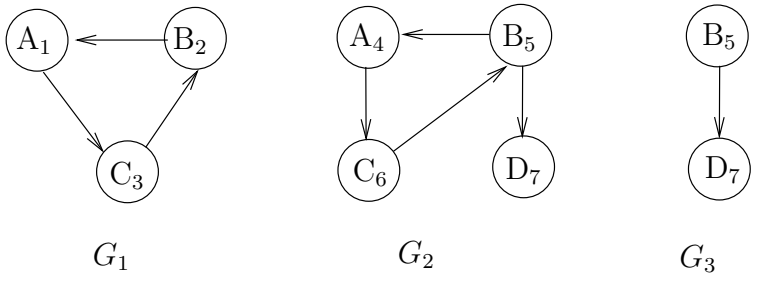


Fig. 9.3. The graphs G_1, G_2 and G_3

specific one. The nodes in the subgraph, that is, the more general graph, play the role of logical variables, and, as for OI-subsumption, two different nodes (or logical variables) must be mapped onto different nodes (or terms) in the subsumed graph. An alternative is the notion of subgraph *homeomorphism*, which is closely related to θ-subsumption, introduced in Sect. 5.4, and does allow for two nodes in the more general graph being mapped onto the same node in the more specific graph. For simplicity, we employ in this chapter a restricted form of subgraph isomorphism called *induced subgraph ismorphism*. It is restricted because it employs subgraphs that are induced by the subset relation at the level of vertices.[3] More formally, $G_s = (V_s, E_s, l_s)$ is an (induced) *subgraph* of a graph $G = (V, E, l)$ if and only if

$$V_s \subseteq V, E_s = E \cap (V_s \times V_s), \text{ and } \forall v \in V_s : l_s(v) = l(v) \tag{9.46}$$

Because the subgraph G_s is completely determined by V_s and G, we refer to G_s as the subgraph of G *induced* by V_s. The reader may want to verify that in Fig. 9.3 G_3 is the subgraph of G_2 induced by $\{5, 7\}$.

A bijective function $f : V_1 \to V_2$ is a *graph isomorphism* from $G_1 = (V_1, E_1, l_1)$ to $G_2 = (V_2, E_2, l_2)$ if and only if

$$\forall v_1 \in V_1 : l_2(f(v_1)) = l_1(v_1) \text{ and} \tag{9.47}$$

$$\forall (v, w) \in E_1 : (f(v), f(w)) \in E_2 \text{ and, vice versa} \tag{9.48}$$

$$\forall (v, w) \in E_2 : (f^{-1}(v), f^{-1}(w)) \in E_1 \tag{9.49}$$

Graph isomorphisms are important because one typically does not distinguish two graphs that are isomorphic, a convention that we will follow throughout this book. The notion is also used in the definition of the generality relation.

An injective function $f : V_1 \to V_2$ is a *subgraph isomorphism* from $G_1 = (V_1, E_1, l_1)$ to $G_2 = (V_2, E_2, l_2)$ if G_1 is an (induced) subgraph of G_2 and f is a graph isomorphism from G_1 to G_2.

[3] The more general notion of subgraph ismorphism takes into account the edges as well, but this is more complicated.

Example 9.14. There exists a subgraph isomorphism from G_1 to G_2. Consider the subgraph of G_2 induced by $\{4, 5, 6\}$. This subgraph is isomorphic to G_1 by using the function $f(1) = 4, f(2) = 5$ and $f(3) = 6$.

Exercise 9.15. * Represent the graphs of the previous example as sets of facts. Use this representation to relate the various types of graph isomorphism to the different notions of subsumption; cf. Sect. 5.9.

Graph Kernels

When developing kernels for graphs or unordered rooted trees, it is tempting to make an analogy with the string kernels. The analogy suggests decomposing the graph into its subgraphs because strings were decomposed into substrings or subsequences.

In terms of a feature mapping, this corresponds to the choice

$$\Psi_{subgraph}(G) = (\psi_{g_1}(G), \psi_{g_2}(G), \psi_{g_3}(G) \cdots) \tag{9.50}$$

for a particular (possibly infinite) enumeration of graphs $g_1, g_2, \cdots$ and the features

$$\psi_{g_i}(G) = \lambda_{|edges(g_i)|} \times |\{g \text{ subgraph of } G | g \text{ is isomorphic to } g_i\}| \tag{9.51}$$

where it is assumed that there is a sequence of decay factors $\lambda_i \in \mathbb{R}$, and the $\lambda_i > 0$ account for the possibly infinite number of graphs. The result is a valid kernel (as shown by Gärtner et al. [2003]). However, computing the kernel is NP-complete, which can be proven using a reduction from the Hamiltonian path problem, the problem of deciding whether there exists a path in a directed graph that visits all nodes exactly once; cf. Gärtner et al. [2003]. The hardness even holds when restricting the considered subgraphs to paths. Recall that a *path* in a graph is a sequence of vertices $v_0, \cdots, v_n$ in the graph such that each pair of consecutive vertices v_i, v_{i+1} is connected by an edge and no vertex occurs more than once. For instance, in G_2 of Fig. 9.3, $4, 6, 5, 7$ is a path. It is convenient to denote the paths through the string of labels $l(v_0) \cdots l(v_n)$ from Σ^*, Σ being the alphabet of labels used in the graph. For instance, the previous path can be written as $ACBD$.

Because of the computational problems with the above subgraph kernels, the kernel community has developed a number of alternative, less expressive graph kernels. One approach considers only specific types of subgraphs in the decomposition, for instance, the frequent subgraphs, or all paths up to a fixed length. A quite elegant and polynomial time computable alternative considers all (possibly infinite) walks in the graph; cf. Gärtner et al. [2003].

A *walk* in a graph is a sequence of vertices where each pair of consecutive vertices is connected but it is not required that each vertex occurs only once. As a consequence, any (finite) graph contains only a finite number of paths,

but may contain an infinite number of walks. For instance, in graph G_2, the sequence $4, 6, 5, 4, 6, 5, 7$ is a walk (in label notation $ACBACBD$).

The feature mapping considered by Gärtner et al. [2003] is based on the features ψ_s with $s \in \Sigma^*$:

$$\psi_s(G) = \sqrt{\lambda_{|s|}}|\{w|w \text{ is a walk in } G \text{ encoding } s\}| \tag{9.52}$$

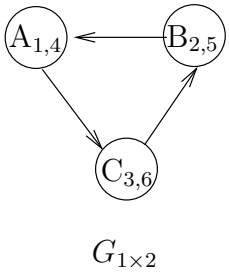
where the λ_k are decay factors as before. The interesting point about the resulting kernel K_{walk} is that it can be computed in polynomial time when choosing the geometric or the exponential series for the λ_i. (In the geometric series, $\lambda_i = \gamma^{-i}$; in the exponential one, $\lambda_i = \frac{\beta^i}{i!}$).

The algorithm for computing K_{walk} employs product graphs and adjacency matrices, which we now introduce. Given two labeled graphs $G_1 = (V_1, E_1, l_1)$ and $G_2 = (V_2, E_2, l_2)$, the *product graph* $G_{1\times 2} = G_1 \times G_2$ is defined as

$$\begin{aligned} V_{1\times 2} &= \{(v_1, v_2) \in V_1 \times V_2 | l_1(v_1) = l_2(v_2)\} \\ E_{1\times 2} &= \{((u_1, u_2), (v_1, v_2)) \in V_{1\times 2}^2 | (u_1, v_1) \in E_1 \text{ and } (u_2, v_2) \in E_2\} \\ \forall (v_1, v_2) \in V_{1\times 2} : \quad & l_{1\times 2}((v_1, v_2)) = l_1(v_1) \end{aligned} \tag{9.53}$$

The product graph corresonds to a kind of generalization of the two graphs.

Example 9.16. Fig. 9.4 contains the product graph of G_1 and G_2 of Fig. 9.3.



$G_{1\times 2}$

Fig. 9.4. The product graph $G_1 \times G_2$ of G_1 and G_2

Graphs can be conveniently represented using their adjacency matrices. For a particular ordering of the vertices $v_1, \cdots, v_n$ of a graph $G = (V, E, l)$, define the matrix of dimension $n \times n$ where $\mathbf{A}_{ij} = 1$ if $(v_i, v_j) \in E$, and 0 otherwise. It is well-known that the nth power of the adjacency matrix $\mathbf{A}_{ij}^n$ denotes the number of walks of length n from vertex i to vertex j; cf. [Rosen, 2007].

Example 9.17. The product graph $G_1 \times G_2$ in Fig. 9.4 can be represented by the matrix (for the order $(1, 4), (2, 5), (3, 6)$):

$$\mathbf{A}_{1\times 2} = \begin{pmatrix} 0\,0\,1 \\ 1\,0\,0 \\ 0\,1\,0 \end{pmatrix}$$

and the second power of the matrix is

$$\mathbf{A}^2_{1\times 2} = \begin{pmatrix} 0\,1\,0 \\ 0\,0\,1 \\ 1\,0\,0 \end{pmatrix}$$

The reader may want to verify that there are indeed only three walks of length 2 through the product graph and that these correspond to $(1,4)$ to $(2,5)$, $(2,5)$ to $(3,6)$ and $(3,6)$ to $(1,4)$.

Gaertner et al. show that

$$K_{walk}(G_1, G_2) = \sum_{i,j=1}^{|V_{1\times 2}|} \Big(\sum_{n}^{\infty} \lambda_n A^n_{1\times 2} \Big)_{ij} \tag{9.54}$$

where $A_{1\times 2}$ denotes the adjacency matrix of $G_1 \times G_2$. This series can be computed in cubic time when choosing the exponential and geometric series for λ_i.

Devising kernels for structured data types, such as graphs, trees, and sequences, is an active research area and several alternative kernels exist.

We now turn our attention to distances and metrics, where we identify similar principles that can be used to define such measures for structured data types.

9.5 Distances and Metrics

Although there exists a wide range of possible distance measures and metrics for structured and unstructured data, there are some common ideas that we will introduce throughout this section. The first idea is that of decomposition. Under conditions to be specified, metrics and distances for structured objects can be defined by decomposing the objects and then aggregating metrics or distances defined on these components. Various metrics to be presented, such as the Hausdorff and the matching distances on sets, work in this way. The second idea, related to generalization, is based on the observation that many distances try to identify common parts amongst these instances. A recent result by De Raedt and Ramon [2008], introduced in Sect. 9.5.1, shows that many distance metrics can be related to the size of the minimally general generalizations of the two instances. The larger the common part or generaliation, the smaller the distance. The distance between the two instances can be defined as the cost of the operations needed to specialize the minimally general generalizations to the instances. This is sometimes referred to as an

edit-distance. The third observation is that some distances do not rely on generalizations of the two instances but rather on a correspondence, a so-called match or matching, of the parts of one instance to those of the second one. Thus, very often, distances are related to generalization and matchings. In this section, we shall first study the relationship between metrics and generalization, and then introduce a wide variety of metrics for vectors, sets, strings, atoms and graphs.

9.5.1 Generalization and Metrics

We consider partially ordered pattern or hypothesis spaces $(\mathcal{L}_e, \preceq)$, where we write, as in Chapter 5, $s \preceq g$ (or $s \prec g$) to denote that s is more specific than (or strictly more specific than) g. Throughout this section, it will be assumed that the relation $\preceq$ is partially ordered, that is, it is *reflexive*, *anti-symmetric* and *transitive*. Recall from Chapter 5 that not all generalization relations satisfy these requirements. We shall also assume

1. that there is a function *size* $|\_| : \mathcal{L}_e \to \mathbb{R}$ that is *anti-monotonic* w.r.t. $\preceq$, that is,

$$\forall g, s \in \mathcal{L}_e : s \preceq g \to |s| \geqslant |g| \tag{9.55}$$

   and *strictly order preserving*

$$\forall g, s \in \mathcal{L}_e : s \prec g \to |s| > |g| \tag{9.56}$$

2. that the *minimally general generalizations* and *maximally general specializations*, introduced in Eq. 3.26, of two patterns are defined and yield at least one element:

$$mgg(x, y) = min\{l \in \mathcal{L}_e | x \preceq l \text{ and } y \preceq l\} \tag{9.57}$$
$$mgs(x, y) = min\{l \in \mathcal{L}_e | l \preceq x \text{ and } l \preceq s\} \tag{9.58}$$

   We also define:

$$|mgg(x, y)| = max_{m \in mgg(x,y)} |m| \tag{9.59}$$
$$|mgs(x, y)| = min_{m \in mgs(x,y)} |m| \tag{9.60}$$

We can then define the *generalization* distance $d_{\preceq}(x, y)$ on elements $x, y \in \mathcal{L}_e$:

$$d_{\preceq}(x, y) = |x| + |y| - 2|mgg(x, y)| \tag{9.61}$$

That is, to go from x to y, one should go from x to $mgg(x, y)$ and then to y. Therefore, the distance $d_{\preceq}$ can be interpreted as an *edit-distance*, measuring the cost of performing operations on x to turn it into y, or vice versa. The costs are here interpreted as the difference in size between the $mgg(x, y)$ and x or y.

An interesting property, called the *diamond inequality*, for this type of distance function states that

$$\forall x, y : d_{\preceq}(x, y) \leqslant d_{\preceq}(x, mgs(x, y)) + d_{\preceq}(mgs(x, y), y) \tag{9.62}$$

or, equivalently,

$$\forall x, y : 2|x| + 2|y| \leqslant 2|mgg(x, y)| + 2|mgs(x, y)|. \tag{9.63}$$

When a distance function satisfies the diamond equation, the distance from x to y via $mgg(x, y)$ is shorter than that via $mgs(x, y)$. This is illustrated in Fig. 9.5. De Raedt and Ramon [2008] show that the distance $d_{\preceq}$ is a metric if it satisfies the diamond inequality (and $(\mathcal{L}_e, \preceq)$ is a partial order and the size $|\_|$ satisfies the above stated requirements). Metrics obtained in this way are called *generalization distances*.

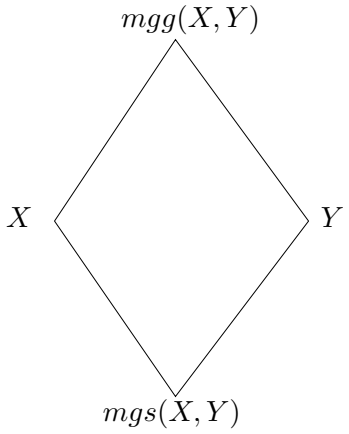


Fig. 9.5. The diamond inequality. Adapted from [De Raedt and Ramon, 2008]

This is a fairly interesting result, because it connects the key notion from logical learning (generality) with the key notion of distance-based learning (metrics). Its use will be illustrated below, when developing metrics for vectors, sets, sequences, trees and graphs.

9.5.2 Vectors and Tuples

There are two ways to obtain a distance metric on vectors or tuples. The first applies the idea of decomposition. For (symbolic) attribute-value representations, it is convenient to define

$$d_{att}(\mathbf{x}, \mathbf{y}) = \sum_i d_i(x_i, y_i) \tag{9.64}$$

The distance d_{att} will be a metric provided all the functions d_i are metrics as well. A convenient choice for the function d_i is $\overline{\delta}$, the distance corresponding to the kernel δ from Eq. 9.34:

$$\overline{\delta}(x,y) = \begin{cases} 0 & \text{if } x = y \\ 1 & \text{otherwise} \end{cases} \tag{9.65}$$

The second way of obtaining the same distance is to use the natural generalization relation on attribute-value representations and then define a generalization distance. Using the notation of logical atoms to represent attribute-value tuples, the subsumption relation on logical atoms studied in Sect. 5.3 as the generalization relation, and the number of constants in a description as size yields the same metric d_{att}.

Example 9.18. Consider the examples

$$\mathsf{playtennis}(\mathsf{sunny}, \mathsf{hot}, \mathsf{high}, \mathsf{no})$$
$$\mathsf{playtennis}(\mathsf{sunny}, \mathsf{mild}, \mathsf{low}, \mathsf{no})$$

and their generalization

$$\mathsf{playtennis}(\mathsf{sunny}, \mathsf{X}, \mathsf{Y}, \mathsf{no})$$

Thus the distance according to $d_{\preceq}$ is $2 + 2 - 2 = 2$, which is the same as that when using the d_{att} with $\overline{\delta}$.

Exercise 9.19. Prove that d_{att} is a metric when all of the d_i are metrics.

Exercise 9.20. Show that $d_{\preceq}$ satisfies the diamond inequality in this case.

When working with tuples on continuous attributes, it is convenient to use the natural distance $d_a(x,y) = |x - y|$ on $\mathbb{R}$. Decomposing the vectors along the different dimensions and adding the corresponding distances yields the distance d_{man}, known in the literature as the *Manhattan* or *cityblock* distance:

$$d_{man}(\mathbf{x}, \mathbf{y}) = \sum_i d_a(x_i, y_i) \tag{9.66}$$

9.5.3 Sets

Even though sets are – like vectors – relatively simple data structures, they are much more challenging from a distance point of view. There actually exists a wide range of possible distance measures and metrics for sets. It is instructive to develop a number of key representatives carefully and to relate them to notions of generalization and matching, which will be useful for more complex data structures.

A Simple Metric for Sets

Perhaps the simplest metric for sets is d_{Set}:

$$d_{Set}(X,Y) = |X \setminus Y| + |Y \setminus X| \tag{9.67}$$

A normalized version can be obtained by dividing by $|X \cup Y|$.

This metric can be reinterpreted as another instance of the generalization distance (using the generalization relation for sets, which was extensively studied in Chapter 3, and the natural size measures on sets). The *lgg* of two sets is their intersection, and the *glb* their union. Therefore,

$$\begin{aligned} d_{\preceq}(X,Y) &= |X| + |Y| - 2|lgg(X,Y)| \\ &= |X| - |X \cap Y| + |Y| - |X \cap Y| \\ &= |X - Y| + |Y - X| \\ &= d_{Set}(X,Y) \end{aligned} \tag{9.68}$$

The Hausdorff Metric

The distance d_{Set} is to some extent the natural metric analogue to the $K_{Set-\delta}$ kernel (cf. Eq. 9.36). We may now try to express d_{Set} using decomposition and a metric d_{el} defined on elements of the set. The natural candidate for d_{el} is the distance $\overline{\delta}$ corresponding to δ. It is, however, not easy to employ decomposition as this results in the computation of an aggregate over all *pairs* of elements. However, a reformulation that splits up the contribution over the different sets works for $\overline{\delta}$, though not in general.

$$d_{Set-d_{el}}(X,Y) = \sum_{x_i \in X} \Big(\min_{y_j \in Y} d_{el}(x_i, y_j) \Big) + \sum_{y_j \in Y} \Big(\min_{x_i \in X} d_{el}(x_i, y_j) \Big) \tag{9.69}$$

The expression $\min_{y_j \in Y} d_{el}(x_i, y_j))$ captures the idea that the distance $d(x_i, Y)$ from x_i to the set Y is given by the distance of x_i to the closest point on Y. This formulation tries to match elements of one set to elements of the other sets. The reader may want to verify that $d_{Set-\overline{\delta}} = d_{Set}$, that is, applying decomposition with the $\overline{\delta}$ distance at the level of elements results in the distance d_{Set} for sets specified in Eq. 9.67.

Exercise 9.21. Show that, in general, the distance $d_{Set-d_{el}}$ is not a metric even when d_{el} is. Hint: use $d_a(x,y) = |x - y|$ defined on $\mathbb{R}$ as d_{el}.

A refinement of the formulation in Eq. 9.69 does work. It is perhaps the most famous metric for sets, the *Hausdorff* metric. It follows a similar pattern:

$$d_{HD-d_{el}}(X,Y) = \max \Big(\max_{x_i \in X} \big(\min_{y_j \in Y} d_{el}(x_i, y_j) \big), \max_{y_j \in Y} \Big(\min_{x_i \in X} d_{el}(x_i, y_j) \Big) \Big) \tag{9.70}$$

The Hausdorff metric is based on the previous idea that the distance $d(x, Y)$ between an instance x and a set of instances Y is defined as $\min_{y \in Y} d_{el}(x, y)$. But then, the distance between two sets X and Y is defined as the maximum distance of a point in X to the set Y, that is, $\max_{x \in X} \min_{y \in Y} d_{el}(x, y)$. Unfortunately this is not a metric, because the symmetry requirement does not hold. Symmetry can be restored by taking the maximum of the distance of X to Y and that of Y to X, and this is exactly what the Hausdorff distance does. The Hausdorff metric is a metric whenever the underlying distance d_{el} is a metric.

Example 9.22. (from Ramon [2002]) Let $\mathcal{L}_e = \mathbb{R}^2$ and consider the Manhattan distance d_{man}. Let

$$X = \{(0, 0), (1, 0), (0, 1)\}$$

and

$$Y = \{(2, 2), (1, 3), (3, 3)\}$$

One can then easily verify that

$$\begin{aligned} d((0, 0), Y) &= 3 \\ d((1, 0), Y) &= 2 \\ d((0, 1), Y) &= 2 \\ d((2, 2), X) &= 3 \\ d((1, 2), X) &= 2 \\ d((3, 3), X) &= 5 \end{aligned}$$

Therefore,

$$d_{HD_{d_{man}}}(X, Y) = 5$$

Even though the Hausdorff metric is a metric, it does not capture much information about the two sets as it is completely determined by the distance of the most distant elements of the sets to the nearest neighbor in the other set.

Exercise 9.23. Show that the function

$$d_{min}(X, Y) = \min_{x_i \in X, y_j \in Y} d_{el}(x_i, y_j)$$

defined on sets is not necessarily a metric when d_{el} is.

Matching Distances

These drawbacks actually motivate the introduction of a different notion of matching between two sets. The previous two distances associate elements y in one set to the closest element in the other set X using $\min_{x \in X} d_{el}(x, y)$. This type of approach allows one element in one set to match with multiple

elements in the other set, which may be undesirable. Many distances therefore employ matchings, which associate one element in a set to at most one other element. Formally, a *matching* m between two sets X and Y is a relation $m \subseteq X \times Y$ such that every element of X and of Y is associated with at most one other element of the other set. Furthermore, if $|m| = min(|X|, |Y|)$, then the matching is called *maximal*.

Example 9.24. Consider the sets $X = \{a, b, c\}$ and $Y = \{d, e\}$. Then one matching is $\{(a, d), (c, e)\}$. It is also maximal.

Given a matching m, define the distance:

$$d(m, X, Y) = \sum_{(x,y)\in m} d_{el}(x, y) + \frac{|X - m^{-1}(Y)| + |Y - m(X)|}{2} \times M \quad (9.71)$$

where M is a large constant, larger than the largest possble distance according to d_{el} between two possible instances. So, $M > \max_{x,y} d_{el}(x, y)$.

This can be used to define a matching distance for sets:

$$d_{Set-match}(X, Y) = \min_{m\in matching(X,Y)} d(m, X, Y) \quad (9.72)$$

Example 9.25. Consider the sets $X = \{a, b, c\}$ and $Y = \{a, e\}$, the matching $\{(a, a), (c, e)\}$, the distance $\overline{\delta}$ and set $M = 4$ then

$$d(m, X, Y) = \overline{\delta}(a, a) + \overline{\delta}(c, e) + \frac{2 + 0}{2} \times 4 = 3$$

The best matching is also m. and therefore $d_{Set-match} = 2$.

This distance can be related to the generalization distance introduced before. The key difference is that we now employ a *matching*, which is identifying not only common parts but also parts that are close to one another. The matching distance can be written as:

$$d(m, X, Y) = c\big(m(X, Y)\big) + c\big(m(X, Y) \rhd X\big) + c\big(m(X, Y) \rhd Y\big) \quad (9.73)$$

where the matching part is denoted as $m(X, Y)$, a kind of "approximative" *mgg*. The function c represents an edit cost, that is the cost $c(m(X, Y))$ the matching m needs to turn X into Y, or the cost $c(m(X, Y) \rhd X)$ for turning $m(X, Y)$ into X. The formula for $d(m, X, Y)$ generalizes that for the generalization distance of Eq. 9.61, because for the generalization distance, m would be taken as the maximal size *mgg*, $c(m(X, Y)) = 0$, and $c(m(X, Y) \rhd X) = |X| - |mgg(X, Y)|$. Thus a non-zero cost can now be associated with the "matching" part of X and Y.

Computing Matching Distances for Sets*

The distance $d_{Set-match}$ can be computed in polynomial time using flow networks; cf. Ramon [2002], Ramon and Bruynooghe [2001]. A flow network is shown in Fig. 9.6. The network is essentially a directed acyclic graph, with a designated start node s and terminal node t, and, furthermore, the edges in the network are labeled as a tuple $(cap, cost) : flow$, where the capacity *cap* denotes the possible flow through the edge, and the *cost* per unit of flow that needs to be paid for flow through the edge. It is convenient to view a flow network as an abstract model of a pipeline system. The edges then correspond to the different pipes in the system, and the nodes to points where different pipes come together. The *maximum flow* problem through the network models the problem of transporting as much flow as possible from s to t. The flow $flow$ is a function from the edges in the network to $\mathbb{N}$ or $\mathbb{R}$. The inflow into a node n is the sum of the flows on the edges leading to that node, and the outflow is the sum of the flows on edges starting from that node. A flow network satisfies some natural constraints. One constraint states that the inflow is equal to the outflow for all nodes (except s and t) in the network. Another one that the flow on each edge can be at most equal to the capacity.

The *maximum flow problem* is concerned with finding

$$maxflow = \arg \max_{f \in Flow} \sum_{x} f(s, x) \tag{9.74}$$

$$= \arg \max_{f \in Flow} \sum_{y} f(y, t) \tag{9.75}$$

A simple flow network with corresponding maximum flow is shown in Fig. 9.6. Typically, one is not only interested in the maximum flow, but also in the maximum flow that has the mimimum cost. This is known as the *minimum cost maximum flow problem*. It consists of finding the maximum flow f for which

$$\sum_{(x,y) \in edges} f(x, y)\ c(x, y)$$

is minimal, where $c(x, y)$ denotes the cost for flow from x to y. The flow illustrated in Fig. 9.6 has minimum cost. There are polynomial algorithms for computing the minimum cost maximum flow [Cormen et al., 2001].

There are now two interesting applications of the maximum flow minimum cost problem to computing macthing distances between sets; cf. Ramon and Bruynooghe [2001], Ramon [2002] The first computes the distance $d_{Set-match}$ above using the flow network specified in Fig. 9.7:

- there is a node x_i for each element of the set X, as well as an extra *sink* node x_- for X,
- there is a node y_i for each element of the set Y, as well as an extra *sink* node y_- for Y,

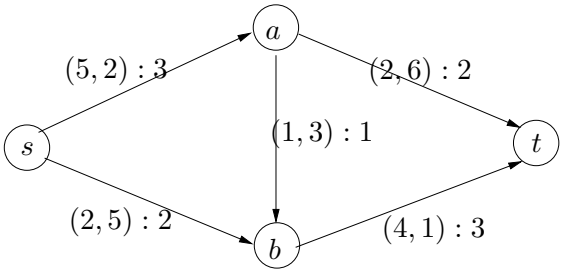


Fig. 9.6. The minimum cost maximum flow problem. The labels $(cap, cost) : flow$ denote the optimal $flow$ through the edge, the capacity cap of the edge, and the $cost$ per unit of flow through the edge

- there are edges from s to the x_i with label $(1, 0)$,
- there is an edge from s to x_- with label $(N, 0)$ where N is a constant such that $N \geqslant max(|X|, |Y|)$,
- there are edges from the y to t with label $(1, 0)$,
- there is an edge from y_- to t with label $(N, 0)$,
- there are edges from x_i to the y_i labeled $(1, d(x_i, y_i))$,
- there are edges from the x_i to y_- and from x_- to the y_i, with labels $(\infty, M/2)$, where M is the constant defined in Eq. 9.72.

Furthermore, only integers are allowed as flows. The minimum cost maximum flow then encodes the optimal matching. It essentially matches elements for which $flow(x_i, y_i) = 1$.

Jan Ramon has introduced also a variant of the distance $d_{Set-match}$. This is motivated by the fact that $d_{Set-match}$ is often dominated by the sizes of the sets. Certainly, when the sizes of the two sets are far apart, the largest contribution will come from the unmatched parts. The variant works with *weighted* sets; these are sets S where each element carries a particular weight $w_S(x)$. The idea is that the weight denotes the importance of the element in the set. The resulting distance measure can be computed using the flow network depicted in Fig. 9.7, but with the edges from s to x_i labeled by $(w_X(x_i), 0)$, and those from y_i to t by $(w_Y(y_i), 0)$. Furthermore, it is no longer required that the flow be an integer function. This formulation allows an element to match with multiple elements in the other set, and the flow will indicate the importance of the individual matches.

9.5.4 Strings

Some of the ideas for sets carry over to strings. Let us first look at distances between strings of fixed length (or size) n, that is strings in Σ^n. Let us denote the two strings $s_1 \cdots s_n$ and $t_1 \cdots t_n$. Applying decomposition leads to the Hamming distance:

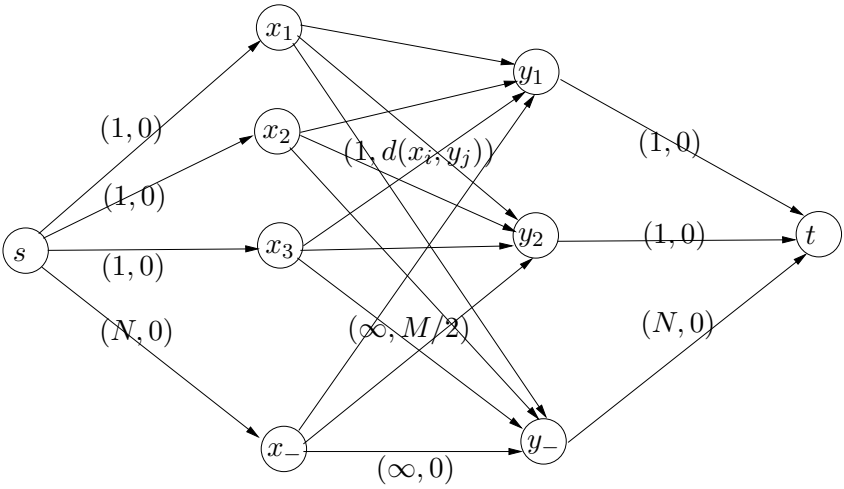


Fig. 9.7. The minimum cost maximum flow problem. The labels $(cap, cost)$ denote the capacities and the costs per flow of the edges. Adapted from [Ramon and Bruynooghe, 2001]

$$d_{Hamming}(s,t) = \sum_i \overline{\delta}(s_i, t_i) \tag{9.76}$$

Applying the generalization distance to strings leads also to a Hamming distance (up to a factor 2).

Example 9.26. Consider the strings `abcdefgh` and `xbcdyfgh`. Their *lgg* would be `-bcd-fgh` of size 6, and hence the generalization distance between these two strings is 4, whereas the Hamming distance between them is 2.

More popular distances for strings (certainly in the field of bioinformatics) are edit distances, which also produce *alignments*. They do not usually require the strings to be of equal length. The best known example of such a distance is the Levenshtein distance [1966], which formed the basis for many routinely applied techniques in bioinformatics applications, such as the famous Blast algorithm [Altschul et al., 1990]. More formally, the edit distance between two strings is defined as the minimum number of operations needed to turn one string into the other. The allowed operations are the insertion, deletion and substitution of a symbol in Σ.

Example 9.27. For instance, the string `artificial` (English) can be turned into `artificieel` (Dutch) using two operations. Inserting an `e` and substituting an `a` by an `e`. If one assumes that all operations have unit cost, the Levenshtein distance $d_{lev}(\texttt{artificieel}, \texttt{artificial}) = 2$. The parts of the two strings that match are typically depicted as an alignment. For our example, there are two possible alignments:

```
artificieel      artificieel
|||||||| +|      |||||||+ |
artifici-al      artificia-l
```

From these alignments, `artificil` is a minimally general generalization, which is called the *least common subsequence* in the literature on string matching. The matched part is indicated by the vertical bars. This edit-distance is a generalization distance, as shown by De Raedt and Ramon [2008]. Like the Hamming distance, taking the length of the string as size yields an edit distance with cost 2 for each operation.

The Levenshtein distance can be computed in polynomial time using a simple application of dynamic programming [Cormen et al., 2001]. For two strings $s_1 \cdots s_n$ and $t_1 \cdots t_m$, it employs an $(n+1) \times (m+1)$ matrix M; which is defined as follows (assuming unit costs for the edit operations):

$$\begin{aligned} M(0,0) &= 0 \\ M(i,0) &= i \text{ for } (n \geqslant i \geqslant 1) \\ M(0,j) &= j \text{ for } (m \geqslant j \geqslant 1) \\ M(i,j) &= \min \begin{cases} M(i-1,j-1) + \overline{\delta}(s_i,t_j) & \\ M(i-1,j)+1 & \text{insert for } i,j \geqslant 1 \\ M(i,j-1)+1 & \text{delete} \end{cases} \end{aligned} \tag{9.77}$$

An example computation is depicted in Table 9.1.

Table 9.1. Computing the Levenshtein distance

		a	r	t	i	f	i	c	i	a	l
	0	1	2	3	4	5	6	7	8	9	10
a	1	0	1	2	3	4	5	6	7	8	9
r	2	1	0	1	2	3	4	5	6	7	8
t	3	2	1	0	1	2	3	4	5	6	7
i	4	3	2	1	0	1	2	3	4	5	6
f	5	4	3	2	1	0	1	2	3	4	5
i	6	5	4	3	2	1	0	1	2	3	4
c	7	6	5	4	3	2	1	0	1	2	3
i	8	7	6	5	4	3	2	1	0	1	2
e	9	8	7	6	5	4	3	2	1	1	2
e	10	9	8	7	6	5	4	3	2	2	2
l	11	10	9	8	7	6	5	4	3	3	2

The best aligment can be obtained by storing in each cell $M(i,j)$ of the matrix a backward pointer to the previous best cell, that is, that points to

that cell which yielded the minimum value in the recursive case. When the algorithm terminates, the backwards pointers can be followed from the cell $M(n+1, m+1)$, which contains the minimum edit-distance.

Exercise 9.28. Compute the Levenshtein distance and the least common subsequence of `machine learning` and `machinelles lernen`.

9.5.5 Atoms and Trees

Shan-Hwei Nienhuys-Cheng [1997] introduced the following metric for ground atoms or terms $d_{term}(t, s)$

$$d_{term}(t,s) = \begin{cases} \overline{\delta}(t,s) & \text{if } t \text{ and } s \text{ are constants} \\ \overline{\delta}(g,h) & \text{if } t = g(t_1, \cdots, t_k) \text{ and } s = h(s_1, \cdots, s_m) \\ & \quad \text{and } g \neq h \\ \frac{1}{2k} \sum_i d_{term}(t_i, s_i) & \text{if } t = g(t_1, \cdots, t_k) \text{ and } s = g(s_1, \cdots, s_k) \end{cases} \tag{9.78}$$

One can arrive at this distance in various possible ways. First, following the idea of decomposition, there is a clear analogy with the kernel K_{term} in Eq. 9.45. The key difference other than the use of $\overline{\delta}$ instead of δ lies in the factor $\frac{1}{2k}$, which is needed to obtain a metric that satisfies the triangle inequality. Second, the distance can be viewed as a matching or edit-distance, cf. Eq. 9.73, where the cost of the matching is 0 and the edit cost for extending the common part by inserting arguments in the term depends on the position and depth of the term as indicated by the factor $\frac{1}{2k}$.

From a logical perspective, there is one drawback of the distance d_{term} (and also of the kernel K_{term}): they take into account neither variables nor unification.

Example 9.29.

$$d_{term}(r(a,b,d), r(a,c,c)) = \frac{2}{6}$$

and

$$d_{term}(r(a,b,b), r(a,c,c)) = \frac{2}{6}$$

This is counterintuive as the similarity between $r(a,b,b)$ and $r(a,c,c)$ is larger than that between $r(a,b,d)$ and $r(a,c,c)$. This is clear when looking at the *lgg*s. The *lgg* of the second pair of terms is more specific than that of the first one. Therefore one would expect their distance to be smaller.

Various researchers such as Hutchinson [1997] and Ramon [2002] have investigated distance metrics to alleviate this problem. To the best of the author's knowledge this problem has not yet been addressed using kernels. The approaches of Hutchinson [1997] and Ramon [2002] rely on the use of the

lgg operation that was introduced in Chapter 5. More specifically, by using as matching operation the lgg and then applying the matching distance of Eq. 9.73, one obtains:

$$d_{lgg}(t,s) = c\big(lgg(t,s) \rhd t\big) + c\big(lgg(t,s) \rhd s\big) \tag{9.79}$$

So, the distance d_{lgg} measures how far the terms are from their lgg, and $c(lgg(t,s) \rhd t)$ indicates the cost for specializing the lgg to t. This can be related to the size of the substitution θ needed so that $lgg(t,s)\theta = t$, as the following example illustrates.

Example 9.30. Applying this idea to the instances in the previous example yields

$$d_{lgg}(r(a,b,d), r(a,c,c)) = c\big(r(a,X,Y) \rhd sr(a,b,d)\big) + c\big(r(a,X,Y) \rhd r(a,c,c)\big)$$

and

$$d_{lgg}(r(a,b,b), r(a,c,c)) = c\big(r(a,X,X) \rhd r(a,b,b)\big) + c\big(r(a,X,X) \rhd r(a,c,c)\big)$$

Under any reasonable definition of the cost c, the latter expression will be smaller than the former. For instance, Hutchinson uses as c a measure on the size of the substitutions; more specifically, he assigns a weight to each functor and constant symbol, and then takes the sum of the symbols occurring on the right-hand side of the substitutions. In the example, this yields $c(r(a,X,X) \rhd r(a,b,b)) = size(\{X/a\}) = w_b$, and $c(r(a,X,Y) \rhd r(a,b,d)) = size(\{X/b, Y/d\}) = w_b + w_c$.

The distances studied by Hutchinson [1997] and Ramon [2002] capture some interesting logical intuitions but still exhibit some problems (for instance, how to measure the distance between $p(X,Y,Z)$ and $p(W,W,W)$). Furthermore, they are also quite involved, especially when applied to clauses rather than terms or atoms, which explains why we do not discuss them in more detail here.

9.5.6 Graphs

To develop a metric on graphs, it is useful to look for a notion of mimimally general generalization, or alignment, and to apply the generalization distance. For graphs, a natural notion is given by the maximal common subgraph.

Formally, a *maximal common subgraph* $m(G_1, G_2)$ of two graphs G_1 and G_2 is a graph G such that 1) there exist subgraph isomorphisms from G to G_1 and from G to G_2 and 2) there exists no graph containing more nodes than G that satisfies 1). Like kernels, we employ the restricted notion of induced subgraph ismorphism here. The notion of a maximal common subgraph is illustrated in Fig. 9.8.

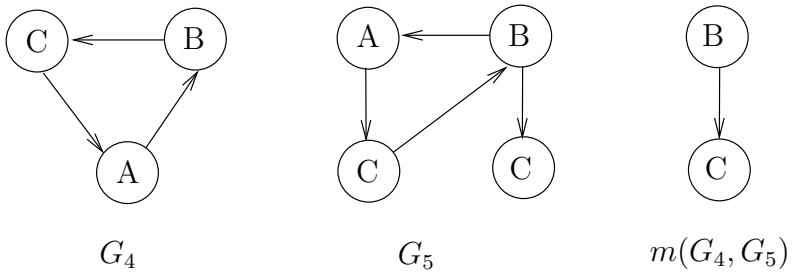


Fig. 9.8. The maximal common subgraph $m(G_4, G_5)$ of G_4 and G_5

The computation of a maximal common subgraph is an NP-complete problem and the maximal common subgraph of two graphs is not necessarily unique. Notice that a maximal common subgraph is uniquely characterized by and can therefore be represented by the (maximal) subgraph isomorphism f from G_1 to G_2 that maps vertices in G_1 to vertices in G_2. Such a maximal subgraph isomorphism is computed using a backtracking algorithm due to [McGregor, 1982], of which a variant along the lines of [Bunke et al., 2002] is summarized in Algo. 9.3.

The algorithm repeatedly adds a pair of nodes $(n_1, n_2) \in V_1 \times V_2$ to the function f, and when doing so it ensures that the resulting mapping is a feasible subgraph isomorphism by testing that the mapping f is injective and that for any two tuples (n_1, m_1) and $(n_2, m_2) \in f$: $(n_1, n_2) \in E_1$ if and only if $(m_1, m_2) \in E_2$. If the cardinality of the resulting mapping f_m is larger than that of previously considered subgraph ismorphisms, then the maximal subgraph isomorphism and corresponding size are updated. Finally, if the function f_m can still be expanded because there are still unmatched and untried vertices in V_1, and one cannot prune these refinements (because the size of f_m plus the number of such vertices is larger than $maxsize$), the procedure is called recursively.

The maximal common subgraph can be used as a minimally general generalization. A natural size measure on graphs is the number of vertices they contain. It is possible to show that the size and generality relation satisfy the necessary requirements for generalization distances; cf. [De Raedt and Ramon, 2008]. Therefore, the distance

$$d_{graph}(G_1, G_2) = |G_1| + |G_2| - 2|mcs(G_1, G_2)| \tag{9.80}$$

is a metric. Furtheremore, its normalized form

$$d_{graph-nrom}(G_1, G_2) = 1 - \frac{|mcs(G_1, G_2)|}{\max(|G_1|, |G_2|)} \tag{9.81}$$

corresponds to a well-known metric introduced by Bunke and Shearer [1998]. This distance can also be related to some type of edit-distance because the

$d_{graph-norm}(G_1, G_2)$ captures the proportion of common vertices. Bunke and Shearer argue that edit distances for graphs are – depending on the choice of cost function – not always metrics.

Example 9.31. Using these notions, $d_{graph}(G_4, G_5) = 3 + 4 - 2 \times 2 = 3$ and $d_{graph-norm}(G_4, G_5) = 1 - \frac{2}{4} = 0.5$.

Algorithm 9.3 The function $mcs(f)$ that computes a maximal subgraph isomorphism f_m of two graphs $G_i = (V_i, E_i, l_i)$ and is called with the empty relation. It uses the global parameters $maxsize$, which represents the cardinality of the relation f_m and which is initialized to 0, and $maxisomorphism$, which represents the maximal subgraph isomorphism and which is initialized to $\emptyset$

---

while there is a next pair $(n_1, n_2) \in (V_1 \times V_2)$ with $l_1(v_1) = l_2(v_2)$ **do**
  if feasible($f \cup \{(n_1, n_2)\}$) **then**
    $f_m := f \cup \{(n_1, n_2)\}$
    if $|f_m| > maxsize$ **then**
      $maxsize := size(f_m)$
      $maxisomophism := f_m$
    end if
    if nopruning(f_m) and expansionpossible(f_m) **then**
      call $mcs(f_m)$
    else
      backtrack on f_m
    end if
  end if
end while
return f_m

Exercise 9.32. Apply Algo. 9.3 to compute the maximal common subgraph of G_4 and G_5 as in Fig. 9.8.

Exercise 9.33. * Discuss the relationship between the concept of maximal common subgraph and maximally general generalization under *OI*-subsumption. (Hint: represent graphs as sets of facts.)

9.6 Relational Kernels and Distances

Whereas kernels and distances have been developed and studied for a wide variety of structured data types, such as vectors, sets, strings, trees and graphs, kernels and distances that work directly on logical and relational representations, say, clausal logic, have proven to be more challenging to develop.

Therefore, several researchers have looked into heuristic approaches to this problem.

One popular heuristic approach, due to Bisson [1992a,b], and refined by Emde and Wettschereck [1996] and Kirsten et al. [2001], has been succesfully used in a series of distance-based learning methods targeting both classification and clustering. As in the learning from entailment setting, examples (ground facts) as well as a background theory are given. The computation of a distance measure proceeds by computing for each example a tree, and then applying a distance on trees to the corresponding examples. The resulting distance is not a metric. The idea is illustrated in Ex. 9.34 due to Kirsten et al. [2001].

Example 9.34. Let us assume that the instance $\mathsf{positive}(\mathsf{set1}, 125, \mathsf{personal})$ is given together with the background theory:

$$\mathsf{cheese}(\mathsf{set1}, \mathsf{camembert}, 150, \mathsf{france}) \leftarrow$$
$$\mathsf{vineyard}(\mathsf{gallo}, \mathsf{famous}, \mathsf{large}, \mathsf{usa}) \leftarrow$$
$$\mathsf{vineyard}(\mathsf{mouton}, \mathsf{famous}, \mathsf{small}, \mathsf{france}) \leftarrow$$
$$\mathsf{wine}(\mathsf{set1}, \mathsf{gallo}, 1995, 0.5) \leftarrow$$
$$\mathsf{wine}(\mathsf{set1}, \mathsf{mouton}, 1988, 0.75) \leftarrow$$

It describes a particular basket from a gourmet shop. The instance can be turned into a tree, of which it forms the root. Subtrees are then constructed by following the foreign links on the arguments of the instance. For instance, the argument set1 occurs in three further facts, and hence these facts can be turned into children of set1. Furthermore, these facts in turn contain arguments, such as gallo, camembert and mouton, that occur in further facts, and hence can be used at the next level in the tree. The resulting tree is shown in Fig. 9.6. The process of following foreign links is typically terminated when a particular depth is reached. Given two such trees, corresponding to two instances, standard tree kernels and distance measures based on the idea of decomposition can be used.

A closely related approach is taken by Passerini et al. [2006], who provide special *visitor* predicates in addition to the background theory and the instances. The visitor predicates are then called on the instances, and the resulting proof trees are computed and passed on to a kernel. In this way, the visitor predicates encode the features that will be used by the kernel and the way that the instances will be traversed.

Transforming instances and background theory into trees forms an interesting alternative to traditional propositionalization. Rather than targeting a purely flat representation, an intermediate representation is generated that still possesses structure but is less complex than the relational one.

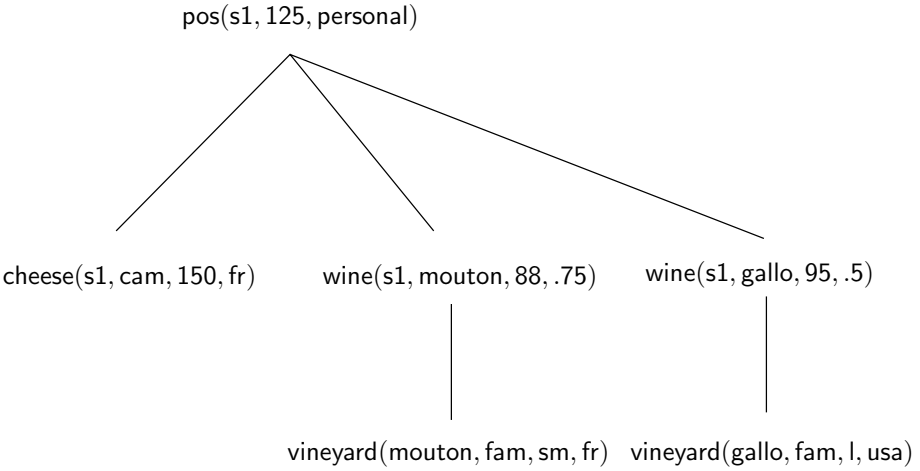


Fig. 9.9. A tree computed from an instance and the background theory. Reproduced with permission from [Kirsten et al., 2001]

9.7 Conclusions

In this chapter we have studied how logical and relational representations can be used together with support vector machines and instance-based learning techniques. After a short introduction to these classes of machine learning techniques, we have studied how distance metrics and kernels can be defined. For kernels, convolution and decomposition can be used to define kernels for complex data structures in terms of kernels for simple data structures. For distance metrics, we have investigated the relation of edit-distances to the generality relation, and we have argued that, under certain conditions, distances can be generated using a size measure and the minimimally general generalization operation. Variants of this idea, based on the notion of matching, have also been introduced. These principles have then been applied to some well known data structures, such as sets, strings, trees and graphs.

9.8 Bibliographical and Historical Notes

Many contempprary machine learning and data mining techniques employ kernel or distances. Kernels provide a basis for the popular support vector machines (see [Cristianini and Shawe-Taylor, 2000] for a gentle introduction) and distances for case-based reasoning [Aamodt and Plaza, 1994].

The introduction to kernels in Sect. 9.2 follows the introductory chapter of [Schölkopf and Smola, 2002]; and the overview of kernels for structured data follows the exposition in the overview article of [Gärtner, 2003]. A good and rather complete overview of distances and metrics for structured data

is contained in [Ramon, 2002], which also forms the basis for the present overview, though some new materials and insights from [De Raedt and Ramon, 2008] have been incorporated.

Within logic learning, the first distance measure for relational data was contributed by Bisson [1992b] for use in clustering, and later employed in the RIBL system of Emde and Wettschereck [1996]. RIBL is a relational k-nearest neighbour algorithm. Further contributions to distance-based learning were made by Kirsten and Wrobel [2000] in the context of clustering (k-means and k-medoid) and by Horvath et al. [2001] for instance-based learning. However, the distances employed in this line of work were not metrics. The systems based on this type of distance performed quite well in practice, but at the same time they motivated a more theoretical stream of work that focussed on developing metrics for relational representations, in particular the work of Ramon [2002] and Nienhuys-Cheng [1997, 1998].

Due to the importance of structured data, there is a lot of interest in developing kernels for different types of data structures. A seminal contribution in this regard is the notion of a convolution kernel due to Haussler [1999]. Since then, many kernels have been developed for a variety of data structures, including multi-instance learning [Gärtner et al., 2002], graphs [Kashima et al., 2003, Gärtner et al., 2003] and hypergraphs [Wachman and Khardon, 2007] and terms in higher-order logic [Gärtner et al., 2004]. To the best of the author's knowledge, there have only been a few attempts to integrate kernel methods with logic; cf. Muggleton et al. [2005], Passerini et al. [2006], Landwehr et al. [2006]. Some recent contributions are contained in an ongoing series of workshops on *Mining and Learning with Graphs* [Frasconi et al., 2007].

Various exciting applications of distance and kernels for structured, relational data exist. Well known are the applications to structure activity relationship prediction [Ralaivola et al., 2005], analysis of NMR spectra [Džeroski et al., 1998], and classical music expression analysis [Tobudic and Widmer, 2005].