11

Kernels for structured data: strings, trees, etc.

Probably the most important data type after vectors and free text is that of symbol strings of varying lengths. This type of data is commonplace in bioinformatics applications, where it can be used to represent proteins as sequences of amino acids, genomic DNA as sequences of nucleotides, promoters and other structures. Partly for this reason a great deal of research has been devoted to it in the last few years. Many other application domains consider data in the form of sequences so that many of the techniques have a history of development within computer science, as for example in stringology, the study of string algorithms.

Kernels have been developed to compute the inner product between images of strings in high-dimensional feature spaces using dynamic programming techniques. Although sequences can be regarded as a special case of a more general class of structures for which kernels have been designed, we will discuss them separately for most of the chapter in order to emphasise their importance in applications and to aid understanding of the computational methods. In the last part of the chapter, we will show how these concepts and techniques can be extended to cover more general data structures, including trees, arrays, graphs and so on.

Certain kernels for strings based on probabilistic modelling of the data-generating source will not be discussed here, since Chapter 12 is entirely devoted to these kinds of methods. There is, however, some overlap between the structure kernels presented here and those arising from probabilistic modelling covered in Chapter 12. Where appropriate we will point out the connections.

The use of kernels on strings, and more generally on structured objects, makes it possible for kernel methods to operate in a domain that traditionally has belonged to syntactical pattern recognition, in this way providing a bridge between that field and statistical pattern analysis.

11.1 Comparing strings and sequences

In this chapter we consider the problem of embedding two sequences in a high-dimensional space in such a way that their relative distance in that space reflects their similarity and that the inner product between their images can be computed efficiently. The first decision to be made is what similarity notion should be reflected in the embedding, or in other words what features of the sequences are revealing for the task at hand. Are we trying to group sequences by length, composition, or some other properties? What type of patterns are we looking for? This chapter will describe a number of possible embeddings providing a toolkit of techniques that either individually or in combination can be used to meet the needs of many applications.

Example 11.1 The different approaches all reduce to various ways of counting substrings or subsequences that the two strings have in common. This is a meaningful similarity notion in biological applications, since evolutionary proximity is thought to result both in functional similarity and in sequence similarity, measured by the number of insertions, deletions and symbol replacements. Measuring sequence similarity should therefore give a good indication about the functional similarity that bioinformatics researchers would like to capture.

We begin by defining what we mean by a string, substring and subsequence of symbols. Note that we will use the term substring to indicate that a string occurs contiguously within a string s, and subsequence to allow the possibility that gaps separate the different characters resulting in a non-contiguous occurrence within s.

Definition 11.2 [Strings and substrings] An *alphabet* is a finite set Σ of $|\Sigma|$ symbols. A *string*

$$s = s_1 \dots s_{|s|},$$

is any finite sequence of symbols from Σ, including the empty sequence denoted ε, the only string of length 0. We denote by Σ^n the set of all finite strings of length n, and by Σ^* the set of all strings

$$\Sigma^* = \bigcup_{n=0}^{\infty} \Sigma^n.$$

We use $[s = t]$ to denote the function that returns 1 if the strings s and t are identical and 0 otherwise. More generally $[p(s,t)]$ where $p(s,t)$ is a boolean

expression involving s and t, returns 1 if $p(s,t)$ is true and 0 otherwise. For strings s, t, we denote by $|s|$ the *length* of the string s and by st the string obtained by concatenating the strings s and t. The string t is a substring of s if there are (possibly empty) strings u and v such that

$$s = utv.$$

If $u = \varepsilon$, we say that t is a *prefix* of s, while if $v = \varepsilon$, t is known as a *suffix*. For $1 \leq i \leq j \leq |s|$, the string $s(i:j)$ is the substring $s_i \dots s_j$ of s. The substrings of length k are also referred to as *k-grams* or *k-mers*. ■

Definition 11.3 [Subsequence] We say that u is a *subsequence* of a string s, if there exist indices $\mathbf{i} = (i_1, \dots, i_{|u|})$, with $1 \leq i_1 < \cdots < i_{|u|} \leq |s|$, such that $u_j = s_{i_j}$, for $j = 1, \dots, |u|$. We will use the short-hand notation $u = s(\mathbf{i})$ if u is a subsequence of s in the positions given by $\mathbf{i}$. We denote by $|\mathbf{i}| = |u|$ the number of indices in the sequence, while the length $l(\mathbf{i})$ of the subsequence is $i_{|u|} - i_1 + 1$, that is, the number of characters of s covered by the subsequence. The empty tuple is understood to index the empty string ε. Throughout this chapter we will use the convention that bold indices $\mathbf{i}$ and $\mathbf{j}$ range over strictly ordered tuples of indices, that is, over the sets

$$I_k = \{(i_1, \dots, i_k) : 1 \leq i_1 < \cdots < i_k\} \subset \mathbb{N}^k,\ k = 0, 1, 2, \dots.$$

■

Example 11.4 For example, the words in this sentence are all strings from the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}, ..., \mathtt{z}\}$. Consider the string $s =$"`kernels`". We have $|s| = 7$, while $s(1:3) =$"`ker`" is a prefix of s, $s(4:7) =$"`nels`" is a suffix of s, and together with $s(2:5) =$"`erne`"all three are substrings of s. The string $s(1,2,4,7) =$"`kens`" is a subsequence of length 4, whose length $l(1,2,4,7)$ in s is 7. Another example of a subsequence is $s(2,4,6) =$"`enl`".

Remark 11.5 [Binary representation] There is a 1–1 correspondence between the set I_k and the binary strings with exactly k 1s. The tuple $(i_1, \dots, i_k)$ corresponds to the binary string with 1s in positions i_j, $j = 1, \dots, k$. For some of the discussions below it can be easier to think in terms of binary strings or vectors, as in Example 11.18. ■

Remark 11.6 [On strings and sequences] There is some ambiguity between the terms 'string' and 'sequence' across different traditions. They are sometimes used equivalently both in computer science and biology, but in

most of the computer science literature the term 'string' requires contiguity, whereas 'sequence' implies only order. This makes a difference when moving to substrings and subsequences. What in the biological literature are called 'sequences' are usually called 'strings' in the computer science literature. In this chapter we will follow the computer science convention, though frequently including an indication as to whether the given substring is contiguous or not in order to minimise possible confusion. ■

Embedding space All the kernels presented in this chapter can be defined by an explicit embedding map from the space of all finite sequences over an alphabet Σ to a vector space F. The coordinates of F are indexed by a subset I of strings over Σ, that is by a subset of the input space. In some cases I will be the set Σ^p of strings of length p giving a vector space of dimension $|\Sigma|^p$, while in others it will be the infinite-dimensional space indexed by Σ^*. As usual we use $\boldsymbol{\phi}$ to denote the feature mapping

$$\boldsymbol{\phi}: s \longmapsto (\boldsymbol{\phi}_u(s))_{u \in I} \in F.$$

For each embedding space F, there will be many different maps $\boldsymbol{\phi}$ to choose from. For example, one possibility is to set the value of the coordinate $\boldsymbol{\phi}_u(s)$ indexed by the string u to be a count of how many times u occurs as a contiguous substring in the input string s, while another possible choice is to count how many times u occurs as a (non-contiguous) subsequence. In this second case the weight of the count can also be tuned to reflect how many gaps there are in the different occurrences. We can also count approximate matches appropriately weighting for the number of mismatches.

In the next section we use the example of a simple string kernel to introduce some of the techniques. This will set the scene for the subsequent sections where we develop string kernels more tuned to particular applications.

11.2 Spectrum kernels

Perhaps the most natural way to compare two strings in many applications is to count how many (contiguous) substrings of length p they have in common. We define the *spectrum of order p* (or *p-spectrum*) of a sequence s to be the histogram of frequencies of all its (contiguous) substrings of length p. Comparing the p-spectra of two strings can give important information about their similarity in applications where contiguity plays an important role. We can define a kernel as the inner product of their p-spectra.

Definition 11.7 [The p-spectrum kernel] The feature space F associated with the p-spectrum kernel is indexed by $I = \Sigma^p$, with the embedding given by

$$\phi_u^p(s) = |\{(v_1, v_2) : s = v_1 u v_2\}|, \; u \in \Sigma^p.$$

The associated kernel is defined as

$$\kappa_p(s,t) = \langle \phi^p(s), \phi^p(t) \rangle = \sum_{u \in \Sigma^p} \phi_u^p(s)\, \phi_u^p(t).$$

■

Example 11.8 [2-spectrum kernel] Consider the strings `"bar"`, `"bat"`, `"car"` and `"cat"`. Their 2-spectra are given in the following table:

ϕ	ar	at	ba	ca
bar	1	0	1	0
bat	0	1	1	0
car	1	0	0	1
cat	0	1	0	1

with all the other dimensions indexed by other strings of length 2 having value 0, so that the resulting kernel matrix is:

K	bar	bat	car	cat
bar	2	1	1	0
bat	1	2	0	1
car	1	0	2	1
cat	0	1	1	2

Example 11.9 [3-spectrum kernel] As a further example, consider the following two sequences

$s =$`"statistics"`
$t =$`"computation"`

The two strings contain the following substrings of length 3.

`"sta"`, `"tat"`, `"ati"`, `"tis"`, `"ist"`, `"sti"`, `"tic"`, `"ics"`
`"com"`, `"omp"`, `"mpu"`, `"put"`, `"uta"`, `"tat"`, `"ati"`, `"tio"`, `"ion"`

and they have in common the substrings `"tat"` and `"ati"`, so their inner product would be $\kappa(s,t) = 2$.

Many alternative recursions can be devised for the calculation of this kernel with different costs and levels of complexity. We will present a few of them spread through the rest of this chapter, since they illustrate different points about the design of string kernels.

In this section we present the first having a cost $O(p|s||t|)$. Our aim in later sections will be to show how the cost can be reduced to $O(p(|s|+|t|)) = O(p\max(|s|,|t|))$ making it linear in the length of the longer sequence.

The first method will nonetheless be useful because it will introduce some methods that are important when considering more sophisticated kernels, for example ones that can tolerate partial matches and insertions of irrelevant symbols. It will also illustrate an important consideration that makes it possible to use partial results of the evaluation of one entry in the kernel matrix to speed up the computation of other entries in the same row or column. In such cases the complexity of computing the complete kernel matrix can be less than $O(\ell^2)$ times the cost of evaluating a single entry.

Computation 11.10 [p-spectrum recursion] We can first define an 'auxiliary' kernel known as the k-suffix kernel to assist in the computation of the p-spectrum kernel. The k-suffix kernel $\kappa_k^S(s,t)$ is defined by

$$\kappa_k^S(s,t) = \begin{cases} 1 & \text{if } s = s_1 u,\ t = t_1 u, \text{ for } u \in \Sigma^k \\ 0 & \text{otherwise.} \end{cases}$$

Clearly the evaluation of $\kappa_k^S(s,t)$ requires $O(k)$ comparisons, so that the p-spectrum kernel can be evaluated using the equation

$$\kappa_p(s,t) = \sum_{i=1}^{|s|-p+1} \sum_{j=1}^{|t|-p+1} \kappa_p^S(s(i:i+p), t(j:j+p))$$

in $O(p|s||t|)$ operations. ■

Example 11.11 The evaluation of the 3-spectrum kernel using the above recurrence is illustrated in the following Table 11.1 and 11.2, where each entry computes the kernel between the corresponding prefixes of the two strings.

Remark 11.12 [Computational cost] The evaluation of one row of the table for the p-suffix kernel corresponds to performing a search in the string t for the p-suffix of a prefix of s. Fast string matching algorithms such as the Knuth–Morris–Pratt algorithm can identify the matches in time $O(|t|+p) = O(|t|)$, suggesting that the complexity could be improved to

DP : κ_3^S	ε	c	o	m	p	u	t	a	t	i	o	n
ε	0	0	0	0	0	0	0	0	0	0	0	0
s	0	0	0	0	0	0	0	0	0	0	0	0
t	0	0	0	0	0	0	0	0	0	0	0	0
a	0	0	0	0	0	0	0	0	0	0	0	0
t	0	0	0	0	0	0	0	0	1	0	0	0
i	0	0	0	0	0	0	0	0	0	1	0	0
s	0	0	0	0	0	0	0	0	0	0	0	0
t	0	0	0	0	0	0	0	0	0	0	0	0
i	0	0	0	0	0	0	0	0	0	0	0	0
c	0	0	0	0	0	0	0	0	0	0	0	0
s	0	0	0	0	0	0	0	0	0	0	0	0

Table 11.1. *3-spectrum kernel between two strings.*

DP : κ_3	ε	c	o	m	p	u	t	a	t	i	o	n
ε	0	0	0	0	0	0	0	0	0	0	0	0
s	0	0	0	0	0	0	0	0	0	0	0	0
t	0	0	0	0	0	0	0	0	0	0	0	0
a	0	0	0	0	0	0	0	0	0	0	0	0
t	0	0	0	0	0	0	0	0	1	1	1	1
i	0	0	0	0	0	0	0	0	1	2	2	2
s	0	0	0	0	0	0	0	0	1	2	2	2
t	0	0	0	0	0	0	0	0	1	2	2	2
i	0	0	0	0	0	0	0	0	1	2	2	2
c	0	0	0	0	0	0	0	0	1	2	2	2
s	0	0	0	0	0	0	0	0	1	2	2	2

Table 11.2. *3-spectrum kernel between strings.*

$O(|s||t|)$ operations using this approach. We do not pursue this further since we will consider even faster implementations later in this chapter. ■

Remark 11.13 [Blended spectrum kernel] We can also consider a 'blended' version of this kernel $\tilde{\kappa}_p$, where the spectra corresponding to all values $1 \leq d \leq p$ are simultaneously compared, and weighted according to λ^d. This requires the use of an auxiliary p-suffix kernel $\tilde{\kappa}_p^S$ that records the level of similarity of the suffices of the two strings. This kernel is defined recursively as follows

$$\tilde{\kappa}_0^S(sa, tb) = 0,$$

$$\tilde{\kappa}_p^S(sa, tb) = \begin{cases} \lambda^2 \left(1 + \tilde{\kappa}_{p-1}^S(s,t)\right), & \text{if } a = b; \\ 0, & \text{otherwise.} \end{cases}$$

The blended spectrum kernel can be defined using the same formula as the p-spectrum kernel as follows

$$\tilde{\kappa}_p(s,t) = \sum_{i=1}^{|s|-p+1} \sum_{j=1}^{|t|-p+1} \tilde{\kappa}_p^S(s(i:i+p), t(j:j+p)),$$

resulting in the same time complexity as the original p-spectrum kernel. ■

As mentioned above later in this chapter we will give a more efficient method for computing the p-spectrum kernel. This will be based on a tree-like data structure known as a trie. We first turn to considering subsequences rather than substrings as features in our next example.

11.3 All-subsequences kernels

In this section we consider a feature mapping defined by all contiguous or non-contiguous subsequences of a string.

Definition 11.14 [All-subsequences kernel] The feature space associated with the embedding of the all-subsequences kernel is indexed by $I = \Sigma^*$, with the embedding given by

$$\boldsymbol{\phi}_u(s) = |\{\mathbf{i}: u = s(\mathbf{i})\}|, \ u \in I,$$

that is, the count of the number of times the indexing string u occurs as a subsequence in the string s. The associated kernel is defined by

$$\kappa(s,t) = \langle \boldsymbol{\phi}(s), \boldsymbol{\phi}(t) \rangle = \sum_{u \in \Sigma^*} \boldsymbol{\phi}_u(s) \boldsymbol{\phi}_u(t).$$

■

An explicit computation of this feature map will be computationally infeasible even if we represent $\boldsymbol{\phi}_u(s)$ by a list of its non-zero components, since if $|s| = m$ we can expect of the order of

$$\min\left(\binom{m}{k}, |\Sigma|^k\right)$$

distinct subsequences of length k. Hence, for $m > 2|\Sigma|$ the number of non-zero entries considering only strings u of length $m/2$ is likely to be at least $|\Sigma|^{m/2}$, which is exponential in the length m of the string s.

Remark 11.15 [Set intersection] We could also consider this as a case of a kernel between multisets, as outlined in Chapter 9. The measure of the intersection is a valid kernel. Neither definition suggests how the computation of the kernel might be effected efficiently. ■

Example 11.16 All the (non-contiguous) subsequences in the words `"bar"`, `"baa"`, `"car"` and `"cat"` are given in the following two tables:

ϕ	ε	a	b	c	r	t	aa	ar	at	ba	br	bt
bar	1	1	1	0	1	0	0	1	0	1	1	0
baa	1	2	1	0	0	0	1	0	0	2	0	0
car	1	1	0	1	1	0	0	1	0	0	0	0
cat	1	1	0	1	0	1	0	0	1	0	0	0

ϕ	ca	cr	ct	bar	baa	car	cat
bar	0	0	0	1	0	0	0
baa	0	0	0	0	1	0	0
car	1	1	0	0	0	1	0
cat	1	0	1	0	0	0	1

and since all other (infinite) coordinates must have value zero, the kernel matrix is

K	bar	baa	car	cat
bar	8	6	4	2
baa	6	12	3	3
car	4	3	8	4
cat	2	3	4	8

Notice that in general, the diagonal elements of this kernel matrix can vary greatly with the length of a string and even as illustrated here between words of the same length, when they have different numbers of repeated subsequences. Of course this effect can be removed if we choose to normalise this kernel. Notice also that all the entries are always positive placing all the feature vectors in the positive orthant. This suggests that centering may be advisable in some situations (see Code Fragment 5.2).

Evaluating the kernel We now consider how the kernel κ can be evaluated more efficiently than via an explicit computation of the feature vectors. This will involve the definition of a recursive relation similar to that derived for ANOVA kernels in Chapter 9. Its computation will also follow similar dynamic programming techniques in order to complete a table of values with

the resulting kernel evaluation given by the final entry in the table. In presenting this introductory kernel, we will introduce terminology, definitions and techniques that will be used throughout the rest of the chapter.

First consider restricting our attention to the feature indexed by a string u. The contribution of this feature to the overall inner product can be expressed as

$$\phi_u(s)\,\phi_u(t) = \sum_{\mathbf{i}:u=s(\mathbf{i})} 1 \sum_{\mathbf{j}:u=t(\mathbf{j})} 1 = \sum_{(\mathbf{i},\mathbf{j}):u=s(\mathbf{i})=t(\mathbf{j})} 1,$$

where it is understood that $\mathbf{i}$ and $\mathbf{j}$ range over strictly ordered tuples of indices. Hence, the overall inner product can be expressed as

$$\kappa(s,t) = \langle \boldsymbol{\phi}(s), \boldsymbol{\phi}(t)\rangle = \sum_{u\in I} \sum_{(\mathbf{i},\mathbf{j}):u=s(\mathbf{i})=t(\mathbf{j})} 1 = \sum_{(\mathbf{i},\mathbf{j}):s(\mathbf{i})=t(\mathbf{j})} 1. \tag{11.1}$$

The key to the recursion is to consider the addition of one extra symbol a to one of the strings s. In the final sum of equation (11.1) there are two possibilities for the tuple $\mathbf{i}$: either it is completely contained within s or its final index corresponds to the final a. In the first case the subsequence is a subsequence of s, while in the second its final symbol is a. Hence, we can split this sum into two parts

$$\sum_{(\mathbf{i},\mathbf{j}):sa(\mathbf{i})=t(\mathbf{j})} 1 = \sum_{(\mathbf{i},\mathbf{j}):s(\mathbf{i})=t(\mathbf{j})} 1 + \sum_{u:t=uav} \sum_{(\mathbf{i},\mathbf{j}):s(\mathbf{i})=u(\mathbf{j})} 1, \tag{11.2}$$

where for the second contribution we have used the fact that the last character of the subsequence is a which must therefore occur in some position within t.

Computation 11.17 [All-subsequences kernel] Equation (11.2) leads to the following recursive definition of the kernel

$$\begin{aligned} \kappa(s,\varepsilon) &= 1, \\ \kappa(sa,t) &= \kappa(s,t) + \sum_{k:t_k=a} \kappa(s, t(1:k-1)) \end{aligned} \tag{11.3}$$

where we have used the fact that every string contains the empty string ε exactly once. By the symmetry of kernels, an analogous recurrence relation can be given for $\kappa(s,ta)$

$$\begin{aligned} \kappa(\varepsilon,t) &= 1 \\ \kappa(s,ta) &= \kappa(s,t) + \sum_{k:s_k=a} \kappa(s(1:k-1),t). \end{aligned}$$

Similar symmetric definitions will exist for other recurrences given in this chapter, though in future we will avoid pointing out these alternatives. ■

An intuitive reading of the recurrence states that common subsequences are either contained completely within s and hence included in the first term of the recurrence or must end in the symbol a, in which case their final symbol can be matched with occurrences of a in the second string.

Example 11.18 Consider computing the kernel between the strings $s =$ `"gatt"` and $t =$`"cata"`, where we add the symbol $a =$`"a"` to the string `"gatt"` to obtain the string $sa =$ `"gatta"`. The rows of the tables are indexed by the pairs $(\mathbf{i}, \mathbf{j})$ such that $sa(\mathbf{i}) = t(\mathbf{j})$ with those not involving the final character of sa listed in the first table, while those involving the final character are given in the second table. The binary vectors below each string show the positions of the entries in the vectors $\mathbf{i}$ and $\mathbf{j}$.

g	a	t	t	a	$sa(\mathbf{i}) = t(\mathbf{j})$	c	a	t	a
0	0	0	0	0	ε	0	0	0	0
0	1	1	0	0	at	0	1	1	0
0	1	0	1	0	at	0	1	1	0
0	1	0	0	0	a	0	1	0	0
0	0	1	0	0	t	0	0	1	0
0	0	0	1	0	t	0	0	1	0
0	1	0	0	0	a	0	0	0	1

g	a	t	t	a	$sa(\mathbf{i}) = t(\mathbf{j})$	c	a	t	a
0	0	0	0	1	a	0	1	0	0
0	0	0	0	1	a	0	0	0	1
0	1	0	0	1	aa	0	1	0	1
0	0	0	1	1	ta	0	0	1	1
0	0	1	0	1	ta	0	0	1	1
0	1	1	0	1	ata	0	1	1	1
0	1	0	1	1	ata	0	1	1	1

Hence, we see that the 14 rows implying $\kappa(sa, t) = 14$ are made up of the 7 rows of the first table giving $\kappa(s, t) = 7$ plus $\kappa($`"gatt"`, `"c"`$) = 1$ given by the first row of the second table and $\kappa($`"gatt"`, `"cat"`$) = 6$ corresponding to rows 2 to 7 of the second table.

Cost of the computation As for the case of the ANOVA kernels discussed in Chapter 9 a recursive evaluation of this kernel would be very expensive.

Clearly, a similar strategy of computing the entries into a table based on dynamic programming methods will result in an efficient computation. The table has one row for each symbol of the string s and one column for each symbol of the string t; the entries $\kappa_{ij} = \kappa(s(1:i), t(1:j))$ give the kernel evaluations on the various prefices of the two strings:

DP	ε	t_1	t_2	$\cdots$	t_m
ε	1	1	1	$\cdots$	1
s_1	1	κ_{11}	κ_{12}	$\cdots$	κ_{1m}
s_2	1	κ_{21}	κ_{22}	$\cdots$	κ_{2m}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
s_n	1	κ_{n1}	κ_{n2}	$\cdots$	$\kappa_{nm} = \kappa(s,t)$

The first row and column are given by the base case of the recurrence. The general recurrence of (11.3) allows us to compute the (i,j)th entry as the sum of the $(i-1,j)$th entry together with all the entries $(i-1,k-1)$ with $1 \leq k < j$ for which $t_k = s_i$. Clearly, filling the rows in sequential order ensures that at each stage the values required to compute the next entry have already been computed at an earlier stage.

The number of operations to evaluate the single (i,j)th entry is $O(j)$ since we must check through all the symbols in t up to jth. Hence, to fill the complete table will require $O\left(|s|\,|t|^2\right)$ operations using this still slightly naive approach.

Example 11.19 Example 11.18 leads to the following table:

DP	ε	g	a	t	t	a
ε	1	1	1	1	1	1
c	1	1	1	1	1	1
a	1	1	2	2	2	3
t	1	1	2	4	6	7
a	1	1	3	5	7	14

Speeding things up We can improve the complexity still further by observing that as we fill the ith row the sum

$$\sum_{k \leq j : t_k = s_i} \kappa(s(1:i-1), t(1:k-1))$$

required when we reach the jth position could have been precomputed into an array P by the following computation

$$
\begin{array}{l}
\text{last} = 0; \\
P(0) = 0; \\
\text{for } k = 1 : m \\
\quad P(k) = P(\text{last}); \\
\quad \text{if } t_k = s_i \text{ then} \\
\qquad P(k) = P(\text{last}) + \text{DP}(i-1, k-1); \\
\qquad \text{last} = k; \\
\quad \text{end} \\
\text{end}
\end{array}
$$

Using the array p we can now compute the next row with the simple loop

$$
\begin{array}{l}
\text{for } k = 1 : m \\
\quad \text{DP}(i,k) = \text{DP}(i-1,k) + P(k); \\
\text{end}
\end{array}
$$

We now summarise the algorithm.

Algorithm 11.20 [All–non-contiguous subsequences kernel] The all–non-contiguous subsequences kernel is computed in Code Fragment 11.1. ■

Input	strings s and t of lengths n and m
Process	for $j = 1 : m$
2	$\quad \text{DP}(0, j) = 1;$
3	end
4	for $i = 1 : n$
5	$\quad \text{last} = 0;\ P(0) = 0;$
6	$\quad$ for $k = 1 : m$
7	$\qquad P(k) = P(\text{last});$
8	$\qquad$ if $t_k = s_i$ then
9	$\qquad\quad P(k) = P(\text{last}) + \text{DP}(i-1, k-1)$
10	$\qquad\quad \text{last} = k;$
11	$\qquad$ end
12	$\quad$ end
13	$\quad$ for $k = 1 : m$
14	$\qquad \text{DP}(i,k) = \text{DP}(i-1,k) + P(k);$
15	$\quad$ end
16	end
Output	kernel evaluation $\kappa(s,t) = \text{DP}(n,m)$

Code Fragment 11.1. Pseudocode for the all-non-contiguous subsequences kernel.

Example 11.21 The array p for each row is interleaved with the evaluations of DP for the Example 11.18 in the following table:

DP	ε	g	a	t	t	a
ε	1	1	1	1	1	1
p	0	0	0	0	0	0
c	1	1	1	1	1	1
p	0	0	1	1	1	2
a	1	1	2	2	2	3
p	0	0	0	2	4	4
t	1	1	2	4	6	7
p	0	0	1	1	1	7
a	1	1	3	5	7	14

Cost of the computation Since the complexity of both loops individually is $O(|t|)$ their complexity performed sequentially is also $O(|t|)$, making the overall complexity $O(|s||t|)$.

Algorithm 11.20 is deceptively simple. Despite this simplicity it is giving the exponential improvement in complexity that we first observed in the ANOVA computation. We are evaluating an inner product in a space whose dimension is exponential in $|s|$ and in $|t|$ with just $O(|s||t|)$ operations.

Remark 11.22 [Computational by-products] We compute the kernel $\kappa(s,t)$ by solving the *more general* problem of computing $\kappa(s(1:i), t(1:j))$ for all values of $i \leq |s|$ and $j \leq |t|$. This means that at the end of the computation we could also read off from the table the evaluation of the kernel between any prefix of s and any prefix of t. Hence, from the table of Example 11.19 we can see that $\kappa(\texttt{"gatt"}, \texttt{"cat"}) = 6$ by reading off the value of DP $(4,3)$. ■

11.4 Fixed length subsequences kernels

We can adapt the recursive relation presented above in a number of ways, until we are satisfied that the feature space implicitly defined by the kernel is tuned to the particular task at hand.

Our first adaptation reduces the dimensionality of the feature space by only considering non-contiguous substrings that have a given *fixed* length p.

Definition 11.23 [Fixed Length Subsequences Kernel] The feature space associated with the fixed length subsequences kernel of length p is indexed

by Σ^p, with the embedding given by

$$\phi_u^p(s) = |\{\mathbf{i}: u = s(\mathbf{i})\}|, \; u \in \Sigma^p.$$

The associated kernel is defined as

$$\kappa_p(s,t) = \langle \phi^p(s), \phi^p(t) \rangle = \sum_{u \in \Sigma^p} \phi_u^p(s) \phi_u^p(t).$$

■

Evaluating the kernel We can perform a similar derivation as for the all-subsequences kernel except that we must now restrict the index sequences to the set I_p of those with length p

$$\kappa(s,t) = \langle \phi(s), \phi(t) \rangle = \sum_{u \in \Sigma^p} \sum_{(\mathbf{i},\mathbf{j}): u = s(\mathbf{i}) = t(\mathbf{j})} 1 = \sum_{(\mathbf{i},\mathbf{j}) \in I_p \times I_p : s(\mathbf{i}) = t(\mathbf{j})} 1.$$

Following the derivations of the previous section we arrive at

$$\sum_{(\mathbf{i},\mathbf{j}) \in I_p \times I_p : sa(\mathbf{i}) = t(\mathbf{j})} 1 = \sum_{(\mathbf{i},\mathbf{j}) \in I_p \times I_p : s(\mathbf{i}) = t(\mathbf{j})} 1 + \sum_{u : t = uav} \sum_{(\mathbf{i},\mathbf{j}) \in I_{p-1} \times I_{p-1} : s(\mathbf{i}) = u(\mathbf{j})} 1,$$

where, as before, the lengths of the sequences considered in the two components of the sum differ.

Computation 11.24 [Fixed length subsequence kernel] This leads to the following recursive definition of the fixed length subsequences kernel

$$\begin{aligned} \kappa_0(s,t) &= 1, \\ \kappa_p(s,\varepsilon) &= 0, \text{ for } p > 0, \\ \kappa_p(sa,t) &= \kappa_p(s,t) + \sum_{k : t_k = a} \kappa_{p-1}(s, t(1:k-1)). \end{aligned} \tag{11.4}$$

■

Note that now we not only have a recursion over the prefixes of the strings, but also over the lengths of the subsequences considered. The following algorithm includes this extra recursion to compute the kernel by storing the evaluation of the previous kernel in the table DPrec.

Algorithm 11.25 [Fixed length subsequences kernel] The fixed length subsequences kernel is computed in Code Fragment 11.2. ■

Input	strings s and t of lengths n and m length p
Process	$\mathrm{DP}(0:n, 0:m) = 1;$
2	for $l = 1:p$
3	$\mathrm{DPrec} = \mathrm{DP};$
4	for $j = 1:m$
5	$\mathrm{DP}(0, j) = 1;$
6	end
7	for $i = 1:n-p+l$
8	last $= 0$; $P(0) = 0;$
9	for $k = 1:m$
10	$P(k) = P(\mathrm{last});$
11	if $t_k = s_i$ then
12	$P(k) = P(\mathrm{last}) + \mathrm{DPrec}(i-1, k-1);$
13	last $= k$;
14	end
15	end
16	for $k = 1:m$
17	$\mathrm{DP}(i, k) = \mathrm{DP}(i-1, k) + P(k);$
18	end
19	end
20	end
Output	kernel evaluation $\kappa_p(s, t) = \mathrm{DP}(n, m)$

Code Fragment 11.2. Pseudocode for the fixed length subsequences kernel.

Cost of the computation At the pth stage we must be able to access the row above both in the current table and in the table DPrec from the previous stage. Hence, we must create a series of tables by adding an extra loop to the overall algorithm making the overall complexity $O(p|s||t|)$. On the penultimate stage when $l = p - 1$ we need only compute up to row $n - 1$, since this is all that is required in the final loop with $l = p$. In the implementation given above we have made use of this fact at each stage, so that for the lth stage we have only computed up to row $n - p + l$, for $l = 1, \ldots, p$.

Example 11.26 Using the strings from Example 11.18 leads to the following computations for $p = 0, 1, 2, 3$ given in Table 11.3. Note that the sum of all four tables is identical to that for the all-subsequences kernel. This is because the two strings do not share any sequences longer than 3 so that summing over lengths $0, 1, 2, 3$ subsumes all the common subsequences. This also suggests that if we compute the p subsequences kernel we can, at almost no extra cost, compute the kernel κ_l for $l \leq p$. Hence, we have the flexibility

DP : κ_0	ε	g	a	t	t	a	DP : κ_1	ε	g	a	t	t	a
ε	1	1	1	1	1	1	ε	0	0	0	0	0	0
c	1	1	1	1	1	1	c	0	0	0	0	0	0
a	1	1	1	1	1	1	a	0	0	1	1	1	2
t	1	1	1	1	1	1	t	0	0	1	2	3	4
a	1	1	1	1	1	1	a	0	0	2	3	4	6
DP : κ_2	ε	g	a	t	t	a	DP : κ_3	ε	g	a	t	t	a
ε	0	0	0	0	0	0	ε	0	0	0	0	0	0
c	0	0	0	0	0	0	c	0	0	0	0	0	0
a	0	0	0	0	0	0	a	0	0	0	0	0	0
t	0	0	0	1	2	2	t	0	0	0	0	0	0
a	0	0	0	1	2	5	a	0	0	0	0	0	2

Table 11.3. *Computations for the fixed length subsequences kernel.*

to define a kernel that combines these different subsequences kernels with different weightings $a_l \geq 0$

$$\kappa(s,t) = \sum_{l=1}^{p} a_l \kappa_l(s,t).$$

The only change to the above algorithm would be that the upper bound of the loop for the variable i would need to become n rather than $n-p+l$ and we would need to accumulate the sum at the end of the l loop with the assignment

$$\text{Kern} = \text{Kern} + a(l)\,\text{DP}(n,m),$$

where the variable Kern is initialised to 0 at the very beginning of the algorithm.

11.5 Gap-weighted subsequences kernels

We now move to a more general type of kernel still based on subsequences, but one that weights the occurrences of subsequences according to how spread out they are. In other words the kernel considers the degree of presence of a subsequence to be a function of how many gaps are interspersed within it. The computation of this kernel will follow the approach developed in the previous section for the fixed length subsequences kernel. Indeed we also restrict consideration to fixed length subsequences in this section.

The kernel described here has often been referred to as the *string kernel* in

the literature, though this name has also been used to refer to the p-spectrum kernel. We have chosen to use long-winded descriptive names for the kernels introduced here in order to avoid further proliferating the existing confusion around different names. We prefer to regard 'string kernel' as a generic term applicable to all of the kernels covered in this chapter.

The key idea behind the gap-weighted subsequences kernel is still to compare strings by means of the subsequences they contain – the more subsequences in common, the more similar they are – but rather than weighting all occurrences equally, the degree of contiguity of the subsequence in the input string s determines how much it will contribute to the comparison.

Example 11.27 For example: the string `"gon"` occurs as a subsequence of the strings `"gone"`, `"going"` and `"galleon"`, but we consider the first occurrence as more important since it is contiguous, while the final occurrence is the weakest of all three.

The feature space has the same coordinates as for the fixed subsequences kernel and hence the same dimension. In order to deal with non-contiguous substrings, it is necessary to introduce a decay factor $\lambda \in (0,1)$ that can be used to weight the presence of a certain feature in a string. Recall that for an index sequence $\mathbf{i}$ identifying the occurrence of a subsequence $u = s(\mathbf{i})$ in a string s, we use $l(\mathbf{i})$ to denote the length of the string in s. In the gap-weighted kernel, we weight the occurrence of u with the exponentially decaying weight $\lambda^{l(\mathbf{i})}$.

Definition 11.28 [Gap-weighted subsequences kernel] The feature space associated with the gap-weighted subsequences kernel of length p is indexed by $I = \Sigma^p$, with the embedding given by

$$\phi_u^p(s) = \sum_{\mathbf{i}:u=s(\mathbf{i})} \lambda^{l(\mathbf{i})},\ u \in \Sigma^p.$$

The associated kernel is defined as

$$\kappa_p(s,t) = \langle \boldsymbol{\phi}^p(s), \boldsymbol{\phi}^p(t) \rangle = \sum_{u \in \Sigma^p} \phi_u^p(s)\, \phi_u^p(t).$$

■

Remark 11.29 [Two limits] Observe that we can recover the fixed length subsequences kernel by choosing $\lambda = 1$, since the weight of all occurrences

will then be 1 so that

$$\phi_u^p(s) = \sum_{\mathbf{i}:u=s(\mathbf{i})} 1^{l(\mathbf{i})} = |\{\mathbf{i}: u = s(\mathbf{i})\}|, \; u \in \Sigma^p.$$

On the other hand, as $\lambda \to 0$, the kernel approximates the p-spectrum kernel since the relative weighting of strings longer than p tends to zero. Hence, we can view the gap-weighted kernel as interpolating between these two kernels. ■

Example 11.30 Consider the simple strings "cat", "car", "bat", and "bar". Fixing $p = 2$, the words are mapped as follows:

ϕ	ca	ct	at	ba	bt	cr	ar	br
cat	λ^2	λ^3	λ^2	0	0	0	0	0
car	λ^2	0	0	0	0	λ^3	λ^2	0
bat	0	0	λ^2	λ^2	λ^3	0	0	0
bar	0	0	0	λ^2	0	0	λ^2	λ^3

So the unnormalised kernel between "cat" and "car" is κ("cat","car") = λ^4, while the normalised version is obtained using

$$\kappa(\texttt{"cat"}, \texttt{"cat"}) = \kappa(\texttt{"car"}, \texttt{"car"}) = 2\lambda^4 + \lambda^6$$

as $\hat{\kappa}$("cat","car") = $\lambda^4/(2\lambda^4 + \lambda^6) = (2 + \lambda^2)^{-1}$.

11.5.1 Naive implementation

The computation of this kernel will involve both techniques developed for the fixed subsequences kernel and for the p-spectrum kernel. When computing the p-spectrum kernel we considered substrings that occurred as a suffix of a prefix of the string. This suggests considering subsequences whose final occurrence is the last character of the string. We introduce this as an auxiliary kernel.

Definition 11.31 [Gap-weighted suffix subsequences kernel] The feature space associated with the gap-weighted subsequences kernel of length p is indexed by $I = \Sigma^p$, with the embedding given by

$$\phi_u^{p,S}(s) = \sum_{\mathbf{i} \in I_p^{|s|}: u=s(\mathbf{i})} \lambda^{l(\mathbf{i})}, \; u \in \Sigma^p,$$

where we have used I_p^k to denote the set of p-tuples of indices $\mathbf{i}$ with $i_p = k$. The associated kernel is defined as

$$\kappa_p^S(s,t) = \left\langle \boldsymbol{\phi}^{p,S}(s), \boldsymbol{\phi}^{p,S}(t) \right\rangle = \sum_{u \in \Sigma^p} \boldsymbol{\phi}_u^{p,S}(s)\, \boldsymbol{\phi}_u^{p,S}(t).$$

■

Example 11.32 Consider again the simple strings "cat", "car", "bat", and "bar". Fixing $p = 2$, the words are mapped for the suffix version as follows:

$\boldsymbol{\phi}_u^{2,S}(s)$	ca	ct	at	ba	bt	cr	ar	br
cat	0	λ^3	λ^2	0	0	0	0	0
car	0	0	0	0	0	λ^3	λ^2	0
bat	0	0	λ^2	0	λ^3	0	0	0
bar	0	0	0	0	0	0	λ^2	λ^3

Hence, the suffix kernel between "cat" and "car" is κ_2^S("cat","car") $= 0$, while suffix kernel between "cat" and "bat" is κ_2^S("cat","bat") $= \lambda^4$.

Figure 11.1 shows a visualisation of a set of strings derived from an embedding given by the gap-weighted subsequences kernel of length 4, with $\lambda = 0.8$. As in the case of the p-spectrum kernel, we can express the gap-

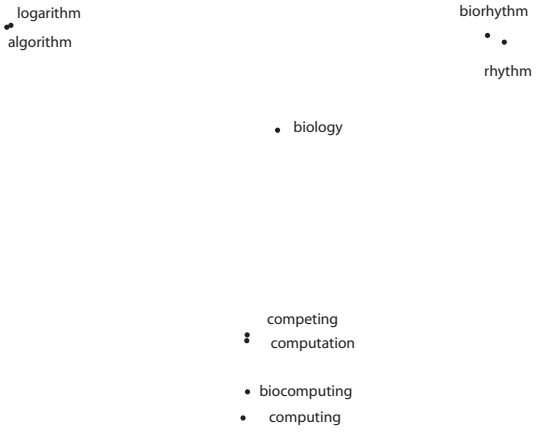

Fig. 11.1. Visualisation of the embedding created by the gap-weighted subsequences kernel.

weighted subsequences kernel in terms of its suffix version as

$$\kappa_p(s,t) = \sum_{i=1}^{|s|}\sum_{j=1}^{|t|} \kappa_p^S(s(1:i), t(1:j)),$$

since

$$\begin{aligned}
\kappa_p(s,t) &= \sum_{u\in\Sigma^p}\sum_{\mathbf{i}\in I_p:u=s(\mathbf{i})}\lambda^{l(\mathbf{i})}\sum_{\mathbf{j}\in I_p:u=t(\mathbf{j})}\lambda^{l(\mathbf{j})} = \sum_{(\mathbf{i},\mathbf{j})\in I_p\times I_p:s(\mathbf{i})=t(\mathbf{j})}\lambda^{l(\mathbf{i})+l(\mathbf{j})} \\
&= \sum_{i=1}^{|s|}\sum_{j=1}^{|t|}\sum_{(\mathbf{i},\mathbf{j})\in I_p^i\times I_p^j:s(\mathbf{i})=t(\mathbf{j})}\lambda^{l(\mathbf{i})+l(\mathbf{j})} \\
&= \sum_{i=1}^{|s|}\sum_{j=1}^{|t|}\kappa_p^S(s(1:i), t(1:j)).
\end{aligned}$$

Recall that $[u=v]$ denotes the function that returns 1 if the strings u and v are identical and 0 otherwise. We can therefore evaluate the suffix version for the case $p=1$ simply as

$$\kappa_1^S(s,t) = \left[s_{|s|} = t_{|t|}\right]\lambda^2.$$

We can now devise a recursion for the suffix version of the kernel by observing that, for extensions of s and t by single symbols a and b respectively, the suffix version is only non-zero if $a=b$, since the last character of the subsequence is constrained to occur as the final symbol.

The pairs $(\mathbf{i},\mathbf{j})$ of subsequences can now be divided according to where their penultimate symbols occurred in the two strings. It follows that when $a=b$ and $p>1$ the value of the kernel can be obtained by summing the $(p-1)$th kernel over all pairs of positions in s and t with appropriate weightings. This is shown in the following computation.

Computation 11.33 [Naive recursion of gap-weighted subsequences kernels] The naive recursion for the gap-weighted subsequences kernel is given as follows

$$\begin{aligned}
\kappa_p^S(sa,tb) &= \sum_{(\mathbf{i},\mathbf{j})\in I_p^{|s|+1}\times I_p^{|t|+1}:sa(\mathbf{i})=tb(\mathbf{j})}\lambda^{l(\mathbf{i})+l(\mathbf{j})} \\
&= [a=b]\sum_{i=1}^{|s|}\sum_{j=1}^{|t|}\lambda^{2+|s|-i+|t|-j}\sum_{(\mathbf{i},\mathbf{j})\in I_{p-1}^i\times I_{p-1}^j:s(\mathbf{i})=t(\mathbf{j})}\lambda^{l(\mathbf{i})+l(\mathbf{j})}
\end{aligned}$$

$$= [a = b] \sum_{i=1}^{|s|} \sum_{j=1}^{|t|} \lambda^{2+|s|-i+|t|-j} \kappa_{p-1}^{S} (s(1:i), t(1:j)). \quad (11.5)$$

■

Example 11.34 Using the strings from Example 11.18 leads to the computations shown in Table 11.4 for the suffix version for $p = 1, 2, 3$. Hence:

DP : κ_1^S	g	a	t	t	a
c	0	0	0	0	0
a	0	λ^2	0	0	λ^2
t	0	0	λ^2	λ^2	0
a	0	λ^2	0	0	λ^2
DP : κ_2^S	g	a	t	t	a
c	0	0	0	0	0
a	0	0	0	0	0
t	0	0	λ^4	λ^5	0
a	0	0	0	0	$\lambda^7 + \lambda^5 + \lambda^4$
DP : κ_3^S	g	a	t	t	a
c	0	0	0	0	0
a	0	0	0	0	0
t	0	0	0	0	0
a	0	0	0	0	$2\lambda^7$

Table 11.4. *Computations for the gap-weighted subsequences kernel.*

the gap-weighted subsequences kernels between $s =$"gatta" and $t =$"cata" for $p = 1, 2, 3$ are

$$\begin{aligned} \kappa_1(\texttt{"gatta"}, \texttt{"cata"}) &= 6\lambda^2, \\ \kappa_2(\texttt{"gatta"}, \texttt{"cata"}) &= \lambda^7 + 2\lambda^5 + 2\lambda^4 \\ \text{and } \kappa_3(\texttt{"gatta"}, \texttt{"cata"}) &= 2\lambda^7. \end{aligned}$$

Cost of the computation If we were to implement a naive evaluation of the gap-weighted suffix subsequences kernel using the recursive definition of Computation 11.33, then for each value of p we must sum over all the entries in the previous table. This leads to a complexity of $O\left(|t|^2 |s|^2\right)$ to complete the table. Since there will be p tables required to evaluate κ_p this gives an overall complexity of $O\left(p |t|^2 |s|^2\right)$.

Remark 11.35 [Possible Computational Strategy] There are two different ways in which this complexity could be reduced. The first is to observe that the only non-zero entries in the tables for the suffix versions are the positions (i, j) for which $s_i = t_j$. Hence, we could keep a list $L(s,t)$ of these pairs

$$L(s,t) = \{(i,j) : s_i = t_j\},$$

and sum over the list $L(s,t)$ rather than over the complete arrays. The entries in the list could be inserted in lexicographic order of the index pairs. When summing over the list to compute an entry in position (k,l) we would need to consider all entries in the list before the (k,l)th entry, but only include those (i,j) with $i < k$ and $j < l$. This approach could also improve the memory requirements of the algorithm while the complexity would reduce to

$$O\left(p\left|L(s,t)\right|^2\right) \leq O\left(p\left|t\right|^2\left|s\right|^2\right).$$

The list version will be competitive when the list is short. This is likely to occur when the size of the alphabet is large, so that the chances of two symbols being equal becomes correspondingly small. ■

We will not develop this approach further but consider a method that reduces the complexity to $O(p\left|t\right|\left|s\right|)$. For small alphabets this will typically be better than using the list approach.

11.5.2 Efficient implementation

Consider the recursive equation for the gap-weighted subsequences kernel given in equation (11.5). We now consider computing an intermediate dynamic programming table DP_p whose entries are

$$\mathrm{DP}_p(k,l) = \sum_{i=1}^{k}\sum_{j=1}^{l} \lambda^{k-i+l-j}\kappa_{p-1}^{S}\left(s(1:i), t(1:j)\right).$$

Given this table we can evaluate the kernel by observing that

$$\kappa_p^S(sa,tb) = \begin{cases} \lambda^2\,\mathrm{DP}_p\left(\left|s\right|,\left|t\right|\right) & \text{if } a = b; \\ 0 & \text{otherwise.} \end{cases}$$

Computation 11.36 [Gap-weighted subsequences kernel] There is a natural recursion for evaluating $\mathrm{DP}_p(k,l)$ in terms of $\mathrm{DP}_p(k-1,l)$, $\mathrm{DP}_p(k,l-1)$

and $\mathrm{DP}_p(k-1, l-1)$

$$\begin{aligned}\mathrm{DP}_p(k,l) &= \sum_{i=1}^{k}\sum_{j=1}^{l}\lambda^{k-i+l-j}\kappa_{p-1}^{S}(s(1:i), t(1:j)) \\ &= \kappa_{p-1}^{S}(s(1:k), t(1:l)) + \lambda\,\mathrm{DP}_p(k,l-1) \\ &\quad + \lambda\,\mathrm{DP}_p(k-1,l) - \lambda^2\,\mathrm{DP}_p(k-1,l-1).\end{aligned} \tag{11.6}$$

■

Correctness of the recursion The correctness of this recursion can be seen by observing that the contributions to the sum can be divided into three groups: when $(i,j) = (k,l)$ we obtain the first term; those with $i = k$, $j < l$ are included in the second term with the correct weighting; those with $j = l$ and $i < k$ are included in the third term; while those with $i < k$, $j < l$, are included in the second, third and fourth terms with opposite weighting in the fourth, leading to their correct inclusion in the overall sum.

Example 11.37 Using the strings from Example 11.18 leads to the DP computations of Table 11.5 for $p = 2, 3$. Note that we do not need to compute the last row and column. The final evaluation of $\kappa_3(\texttt{"gatta"}, \texttt{"cata"})$ is given by the sum of the entries in the κ_3^S table.

The complexity of the computation required to compute the table DP_p for a single value of p is clearly $O(|t|\,|s|)$ as is the complexity of computing κ_p^S from DP_p making the overall complexity of computing the kernel $\kappa_p(s,t)$ equal to $O(p\,|t|\,|s|)$.

Algorithm 11.38 [Gap-weighted subsequences kernel] The gap-weighted subsequences kernel is computed in Code Fragment 11.3. ■

Remark 11.39 [Improving the complexity] Algorithm 11.38 seems to fail to make use of the fact that in many cases most of the entries in the table DPS are zero. This fact forms the basis of the list algorithm with complexity $O\left(p\,|L(s,t)|^2\right)$. It would be very interesting if the two approaches could be combined to create an algorithm with complexity $O(p\,|L(s,t)|)$, though this seems to be a non-trivial task. ■

11.5.3 Variations on the theme

The inspiration for the gap-weighted subsequences kernel is first and foremost from the consideration of DNA sequences where it is likely that inser-

DP : κ_1^S	g	a	t	t	a
c	0	0	0	0	0
a	0	λ^2	0	0	λ^2
t	0	0	λ^2	λ^2	0
a	0	λ^2	0	0	λ^2

DP_2	g	a	t	t
c	0	0	0	0
a	0	λ^2	λ^3	λ^4
t	0	λ^3	$\lambda^4 + \lambda^2$	$\lambda^5 + \lambda^3 + \lambda^2$

DP : κ_2^S	g	a	t	t	a
c	0	0	0	0	0
a	0	0	0	0	0
t	0	0	λ^4	λ^5	0
a	0	0	0	0	$\lambda^7 + \lambda^5 + \lambda^4$

DP_3	g	a	t	t
c	0	0	0	0
a	0	0	0	0
t	0	0	λ^4	$2\lambda^5$

DP : κ_3^S	g	a	t	t	a
c	0	0	0	0	0
a	0	0	0	0	0
t	0	0	0	0	0
a	0	0	0	0	$2\lambda^7$

Table 11.5. *Computations for the dynamic programming tables of the gap-weighted subsequences kernel.*

tions and deletions of base pairs could occur as genes evolve. This suggests that we should be able to detect similarities between strings that have common subsequences with some gaps.

This consideration throws up a number of possible variations and generalisations of the gap-weighted kernel that could be useful for particular applications. We will discuss how to implement some of these variants in order to show the flexibility inherent in the algorithms we have developed.

Character-weightings string kernel Our first variant is to allow different weightings for different characters that are skipped over by a sub-

Input	strings s and t of lengths n and m, length p, parameter λ
Process	$\mathrm{DPS}(1:n, 1:m) = 0;$
2	for $i = 1 : n$
3	for $j = 1 : m$
4	if $s_i = t_j$
5	$\mathrm{DPS}(i,j) = \lambda^2;$
6	end
7	end
8	end
9	$\mathrm{DP}(0, 0:m) = 0;$
10	$\mathrm{DP}(1:n, 0) = 0;$
11	for $l = 2 : p$
12	$\mathrm{Kern}(l) = 0;$
13	for $i = 1 : n-1$
14	for $j = 1 : m-1$
15	$\mathrm{DP}(i,j) = \mathrm{DPS}(i,j) + \lambda\,\mathrm{DP}(i-1,j) +$
16	$\lambda\,\mathrm{DP}(i,j-1) - \lambda^2\,\mathrm{DP}(i-1,j-1);$
17	if $s_i = t_j$
18	$\mathrm{DPS}(i,j) = \lambda^2\,\mathrm{DP}(i-1,j-1);$
19	$\mathrm{Kern}(l) = \mathrm{Kern}(l) + \mathrm{DPS}(i,j);$
20	end
21	end
22	end
23	end
Output	kernel evaluation $\kappa_p(s,t) = \mathrm{Kern}(p)$

Code Fragment 11.3. Pseudocode for the gap-weighted subsequences kernel.

sequence. In the DNA example, this might perhaps be dictated by the expectation that certain base pairs are more likely to become inserted than others.

These gap-weightings will be denoted by λ_a for $a \in \Sigma$. Similarly, we will consider different weightings for characters that actually occur in a subsequence, perhaps attaching greater importance to certain symbols. These weightings will be denoted by μ_a for $a \in \Sigma$. With this generalisation the basic formula governing the recursion of the suffix kernel becomes

$$\kappa_p^S(sa,tb) = [a=b]\,\mu_a^2 \sum_{i=1}^{n}\sum_{j=1}^{m} \lambda\left(s(i+1:n)\right)\lambda\left(t(j+1:m)\right) \kappa_{p-1}^S\left(s(1:i), t(1:j)\right),$$

where $n = |s|$, $m = |t|$ and we have used the notation $\lambda(u)$ for

$$\lambda(u) = \prod_{i=1}^{|u|} \lambda_{u_i}.$$

The corresponding recursion for $\mathrm{DP}_p(k,l)$ becomes

$$\begin{aligned}\mathrm{DP}_p(k,l) &= \kappa_{p-1}^S(s(1:k), t(1:l)) + \lambda_{t_l}\mathrm{DP}_p(k,l-1) \\ &\quad + \lambda_{s_k}\mathrm{DP}_p(k-1,l) - \lambda_{t_l}\lambda_{s_k}\mathrm{DP}_p(k-1,l-1),\end{aligned}$$

with $\kappa_p^S(sa,tb)$ now given by

$$\kappa_p^S(sa,tb) = \begin{cases} \mu_a^2 \mathrm{DP}_p(|s|,|t|) & \text{if } a = b; \\ 0 & \text{otherwise.} \end{cases}$$

Algorithm 11.40 [Character weightings string kernel] An implementation of this variant can easily be effected by corresponding changes to lines 5, 15 and 17 of Algorithm 11.38

```
5   | DPS(i,j) = μ(s_i)^2;
15a | DP(i,j) = DPS(i,j) + λ(s_i) DP(i-1,j) +
15b |                 λ(t_j)(DP(i,j-1) − λ(s_i) DP(i-1,j-1));
17  | DPS(i,j) = μ(s_i)^2 DP(i-1,j-1);
```

■

Soft matching So far, distinct symbols have been considered entirely unrelated. This may not be an accurate reflection of the application under consideration. For example, in DNA sequences it is more likely that certain base pairs will arise as a mutation of a given base pair, suggesting that some non-zero measure of similarity should be set between them.

We will assume that a similarity matrix $\mathbf{A}$ between symbols has been given. We may expect that many of the entries in $\mathbf{A}$ are zero, that the diagonal is equal to 1, but that some off-diagonal entries are non-zero. We must also assume that $\mathbf{A}$ is positive semi-definite in order to ensure that a valid kernel is defined, since $\mathbf{A}$ is the kernel matrix of the set of all single character strings when $p = 1$.

With this generalisation the basic formula governing the recursion of the suffix kernel becomes

$$\kappa_p^S(sa,tb) = \lambda^2 \mathbf{A}_{ab} \sum_{i=1}^{|s|} \sum_{j=1}^{|t|} \lambda^{|s|-i+|t|-j} \kappa_{p-1}^S(s(1:i), t(1:j)).$$

The corresponding recursion for $\mathrm{DP}_p(k,l)$ remains unchanged as in (11.6), while $\kappa_p^S(sa,tb)$ is now given by

$$\kappa_p^S(sa,tb) = \lambda^2 \mathbf{A}_{ab} \underset{p}{\mathrm{DP}}(|s|,|t|).$$

Algorithm 11.41 [Soft matching string kernel] An implementation of soft matching requires the alteration of lines 4, 5, 16 and 17 of Algorithm 11.38

```
4  | if A(s(i), t(j)) ≠ 0
5  |     DPS(i, j) = λ² A(s_i, t_j);
16 |     if A(s_i, t_j) ≠ 0
17 |      DPS(i, j) = λ² A(s_i, t_j) DP(i − 1, j − 1);
```

■

Weighting by number of gaps It may be that once an insertion occurs its length is not important. In other words we can expect bursts of inserted characters and wish only to penalise the similarity of two subsequences by the number and not the length of the bursts. For this variant the recursion becomes

$$\kappa_p^S(sa,tb) = [a=b] \sum_{i=1}^{|s|} \sum_{j=1}^{|t|} \lambda^{[i\neq|s|]} \lambda^{[j\neq|t|]} \kappa_{p-1}^S(s(1:i), t(1:j)),$$

since we only apply the penalty λ if the previous character of the subsequence occurs before position $|s|$ in s and before position $|t|$ in t.

In this case we must create the dynamic programming table DP_p whose entries are

$$\mathrm{DP}_p(k,l) = \sum_{i=1}^{k} \sum_{j=1}^{l} \kappa_{p-1}^S(s(1:i), t(1:j)),$$

using the recursion

$$\begin{aligned} \mathrm{DP}_p(k,l) &= \kappa_{p-1}^S(s(1:k), t(1:l)) + \mathrm{DP}_p(k,l-1) \\ &\quad + \mathrm{DP}_p(k-1,l) - \mathrm{DP}_p(k-1,l-1), \end{aligned}$$

corresponding to $\lambda = 1$. Given this table we can evaluate the kernel by observing that if $a = b$ then

$$\begin{aligned} \kappa_p^S(sa,tb) &= \mathrm{DP}_p(|s|,|t|) + (\lambda-1)(\mathrm{DP}_p(|s|,|t|-1) + \mathrm{DP}_p(|s|-1,|t|)) \\ &\quad + \left(\lambda^2 - 2\lambda + 1\right) \mathrm{DP}_p(|s|-1,|t|-1), \end{aligned}$$

since the first term ensures the correct weighting of $\kappa_{p-1}^{S}(s,t)$, while the second corrects the weighting of those entries involving a single λ factor and the third term adjusts the weighting of the remaining contributions.

Algorithm 11.42 [Gap number weighting string kernel] An implementation of weighting by number of gaps requires setting $\lambda = 1$ in lines 5 and 15 and altering line 17 to

$$\begin{aligned} 17a \qquad & \mathrm{DPS}(i,j) = \mathrm{DP}(i-1,j-1) + (\lambda-1)\,(\mathrm{DP}(i-2,j-1) \\ 17b \qquad & \qquad + \mathrm{DP}(i-1,j-2) + (\lambda-1)\,\mathrm{DP}(i-2,j-2))\,; \end{aligned}$$

■

Remark 11.43 [General symbol strings] Though we have introduced the kernels in this chapter with character symbols in mind, they apply to any alphabet of symbols. For example if we consider the alphabet to be the set of reals we could compare numeric sequences. Clearly we will need to use the soft matching approach described above. The gap-weighted kernel is only appropriate if the matrix $\mathbf{A}$ comparing individual numbers has many entries equal to zero. An example of such a kernel is given in (9.11). ■

11.6 Beyond dynamic programming: trie-based kernels

A very efficient, although less general, class of methods for implementing string kernels can be obtained by following a different computational approach. Instead of using dynamic programming as the core engine of the computation, one can exploit an efficient data structure known as a 'trie' in the string matching literature. The name trie is derived from 'retrieval tree'. Here we give the definition of a trie – see Definition 11.56 for a formal definition of trees.

Definition 11.44 [Trie] A *trie* over an alphabet Σ is a tree whose internal nodes have their children indexed by Σ. The edges connecting a parent to its child are labelled with the corresponding symbol from Σ. A *complete trie of depth p* is a trie containing the maximal number of nodes consistent with the depth of the tree being p. A *labelled trie* is a trie for which each node has an associated label. ■

In a complete trie there is a 1–1 correspondence between the nodes at depth k and the strings of length k, the correspondence being between the node and the string on the path to that node from the root. The string associated with the root node is the empty string ε. Hence, we will refer to

the nodes of a trie by their associated string. The key observation behind the trie-based approach is that one can therefore regard the *leaves* of the complete trie of depth p as the indices of the feature space indexed by the set Σ^p of strings of length p.

The algorithms extract relevant substrings from the source string being analysed and attach them to the root. The substrings are then repeatedly moved into the subtree or subtrees corresponding to their next symbol until the leaves corresponding to that substring are reached.

For a source string s there are $\frac{1}{2}|s|(|s|+1)$ non-trivial substrings $s(i:j)$ being determined by an initial index in $i \in \{1,\ldots,|s|\}$ and a final index $j \in \{i,\ldots,|s|\}$. Typically the algorithms will only consider a restricted subset of these possible substrings. Each substring can potentially end up at a number of leaves through being processed into more than one subtree. Once the substrings have been processed for each of the two source strings, their inner product can be computed by traversing the leaves, multiplying the corresponding weighted values at each leaf and summing.

Computation 11.45 [Trie-based String Kernels] Hence we can summarise the algorithm into its four main phases:

phase 1: Form all substrings $s(i:j)$ satisfying initial criteria;
phase 2: work the substrings of string s down from root to leaves;
phase 3: work the substrings of string t down from root to leaves;
phase 4: compute products at leaves and sum over the tree.

■

This breakdown is just conceptual, but it does suggest an added advantage of the approach when we are computing a kernel matrix for a set

$$S = \left\{s^1,\ldots,s^\ell\right\}$$

of strings. We need to perform phase 2 only once for each row of the kernel, subsequently repeating phases 3 and 4 as we cycle through the set S with the results of phase 2 for the string indexing the row retained at the leaves throughout.

Despite these advantages it is immediately clear that the approach does have its restrictions. Clearly, if all the leaves are populated then the complexity will be at least $|\Sigma|^p$, which for large values of p will become inefficient. Similarly, we must restrict the number of strings that can arrive at each leaf. If we were to consider the general gap-weighted kernel this could be of the order of the number of substrings of the source string s further adding to

the complexity. For the examples we consider below we are able to place a bound on the number of populated leaves.

11.6.1 Trie computation of the p-spectrum kernels

As a simple first example let us consider this approach being used to compute the p-spectrum kernel. In this case the strings that are filtered down the trie are the substrings of length p of the source string s. The algorithm creates a list $L_s(\varepsilon)$ attached to the root node ε containing these $|s| - p + 1$ strings u, each with an associated index i initialised to 0. A similar list $L_t(\varepsilon)$ is created for the second string t. We now process the nodes recursively beginning with the call 'processnode(ε)' using the following strategy after initialising the global variable 'Kern' to 0. The complete algorithm is given here.

Algorithm 11.46 [Trie-based p-spectrum] The trie-based computation of the p-spectrum is given in Code Fragment 11.4. ■

Input	strings s and t, parameter p
Process	Let $L_s(\varepsilon) = \{(s(i : i+p-1), 0) : i = 1 : \lvert s\rvert - p + 1\}$
2	Let $L_t(\varepsilon) = \{(t(i : i+p-1), 0) : i = 1 : \lvert t\rvert - p + 1\}$
3	Kern $= 0$;
4	processnode$(\varepsilon, 0)$;
where	processnode(v, depth)
6	let $L_s(v)$, $L_t(v)$ be the lists associated with v;
7	if depth $= p$
8	Kern $=$ Kern $+ \lvert L_s(v)\rvert \lvert L_t(v)\rvert$;
9	end
10	else if $L_s(v)$ and $L_t(v)$ both not empty
11	while there exists (u, i) in the list $L_s(v)$
12	add $(u, i+1)$ to the list $L_s(vu_{i+1})$;
13	end
14	while there exists (u, i) in the list $L_t(v)$
15	add $(u, i+1)$ to the list $L_t(vu_{i+1})$;
16	end
17	for $a \in \Sigma$
18	processnode$(va, \text{depth}+1)$;
19	end
20	end
Output	$\kappa_p(s, t) =$ Kern

Code Fragment 11.4. Pseudocode for trie-based implementation of spectrum kernel.

Correctness of the algorithm The correctness of the algorithm follows from the observation that for all the pairs (u, i) in the list $L_s(v)$ we have $v = u(1:i)$, similarly for $L_t(v)$. This is certainly true at the start of the algorithm and continues to hold since $(u, i+1)$ is added to the list at the vertex $vu_{i+1} = u(1:i+1)$. Hence, the substrings reaching a leaf vertex v are precisely those substrings equal to v. The length of the list $L_s(v)$ is therefore equal to

$$\phi_v^p(s) = |\{(v_1, v_2) : s = v_1 v v_2\}|,$$

implying the algorithm correctly computes the p-spectrum kernel.

Cost of the computation The complexity of the computation can be divided into the preprocessing which involves $O(|s| + |t|)$ steps, followed by the processing of the lists. Each of the $|s| - p + 1 + |t| - p + 1$ substrings processed gets passed down at most p times, so the complexity of the main processing is $O(p(|s| - p + 1 + |t| - p + 1))$ giving an overall complexity of

$$O(p(|s| + |t|)).$$

At first sight there appears to be a contradiction between this complexity and the size of the feature space $|\Sigma|^p$, which is exponential in p. The reason for the difference is that even for moderate p there are far fewer substrings than leaves in the complete trie of depth p. Hence, the recursive algorithm will rapidly find subtrees that are not populated, i.e. nodes u for which one of the lists $L_s(u)$ or $L_t(u)$ is empty. The algorithm effectively prunes the subtree below these nodes since the recursion halts at this stage. Hence, although the complete tree is exponential in size, the tree actually processed $O(p(|s| + |t|))$ nodes.

Remark 11.47 [Computing the kernel matrix] As already discussed above we can use the trie-based approach to complete a whole row of a kernel matrix more efficiently than if we were to evaluate each entry independently. We first process the string s indexing that row, hence populating the leaves of the trie with the lists arising from the string s. The cost of this processing is $O(p|s|)$. For each string t^i, $i = 1, \ldots, \ell$, we now process it into the trie and evaluate its kernel with s. For the ith string this takes time $O\left(p\left|t^i\right|\right)$. Hence, the overall complexity is

$$O\left(p\left(|s| + \sum_{i=1}^{\ell} \left|t^i\right|\right)\right),$$

rather than

$$O\left(p\left(\ell\,|s| + \sum_{i=1}^{\ell} |t^i|\right)\right),$$

that would be required if the information about s is not retained. This leads to the overall complexity for computing the kernel matrix

$$O\left(p\ell \sum_{i=1}^{\ell} |t^i|\right).$$

Despite the clear advantages in terms of computational complexity, there is one slight drawback of this method discussed in the next remark. ■

Remark 11.48 [Memory requirements] If we are not following the idea of evaluating a row of the kernel matrix, we can create and dismantle the tree as we process the strings. As we return from a recursive call all of the information in that subtree is no longer required and so it can be deleted. This means that at every stage of the computation there are at most $p\,|\Sigma|$ nodes held in memory with the substrings distributed among their lists. The factor of $|\Sigma|$ in the number of nodes arises from the fact that as we process a node we potentially create all of its $|\Sigma|$ children before continuing the recursion into one of them. ■

Remark 11.49 [By-product information] Notice that the trie also contains information about the spectra of order less than p so one could use it to calculate a blended spectrum kernel of the type

$$\kappa_{p,\mathbf{a}}(s,t) = \sum_{i=1}^{p} a_i \kappa_i(s,t),$$

by simply making the variable 'Kern' an array indexed by depth and updating the appropriate entry as each node is processed. ■

11.6.2 Trie-based mismatch kernels

We are now in a position to consider some extensions of the trie-based technique developed above for the p-spectrum kernel. In this example we will stick with substrings, but consider allowing some errors in the substring. For two strings u and v of the same length, we use $d\,(u,v)$ to denote the number of characters in which u and v differ.

Definition 11.50 [Mismatch kernel] The mismatch kernel $\kappa_{p,m}$ is defined by the feature mapping

$$\phi_u^{p,m}(s) = |\{(v_1, v_2) : s = v_1 v v_2 : |u| = |v| = p, d(u, v) \leq m\}|,$$

that is the feature associated with string u counts the number of substrings of s that differ from u by at most m symbols. The associated kernel is defined as

$$\kappa_{p,m}(s, t) = \langle \phi^{p,m}(s), \phi^{p,m}(t) \rangle = \sum_{u \in \Sigma^p} \phi_u^{p,m}(s) \phi_u^{p,m}(t).$$

■

In order to apply the trie-based approach we initialise the lists at the root of the trie in exactly the same way as for the p-spectrum kernel, except that each substring u has two numbers attached (u, i, j), the current index i as before and the number j of mismatches allowed so far. The key difference in the processing is that when we process a substring it can be added to lists associated with more than one child node, though in all but one case the number of mismatches will be incremented. We give the complete algorithm before discussing its complexity.

Algorithm 11.51 [Trie-based mismatch kernel] The trie-based computation of the mismatch kernel is given in Code Fragment 11.5. ■

Cost of the computation The complexity of the algorithm has been somewhat compounded by the mismatches. Each substring at a node potentially gives rise to $|\Sigma|$ substrings in the lists associated with its children. If we consider a single substring u at the root node it will reach all the leaves that are at a distance at most m from u. If we consider those at a distance k for some $0 \leq k \leq m$ there are

$$\binom{p}{k} |\Sigma|^k$$

such strings. So we can bound the number of times they are processed by

$$O\left(p^{k+1} |\Sigma|^k\right),$$

hence the complexity of the overall computation is bounded by

$$O\left(p^{m+1} |\Sigma|^m (|s| + |t|)\right),$$

taking into account the number of substrings at the root node. Clearly, we must restrict the number of mismatches if we wish to control the complexity of the algorithm.

```
Input   | strings s and t, parameters p and m
--------+------------------------------------------------------------
Process | Let L_s(ε) = {(s(i : i+p−1), 0, 0) : i = 1 : |s| − p + 1}
2       | Let L_t(ε) = {(t(i : i+p−1), 0, 0) : i = 1 : |t| − p + 1}
3       | Kern = 0;
4       | processnode(ε, 0);
--------+------------------------------------------------------------
where   | processnode(v, depth)
6       |  let L_s(v), L_t(v) be the lists associated with v;
7       |  if depth = p
8       |     Kern = Kern + |L_s(v)| |L_t(v)|;
9       |  end
10      |  else if L_s(v) and L_t(v) both not empty
11      |    while there exists (u, i, j) in the list L_s(v)
12      |       add (u, i+1, j) to the list L_s(vu_{i+1});
13      |       if j < m
14      |          for a ∈ Σ, a ≠ u_{i+1}
15      |             add (u, i+1, j+1) to the list L_s(va);
16      |    end
17      |     while there exists (u, i, j) in the list L_t(v)
18      |       add (u, i+1, j) to the list L_t(vu_{i+1});
19      |       if j < m
20      |          for a ∈ Σ, a ≠ u_{i+1}
21      |             add (u, i+1, j+1) to the list L_t(va);
22      |    end
23      |    for a ∈ Σ
24      |      processnode(va, depth+1);
25      |    end
26      |  end
--------+------------------------------------------------------------
Output  | κ_{p,m}(s, t) = Kern
```

Code Fragment 11.5. Pseudocode for the trie-based implementation of the mismatch kernel.

Remark 11.52 [Weighting mismatches] As discussed in the previous section when we considered varying the substitution costs for different pairs of symbols, it is possible that some mismatches may be more costly than others. We can assume a matrix of mismatch costs $\mathbf{A}$ whose entries $\mathbf{A}_{ab}$ give the cost of symbol b substituting symbol a. We could now define a feature mapping for a substring u to count the number of substrings whose total mismatch cost is less than a threshold σ. We can evaluate this kernel with a few adaptations to Algorithm 11.51. Rather than using the third component of the triples (u, i, j) to store the number of mismatches, we store the total cost of mismatches included so far. We now replace lines 13–16 of the

algorithm with

```
13 |    for a ∈ Σ, a ≠ u_{i+1}
14 |        if j + A(a, u_{i+1}) < σ
15 |            add (u, i + 1, j + C(a, u_{i+1})) to the list L_s(va);
16 |    end
```

with similar changes made to lines 19–22. ■

11.6.3 Trie-based restricted gap-weighted kernels

Our second extension considers the gap-weighted features of the subsequences kernels. As indicated in our general discussion of the trie-based approach, it will be necessary to restrict the sets of subsequences in some way. Since they are typically weighted by an exponentially-decaying function of their length it is natural to restrict the subsequences by the lengths of their occurrences. This leads to the following definition.

Definition 11.53 [Restricted gap-weighted subsequences kernel] The feature space associated with the *m-restricted gap-weighted subsequences kernel* $\kappa_{p,m}$ *of length* p is indexed by $I = \Sigma^p$, with the embedding given by

$$\phi_u^{p,m}(s) = \sum_{\mathbf{i}:u=s(\mathbf{i})} [l(\mathbf{i}) \leq m] \lambda^{l(\mathbf{i})}, \ u \in \Sigma^p.$$

The associated kernel is defined as

$$\kappa_{p,m}(s,t) = \langle \boldsymbol{\phi}^{p,m}(s), \boldsymbol{\phi}^{p,m}(t) \rangle = \sum_{u \in \Sigma^p} \phi_u^{p,m}(s) \phi_u^{p,m}(t).$$

■

In order to apply the trie-based approach we again initialise the lists at the root of the trie in a similar way to the previous kernels, except that each substring should now have length $p + m$ since we must allow for as many as m gaps. Again each substring u has two numbers attached (u, i, j), the current index i as before and the number j of gaps allowed so far. We must restrict the first character of the subsequence to occur at the beginning of u as otherwise we would count subsequences with fewer than m gaps more than once. We avoid this danger by inserting the strings directly into the lists associated with the children of the root node.

At an internal node the substring can be added to the same list more than once with different numbers of gaps. When we process a substring (u, i, j) we consider adding every allowable number of extra gaps (the variable k in

the next algorithm is used to store this number) while still ensuring that the overall number is not more than m. Hence, the substring can be inserted into as many as m of the lists associated with a node's children.

There is an additional complication that arises when we come to compute the contribution to the inner product at the leaf nodes, since not all of the substrings reaching a leaf should receive the same weighting. However, the number of gaps is recorded and so we can evaluate the correct weighting from this information. Summing for each list and multiplying the values obtained gives the overall contribution. For simplicity we will simply denote this computation by $\kappa\left(L_s\left(v\right), L_t\left(v\right)\right)$ in the algorithm given below.

Algorithm 11.54 [Trie-based restricted gap-weighted subsequences kernel] The trie-based computation of the restricted gap-weighted subsequences kernel is given in Code Fragment 11.6. ■

Cost of the computation Again the complexity of the algorithm is considerably expanded since each of the original $|s| - p - m + 1$ substrings will give rise to

$$\binom{p+m-1}{m-1}$$

different entries at leaf nodes. Hence, the complexity of the overall algorithm can be bounded this number of substrings times the cost of computation on the path from root to leaf, which is at most $O\left(p+m\right)$ for each substring, giving an overall complexity of

$$O\left(\left(|s|+|t|\right)\left(p+m\right)^m\right).$$

In this case it is the number of gaps we allow that has a critical impact on the complexity. If this number is made too large then the dynamic programming algorithm will become more efficient. For small values of the decay parameter λ it is, however, likely that we can obtain a good approximation to the full gap-weighted subsequences kernel with modest values of m resulting in the trie-based approach being more efficient.

Remark 11.55 [Linear time evaluation] For all the trie-based kernels, it is worth noting that if we do not normalise the data it is possible to evaluate a linear function

$$f\left(s\right) = \sum_{i \in \mathrm{sv}} \alpha_i \kappa\left(s_i, s\right)$$

```
Input    strings s and t, parameters p and m

Process  for i = 1 : |s| − p − m + 1
2          add (s(i : i + p + m − 1), 1, 0) to the list L_s(s_i)
3        end
4        for i = 1 : |t| − p − m + 1
5          add (t(i : i + p + m − 1), 1, 0) to the list L_t(t_i)
6        end
7        Kern = 0;
8        for a ∈ Σ
9         processnode(a, 0);
10       end

where    processnode(v, depth)
12        let L_s(v), L_t(v) be the lists associated with v;
13        if depth = p
14          Kern = Kern + κ(L_s(v), L_t(v));
15        end
16        else if L_s(v) and L_t(v) both not empty
17          while there exists (u, i, j) in the list L_s(v)
18            for k = 0 : m − j
19              add (u, i + k + 1, k + j) to the list L_s(vu_{i+k+1});
20          end
21          while there exists (u, i, j) in the list L_t(v)
22            for k = 0 : m − j
23              add (u, i + k + 1, k + j) to the list L_t(vu_{i+k+1});
24          end
25          for a ∈ Σ
26            processnode(va, depth + 1);
27          end
28        end

Output   κ_p(s, t) = Kern
```

Code Fragment 11.6. Pseudocode for trie-based restricted gap-weighted subsequences kernel.

in linear time. This is achieved by creating a tree by processing all of the support vectors in sv weighting the substrings from s_i at the corresponding leaves by α_i. The test string s is now processed through the tree and the appropriately weighted contributions to the overall sum computed directedly at each leaf. ■

11.7 Kernels for structured data

In this chapter we have shown an increasingly sophisticated series of kernels that perform efficient comparisons between strings of symbols of different lengths, using as features:

- all contiguous and non-contiguous subsequences of any size;
- all subsequences of a fixed size; all contiguous substrings of a fixed size or up to a fixed size, and finally;
- all non-contiguous substrings with a penalisation related to the size of the gaps.

The evaluation of the resulting kernels can in all cases be reduced to a dynamic programming calculation of low complexity in the length of the sequences and the length of the subsequences used for the matching. In some cases we have also shown how the evaluation can be sped up by the use of tries to achieve a complexity that is linear in the sum of the lengths of the two input strings.

Our aim in this section is to show that, at least for the case of dynamic programming, the approaches we have developed are not restricted to strings, but can be extended to a more general class we will refer to as 'structured data'.

By structured data we mean data that is formed by combining simpler components into more complex items frequently involving a recursive use of simpler objects of the same type. Typically it will be easier to compare the simpler components either with base kernels or using an inductive argument over the structure of the objects. Examples of structured data include the familiar examples of vectors, strings and sequences, but also subsume more complex objects such as trees, images and graphs.

From a practical viewpoint, it is difficult to overemphasise the importance of being able to deal in a principled way with data of this type as they almost invariably arise in more complex applications. Dealing with this type of data has been the objective of a field known as structural pattern recognition. The design of kernels for data of this type enables the same algorithms and analyses to be applied to entirely new application domains.

While in Part II of this book we presented algorithms for detecting and analysing several different types of patterns, the extension of kernels to structured data paves the way for analysing very diverse sets of data-types. Taken together, the two advances open up possibilities such as discovering clusters in a set of trees, learning classifications of graphs, and so on.

It is therefore by designing specific kernels for structured data that kernel methods can demonstrate their full flexibility. We have already discussed

one kernel of this type when we pointed out that both the ANOVA and string kernels can be viewed as combining kernel evaluations over substructures of the objects being compared.

In this final section, we will provide a more general framework for designing kernels for this type of data, and we will discuss the connection between statistical and structural pattern analysis that this approach establishes. Before introducing this framework, we will discuss one more example with important practical implications that will not only enable us to illustrate many of the necessary concepts, but will also serve as a building block for a method to be discussed in Chapter 12.

11.7.1 Comparing trees

Data items in the form of trees can be obtained as the result of biological investigation, parsing of natural language text or computer programs, XML documents and in some representations of images in machine vision. Being able to detect patterns within sets of trees can therefore be of great practical importance, especially in web and biological applications. In this section we derive two kernel functions that can be used for this task, as well as provide a conceptual stepping stone towards certain kernels that will be defined in the next chapter.

We will design a kernel between trees that follows a similar approach to those we have considered between strings, in that it will also exploit their recursive structure via dynamic programming.

Recall that for strings the feature space was indexed by substrings. The features used to describe trees will be 'subtrees' of the given tree. We begin by defining what we mean by a tree and subtree. We use the convention that the edges of a tree are directed away from the root towards the leaves.

Definition 11.56 [Trees] A *tree* T is a directed connected acyclic graph in which each vertex (usually referred to as *nodes* for trees) except one has in-degree one. The node with in-degree 0 is known as the *root* $r(T)$ of the tree. Nodes v with out-degree $d^+(v) = 0$ are known as *leaf* nodes, while those with non-zero out-degree are *internal nodes*. The nodes to which an internal node v is connected are known as its *children*, while v is their parent. Two children of the same parent are said to be *siblings*. A *structured tree* is one in which the children of a node are given a fixed ordering, $\mathrm{ch}_1(v), \ldots, \mathrm{ch}_{d^+(v)}(v)$. Two trees are *identical* if there is a 1–1 correspondence between their nodes that respects the parent–child relation and the ordering of the children for each internal node. A *proper tree* is one that

contains at least one edge. The size $|T|$ of a tree T is the number of its nodes. ■

Definition 11.57 [k-ary labelled trees] If the out-degrees of all the nodes are bounded by k we say the tree is *k-ary*; for example when $k = 2$, the tree is known as a *binary tree*. A *labelled* tree is one in which each node has an associated label. Two labelled trees are *identical* if they are identical as trees and corresponding nodes have the same labels. We use $\mathcal{T}$ to denote the set of all proper trees, with $\mathcal{T}_A$ denoting labelled proper trees with labels from the set A. ■

Definition 11.58 [Subtrees: general and co-rooted] A *complete subtree* $\tau(v)$ of a tree T at a node v is the tree obtained by taking the node v together with all vertices and edges reachable from v. A *co-rooted subtree* of a tree T is the tree resulting after removing a sequence of complete subtrees and replacing their roots. This is equivalent to deleting the complete subtrees of all the children of the selected nodes. Hence, if a node v is included in a co-rooted subtree then so are all of its siblings. The root of a tree T is included in every co-rooted subtree. A general subtree of a tree T is any co-rooted subtree of a complete subtree. ■

Remark 11.59 [Strings and trees] We can view strings as labelled trees in which the set of labels is the alphabet Σ and each node has at most one child. A complete subtree of a string tree corresponds to a suffix of the string, a rooted subtree to a prefix and a general subtree to a substring. ■

For our purposes we will mostly be concerned with structured trees which are labelled with information that determines the number of children. We now consider the feature spaces that will be used to represent trees. Again following the analogy with strings the index set of the feature space will be the set of all trees, either labelled or unlabelled according to the context.

Embedding map All the kernels presented in this section can be defined by an explicit embedding map from the space of all finite trees possibly labelled with a set A to a vector space F, whose coordinates are indexed by a subset I of trees again either labelled or unlabelled. As usual we use $\boldsymbol{\phi}$ to denote the feature mapping

$$\boldsymbol{\phi}: T \longmapsto (\boldsymbol{\phi}_S(T))_{S \in I} \in F$$

The aim is to devise feature mappings for which the corresponding kernel

can be evaluated using a recursive calculation that proceeds bottom-up from the leaves to the root of the trees. The basic recursive relation connects the value of certain functions at a given node with the function values at its children. The base of the recursion will set the values at the leaves, ensuring that the computation is well-defined.

Remark 11.60 [Counting subtrees] As an example of a recursive computation over a tree consider evaluating the number $N(T)$ of proper co-rooted subtrees of a tree T. Clearly for a leaf node v there are no proper co-rooted subtrees of $\tau(v)$, so we have $N(\tau(v)) = 0$. Suppose that we know the value of $N(\tau(v_i))$ for each of the nodes $v_i = \mathrm{ch}_i(r(T))$, $i = 1, \ldots, d^+(r(T))$ that are children of the root $r(T)$ of T. In order to create a proper co-rooted subtree of T we must include all of the nodes $r(T), v_1, \ldots, v_{d^+(r(T))}$, but we have the option of including any of the co-rooted subtrees of $\tau(v_i)$ or simply leaving the node v_i as a leaf. Hence, for node v_i we have $N(\tau(v_i)) + 1$ options. These observations lead to the following recursion for $N(T)$ for a proper tree T

$$N(T) = \prod_{i=1}^{d^+(r(T))} \left(N(\tau(\mathrm{ch}_i(r(T)))) + 1\right),$$

with $N(\tau(v)) = 0$, for a leaf node v. ■

We are now in a position to consider two kernels over trees.

Co-rooted subtree kernel The feature space for this kernel will be indexed by all trees with the following feature mapping.

Definition 11.61 [Co-rooted subtree kernel] The feature space associated with the *co-rooted subtree kernel* is indexed by $I = \mathcal{T}$ the set of all proper trees with the embedding given by

$$\phi_S^r(T) = \begin{cases} 1 & \text{if } S \text{ is a co-rooted subtree of } T; \\ 0 & \text{otherwise.} \end{cases}$$

The associated kernel is defined as

$$\kappa_r(T_1, T_2) = \langle \boldsymbol{\phi}^r(T_1), \boldsymbol{\phi}^r(T_2) \rangle = \sum_{S \in \mathcal{T}} \phi_S^r(T_1)\, \phi_S^r(T_2).$$

■

If either tree T_1 or T_2 is a single node then clearly

$$\kappa_r(T_1, T_2) = 0,$$

since for an improper tree T

$$\boldsymbol{\phi}^r(T) = \mathbf{0}.$$

Furthermore if $d^+(r(T_1)) \neq d^+(r(T_1))$ then $\kappa_r(T_1, T_2) = 0$ since a co-rooted subtree of T_1 cannot be a co-rooted subtree of T_2. Assume therefore that

$$d^+(r(T_1)) = d^+(r(T_2)).$$

Now to introduce the recursive computation, assume we have evaluated the kernel between the complete subtrees on the corresponding children of $r(T_1)$ and $r(T_2)$; that is we have computed

$$\kappa_r(\tau(\mathrm{ch}_i(r(T_1))), \tau(\mathrm{ch}_i(r(T_2)))), \text{ for } i = 1, \ldots, d^+(r(T_1)).$$

We now have that

$$\begin{aligned}\kappa_r(T_1, T_2) &= \sum_{S \in \mathcal{T}} \boldsymbol{\phi}_S^r(T_1)\, \boldsymbol{\phi}_S^r(T_2) \\ &= \prod_{i=1}^{d^+(r(T_1))} \sum_{S_i \in \mathcal{T}_0} \boldsymbol{\phi}_{S_i}^r(\tau(\mathrm{ch}_i(r(T_1))))\, \boldsymbol{\phi}_{S_i}^r(\tau(\mathrm{ch}_i(r(T_2)))),\end{aligned}$$

where $\mathcal{T}_0$ denotes the set of all trees, both proper and improper, since the co-rooted proper subtrees of T are determined by any combination of co-rooted subtrees of $\tau(\mathrm{ch}_i(r(T)))$, $i = 1, \ldots, d^+(r(T_1))$.

Since there is only one improper co-rooted subtree we obtain the recursion

$$\kappa_r(T_1, T_2) = \prod_{i=1}^{d^+(r(T_1))} (\kappa_r(\tau(\mathrm{ch}_i(r(T_1))), \tau(\mathrm{ch}_i(r(T_2)))) + 1).$$

We have the following algorithm.

Algorithm 11.62 [Co-rooted subtree kernel] The computation of the co-rooted subtree kernel is given in Code Fragment 11.7. ■

Cost of the computation The complexity of the kernel computation is at most $O(\min(|T_1|, |T_2|))$ since the recursion can only process each node of the trees at most once. If the degrees of the nodes do not agree then the attached subtrees will not be visited and the computation will be correspondingly sped up.

Input	trees T_1 and T_2
Process	Kern =processnode$(r(T_1), r(T_2))$;
where	processnode(v_1, v_2)
3	if $d^+(v_1) \neq d^+(v_2)$ or $d^+(v_1) = 0$
4	return 0;
5	end
6	else
7	Kern = 1;
8	for $i = 1 : d^+(v_1)$
9	Kern = Kern $*$ (processnode $(\mathrm{ch}_i(v_1), \mathrm{ch}_i(v_2)) + 1)$;
10	end
11	return Kern;
12	end
Output	$\kappa_r(T_1, T_2) =$ Kern

Code Fragment 11.7. Pseudocode for the co-rooted subtree kernel.

Remark 11.63 [Labelled trees] If the tree is labelled we must include in line 3 of Algorithm 11.62 the test whether the labels of the nodes v_1 and v_2 match by replacing it by the line

3 | if $d^+(v_1) \neq d^+(v_2)$ or $d^+(v_1) = 0$ or label $(v_1) \neq$ label (v_2)

■

All-subtree kernel We are now in a position to define a slightly more general tree kernel. The features will now be all subtrees rather than just the co-rooted ones. Again we are considering unlabelled trees. The definition is as follows.

Definition 11.64 [All-subtree kernel] The feature space associated with the *all-subtree kernel* is indexed by $I = \mathcal{T}$, the set of all proper trees with the embedding given by

$$\phi_S(T) = \begin{cases} 1 & \text{if } S \text{ is a subtree of } T; \\ 0 & \text{otherwise.} \end{cases}$$

The associated kernel is defined as

$$\kappa(T_1, T_2) = \langle \phi(T_1), \phi(T_2) \rangle = \sum_{S \in \mathcal{T}} \phi_S(T_1)\, \phi_S(T_2).$$

■

The evaluation of this kernel can be reduced to the case of co-rooted subtrees by observing that

$$\kappa(T_1, T_2) = \sum_{v_1 \in T_1, v_2 \in T_2} \kappa_r(\tau(v_1), \tau(v_2)). \tag{11.7}$$

In other words the all-subtree kernel can be computed by evaluating the co-rooted kernel for all pairs of nodes in the two trees. This follows from the fact that any subtree of a tree T is a co-rooted subtree of the complete subtree $\tau(v)$ for some node v of T.

Rather than use this computation we would like to find a direct recursion for $\kappa(T_1, T_2)$. Clearly if T_1 or T_2 is a leaf node we have

$$\kappa(T_1, T_2) = 0.$$

Furthermore, we can partition the sum (11.7) as follows

$$\begin{aligned}
\kappa(T_1, T_2) &= \sum_{v_1 \in T_1, v_2 \in T_2} \kappa_r(\tau(v_1), \tau(v_2)) \\
&= \kappa_r(T_1, T_2) + \sum_{i=1}^{d^+(r(T_1))} \kappa(\tau(\mathrm{ch}_i(r(T_1))), T_2) \\
&\quad + \sum_{i=1}^{d^+(r(T_2))} \kappa(T_1, \tau(\mathrm{ch}_i(r(T_2)))) \\
&\quad - \sum_{i=1}^{d^+(r(T_1))} \sum_{j=1}^{d^+(r(T_2))} \kappa(\tau(\mathrm{ch}_i(r(T_1))), \tau(\mathrm{ch}_j(r(T_2)))),
\end{aligned}$$

since the subtrees are either co-rooted in both T_1 and T_2 or are subtrees of a child of $r(T_1)$ or a child of $r(T_2)$. However, those that are not co-rooted with either T_1 or T_2 will be counted twice making it necessary to subtract the final sum. We therefore have Algorithm 11.65, again based on the construction of a table indexed by the nodes of the two trees using dynamic programming. We will assume that the n_j nodes $v_1^j, \ldots, v_{n_j}^j$ of the tree T_j have been ordered so that the parent of a node has a later index. Hence, the final node $v_{n_j}^j$ is the root of T_j. This ordering will be used to index the tables $\mathrm{DP}(i_1, i_2)$ and $\mathrm{DPr}(i_1, i_2)$. The algorithm first completes the table $\mathrm{DPr}(i_1, i_2)$ with the value of the co-rooted tree kernel before it computes the all-subtree kernel in the array $\mathrm{DP}(i_1, i_2)$. We will also assume for the purposes of the algorithm that $\mathrm{child}_k^j(i)$ gives the index of the kth child of the node indexed by i in the tree T_j. Similarly, $d_j^+(i)$ is the out-degree of the node indexed by i in the tree T_j.

Algorithm 11.65 [All-subtree kernel] The computation of the all-subtree kernel is given in Code Fragment 11.8. ■

Input	unlabelled trees T_1 and T_2
Process	Assume $v_1^j, \ldots, v_{n_j}^j$ a compatible ordering of nodes of T_j, $j = 1, 2$
2	for $i_1 = 1 : n_1$
3	for $i_2 = 1 : n_2$
4	if $d_1^+(i_1) \neq d_2^+(i_2)$ or $d_1^+(i_1) = 0$
5	$\mathrm{DPr}(i_1, i_2) = 0$;
6	else
7	$\mathrm{DPr}(i_1, i_2) = 1$;
8	for $k = 1 : d_1^+(i_1)$
9	$\mathrm{DPr}(i_1, i_2) = \mathrm{DPr}(i_1, i_2) * \left(\mathrm{DPr}\left(\mathrm{child}_k^1(i_1), \mathrm{child}_k^2(i_2)\right) + 1\right)$;
10	end
11	end
12	end
13	for $i_1 = 1 : n_1$
14	for $i_2 = 1 : n_2$
15	if $d_1^+(i_1) = 0$ or $d_2^+(i_2) = 0$
16	$\mathrm{DP}(i_1, i_2) = 0$;
17	else
18	$\mathrm{DP}(i_1, i_2) = \mathrm{DPr}(i_1, i_2)$;
19	for $j_1 = 1 : d_1^+(i_1)$
20	$\mathrm{DP}(i_1, i_2) = \mathrm{DP}(i_1, i_2) + \mathrm{DP}\left(\mathrm{child}_{j_1}^1(i_1), i_2\right)$;
21	for $j_2 = 1 : d_2^+(i_2)$
22	$\mathrm{DP}(i_1, i_2) = \mathrm{DP}(i_1, i_2) + \mathrm{DP}\left(i_1, \mathrm{child}_{j_2}^2(i_2)\right)$;
23	for $j_1 = 1 : d_1^+(i_1)$
24	$\mathrm{DP}(i_1, i_2) = \mathrm{DP}(i_1, i_2) - \mathrm{DP}\left(\mathrm{child}_{j_1}^1(i_1), \mathrm{child}_{j_2}^2(i_2)\right)$;
25	end
26	end
27	end
28	end
Output	$\kappa(T_1, T_2) = \mathrm{DP}(n_1, n_2)$

Code Fragment 11.8. Pseudocode for the all-subtree kernel.

Cost of the computation The structure of the algorithm makes clear that the complexity of evaluating the kernel can be bounded by

$$O\left(|T_1|\,|T_2|\,d_{\max}^2\right),$$

where $d_{\max}$ is the maximal out-degree of the nodes in the two trees.

Remark 11.66 [On labelled trees] Algorithm 11.65 does not take into account possible labellings of the nodes of the tree. As we observed for the co-rooted tree kernel, the computation of the co-rooted table $\mathrm{DPr}(i_1, i_2)$ could be significantly sped up by replacing line 4 with

$$4 \;\Big|\; \text{if } d^+(v_1) \neq d^+(v_2) \text{ or } d^+(v_1) = 0 \text{ or } \mathrm{label}(v_1) \neq \mathrm{label}(v_2).$$

However, no such direct simplification is possible for the computation of $\mathrm{DP}(i_1, i_2)$. If the labelling is such that very few nodes share the same label, it may be faster to create small separate DPr tables for each type and simply sum over all of these tables to compute

$$\kappa(T_1, T_2) = \mathrm{DP}(n_1, n_2),$$

avoiding the need to create the complete table for $\mathrm{DP}(i_1, i_2)$, $1 \leq i_1 < n_1$, $1 \leq i_2 < n_2$. ■

Notice how the all-subtree kernel was defined in terms of the co-rooted subtrees. We first defined the simpler co-rooted kernel and then summed its evaluation over all possible choices of subtrees. This idea forms the basis of a large family of kernels, based on convolutions over substructures. We now turn to examine a more general framework for these kernels.

11.7.2 Structured data: a framework

The approach used in the previous sections for comparing strings, and subsequently trees, can be extended to handle more general types of data structures. The key idea in the examples we have seen has been first to define a way of comparing sub-components of the data such as substrings or subtrees and then to sum these sub-kernels over a set of decompositions of the data items.

We can think of this summing process as a kind of convolution in the sense that it averages over the different choices of sub-component. In general the operation can be thought of as a convolution if we consider the different ways of dividing the data into sub-components as analagous to dividing an interval into two subintervals. We will build this intuitive formulation into the notion of a *convolution kernel*. We begin by formalising what we mean by structured data.

As discussed at the beginning of Section 11.7, a data type is said to be 'structured' if it is possible to decompose it into smaller parts. In some cases there is a natural way in which this decomposition can occur. For example a vector is decomposed into its individual components. However, even here we

can think of other possible decompositions by for example selecting a subset of the components as we did for the ANOVA kernel. A string is structured because it can be decomposed into smaller strings or symbols and a tree can be decomposed into subtrees.

The idea is to compute the product of sub-kernels comparing the parts before summing over the set of allowed decompositions. It is clear that when we create a decomposition we need to specify which kernels are applicable for which sub-parts.

Definition 11.67 [Decomposition Structure] A *decomposition structure* for a datatype $\mathcal{D}$ is specified by a multi-typed relation R between an element x of $\mathcal{D}$ and a finite set of tuples of sub-components each with an associated kernel

$$\left((x_1, \kappa_1), \ldots, (x_d, \kappa_d)\right),$$

for various values of d. Hence

$$\mathcal{R}\left(\left((x_1, \kappa_1), \ldots, (x_d, \kappa_d)\right), x\right)$$

indicates that x can be decomposed into components $x_1, \ldots, x_d$ each with an attached kernel. Note that the kernels may vary between different decompositions of the same x, so that just as x_1 depends on the particular decomposition, so does κ_1. The relation $\mathcal{R}$ is a subset of the disjoint sum of the appropriate cartesian product spaces. The set of all *admissible partitionings* of x is defined as

$$\mathcal{R}^{-1}(x) = \bigcup_{d=1}^{D} \left\{\left((x_1, \kappa_1), \ldots, (x_d, \kappa_d)\right) : \mathcal{R}\left(\left((x_1, \kappa_1), \ldots, (x_d, \kappa_d)\right), x\right)\right\},$$

while the *type* $\mathcal{T}(\mathbf{x})$ of the tuple $\mathbf{x} = \left((x_1, \kappa_1), \ldots, (x_d, \kappa_d)\right)$ is defined as

$$\mathcal{T}(\mathbf{x}) = (\kappa_1, \ldots, \kappa_d).$$

■

Before we discuss how we can view many of the kernels we have met as exploiting decompositions of this type, we give a definition of the convolution or $\mathcal{R}$-kernel that arises from a decomposition structure $\mathcal{R}$.

Definition 11.68 [Convolution Kernels] Let $\mathcal{R}$ be a decomposition structure for a datatype $\mathcal{D}$. For x and z elements of $\mathcal{D}$ the associated convolution

kernel is defined as

$$\kappa_{\mathcal{R}}(x,z) = \sum_{\mathbf{x}\in\mathcal{R}^{-1}(x)} \sum_{\mathbf{z}\in\mathcal{R}^{-1}(z)} \left[\mathcal{T}(\mathbf{x}) = \mathcal{T}(\mathbf{z})\right] \prod_{i=1}^{|\mathcal{T}(\mathbf{x})|} \kappa_i(x_i, z_i).$$

We also refer to this as the $\mathcal{R}$-*convolution kernel* or $\mathcal{R}$-*kernel* for short. Note that the product is only defined if the boolean expression is true, but if this is not the case the square bracket function is zero. ■

Example 11.69 In the case of trees discussed above, we can define the decomposition structures $\mathcal{R}_1$ and $\mathcal{R}_2$ by:

$$\begin{aligned}
&\mathcal{R}_1((S,\kappa_0),T) \text{ if and only if } S \text{ is a co-rooted subtree of } T;\\
&\mathcal{R}_2((S,\kappa_0),T) \text{ if and only if } S \text{ is a subtree of } T;
\end{aligned}$$

where $\kappa_0(S_1,S_2) = 1$ if $S_1 = S_2$ and 0 otherwise. The associated $\mathcal{R}$-kernels are clearly the co-rooted subtree kernel and the all-subtree kernel respectively.

Example 11.70 For the case of the co-rooted tree kernel we could also create the recursive decomposition structure $\mathcal{R}$ by:

$$\begin{aligned}
&\mathcal{R}_1(((T_1,\kappa_{\mathcal{R}}+1),\ldots,(T_d,\kappa_{\mathcal{R}}+1)),T)\\
&\quad \text{if and only if } T_1,\ldots,T_d \text{ are the trees at the children of the root } r(T);\\
&\mathcal{R}_1((T,0),T) \text{ if } T \text{ is an improper tree.}
\end{aligned}$$

By the definition of the associated $\mathcal{R}$-kernel we have

$$\kappa_{\mathcal{R}}(S,T) = \begin{cases} 0 & \text{if } d^+(r(S)) \neq d^+(r(T)) \text{ or } d^+(r(S)) = 0;\\ \prod_{i=1}^{d^+(r(T))} \left(\kappa_{\mathcal{R}}(\tau(\mathrm{ch}_i(r(T))),\tau(\mathrm{ch}_i(r(S)))) + 1\right) & \\ \qquad \text{otherwise.} & \end{cases}$$

This is precisely the recursive definition of the co-rooted subtree kernel.

The two examples considered here give the spirit of the more flexible decomposition afforded by $\mathcal{R}$-decomposition structures. We now give examples that demonstrate how the definition subsumes many of the kernels we have considered in earlier chapters.

Example 11.71 Suppose X is a vector space of dimension n. Let κ_i be the kernel defined by

$$\kappa_i(\mathbf{x},\mathbf{z}) = x_i z_i.$$

Consider the decomposition structure

$$\mathcal{R} = \{(((\mathbf{x}, \kappa_{i_1}), \ldots, (\mathbf{x}, \kappa_{i_d})), \mathbf{x}) : 1 \leq i_1 < i_2 < \cdots < i_d \leq n,\ \mathbf{x} \in \mathbb{R}^n\}.$$

Here the associated $\mathcal{R}$-kernel is defined as

$$\kappa_{\mathcal{R}}(\mathbf{x}, \mathbf{z}) = \sum_{1 \leq i_1 < \cdots < i_d \leq n} \prod_{j=1}^{d} \kappa_{i_j}(\mathbf{x}, \mathbf{z}) = \sum_{1 \leq i_1 < \cdots < i_d \leq n} \prod_{j=1}^{d} x_i z_i,$$

which we have already met in Chapter 9 as the ANOVA kernel. The generalisation of the ANOVA kernel to arbitrary subkernels described in Remark 9.20 seems even more natural in this presentation. Note how the kernel has again the sum of products structure that we have seen recurring in many different guises.

Example 11.72 The more complex kernels defined over graphs can also be recreated as $\mathcal{R}$-kernels if we index the subkernels κ_e by the edges $e \in E$ of the graph $G = (V, E)$. The $\mathcal{R}$-structure for a graph kernel defined by the identified vertices a and b in G is now given by

$$\mathcal{R} = \left\{\left(\left(\left(\mathbf{x}, \kappa_{(u_0 \to u_1)}\right), \ldots, \left(\mathbf{x}, \kappa_{(u_{d-1} \to u_d)}\right)\right), \mathbf{x}\right) : \mathbf{x} \in \mathbb{R}^n, (u_0, \ldots, u_d) \in P_{ab}\right\},$$

where P_{ab} is the set of paths from node a to node b, represented by the sequence of edges. This follows straightforwardly from the definition of the graph kernel as

$$\kappa_G(\mathbf{x}, \mathbf{z}) = \sum_{p \in P_{ab}} \prod_{i=1}^{d} \kappa_{(u_{i-1} \to u_i)}(\mathbf{x}, \mathbf{z}).$$

Notice that the type match in the definition of the $\mathcal{R}$-kernel ensures that only matching paths enter into the sum.

Example 11.73 A very similar structure to the ANOVA case can be used to create the Gaussian kernel. Here we consider the base kernels to be

$$\kappa_i(\mathbf{x}, \mathbf{z}) = \exp\left(-\frac{(x_i - z_i)^2}{2\sigma^2}\right)$$

and the structure to be given by

$$\mathcal{R} = \{(((\mathbf{x}, \kappa_1), \ldots, (\mathbf{x}, \kappa_n)), \mathbf{x}) : \mathbf{x} \in \mathbb{R}^n\},$$

so that the associated $\mathcal{R}$-kernel is simply

$$\begin{aligned}\kappa_{\mathcal{R}}(\mathbf{x},\mathbf{z}) &= \prod_{i=1}^{n}\kappa_i(\mathbf{x},\mathbf{z}) = \prod_{i=1}^{d}\exp\left(-\frac{(x_i-z_i)^2}{2\sigma^2}\right)\\ &= \exp\left(-\frac{\sum_{i=1}^{d}(x_i-z_i)^2}{2\sigma^2}\right) = \exp\left(-\frac{\|\mathbf{x}-\mathbf{z}\|^2}{2\sigma^2}\right).\end{aligned}$$

Example 11.74 It is very simple to construct decomposition structures for the various string kernels. For example the gap-weighted k-subsequences kernel is created using

$$\mathcal{R} = \left\{\left(\left(u,\lambda^{l(\mathbf{i})}\kappa_0\right),s\right) : s\in\Sigma^*,\mathbf{i}\in I_k, u = s(\mathbf{i})\right\},$$

where κ_0 is the kernel returning 1 if the substrings are identical and 0 otherwise.

$\mathcal{R}$-convolution kernels seem to provide a very natural generalisation of both ANOVA and graph kernels on the one hand and the tree and substring kernels on the other hand. There are clear distinctions between the structures in these two cases. In the ANOVA and graph kernels the components are chosen to be the whole structure and the variation is made between the kernels used to compare the objects, while in the tree and substring kernel the variation is mainly focussed on the substructure selection with the kernels usually being identical.

These observations suggest that there is much unexplored flexibility in the $\mathcal{R}$-kernels definition that has the potential to lead to many interesting new kernels. For example using a graph structure to determine how the kernels of different sub-components are combined might be a first step in this directions. The simplest example might be combining the ANOVA construction with the recursive definition of the co-rooted tree kernel given in Example 11.70.

Such combinations raise two questions. First, are the associated $\mathcal{R}$-kernels for decomposition structures always kernels? Second, when can we expect to be able to find efficient methods for computing the $\mathcal{R}$-kernel even when the sums involved are exponentially large? We will not attempt to investigate the second question any further, but provide a brief justification that $\mathcal{R}$-kernels are indeed kernels.

Proposition 11.75 *Let $\kappa_{\mathcal{R}}$ be the kernel defined by the decomposition structure $\mathcal{R}$. It follows that $\kappa_{\mathcal{R}}$ is an inner product in an appropriately defined feature space.*

Proof We will apply Proposition 9.44 by showing that the kernel $\kappa_{\mathcal{R}}$ is a subset kernel over an appropriately defined universal set. The set in question for a data item x is $\mathcal{R}^{-1}(x)$, which by the definition of a decomposition structure is finite as required. The elements of the set are compared using the product of valid kernels and hence the subset is defined with elements that are compared with a valid kernel. The result follows. In the case of a recursive definition, an additional inductive argument must be made using structural induction over the decomposition structure. □

Remark 11.76 [Syntactical Pattern Recognition] The class of kernels we have developed in this section provides a link with the field of structural pattern recognition. Syntactic or structural pattern recognition deals with the problem of detecting common patterns in sets of structured data, typically representing them as rules. By using kernels that operate on structured data, we are 'interpolating' between statistical and syntactic pattern recognition, two branches of pattern analysis that rarely interact. ■

11.8 Summary

- Strings of symbols can be compared by kernels that sum the number of common substrings or subsequences.
- Dynamic programming techniques allow us to compute such kernels efficiently.
- A whole range of variations on this theme allow one to tune a kernel to the particular application.
- Trie-based computation provides fast evaluation of some restrictions of these kernels.
- Kernels can also be defined over more complex structures such as trees.
- Convolution kernels define a framework that subsumes both structure and ANOVA style kernels.

11.9 Further reading and advanced topics

Although the first papers dealing with kernels for strings and trees appeared in 1999, there is already a considerable literature dealing with this problem. Watkins and Haussler [155], [154], [52] certainly are to be credited with starting this research direction, where ideas from classical string matching algorithms, based on dynamic programming, were used to construct recursive kernels. A good starting point for understanding the techniques used

by them are the very readable books by Dan Gusfield [50] and Durbin *et al.* [42].

The first application of these theoretical ideas came in the field of text categorisation, with the paper of Lodhi *et al.* [91], followed by applications in bioinformatics. The computational cost of dynamic programming approaches to string kernels slowed down their uptake in larger-scale applications, until the publication of a series of papers demonstrating the use of different computational techniques for performing very similar computations at much cheaper cost, for example using tries [88], [87]. Spectrum kernels and the other methods have been applied to large real world problems in bioinformatics [85], [86]. Novel methods of computation using suffix trees are described in [151]. Our discussion of trie-based kernels was based on these papers, whereas the discussion on dynamic programming kernels goes back to Watkins.

Although motivated by string analysis, Haussler's framework is very general, and can motivate kernels for very diverse types of data. For example [23] presents the dynamic programming method for comparing two trees in the context of language parsing.

Ralf Herbrich's book presents some variations of the basic recursions for gappy string matching kernels [53]. An excellent introduction to string computations by means of dynamic programming is given in [50] and a discussion of the merits of bottom-up over top-down computations of recurrence relations can be found in [24].

Finally, recent work by Mohri, Cortes and Haffner [25] on Rational Kernels is closely related to what we have presented here, but is not discussed in this chapter for lack of space. For constantly updated pointers to online literature and free software see the book's companion website: www.kernel-methods.net