第7章　パターン認識

2枚の顔写真が同じ人物かどうかを言い当てたり、人間の声から言葉を聞きとったりする処理では、沢山のデータの中に隠れている情報を見つけるような**パターン認識**が行われる。人間のパターン認識能力に対して、コンピュータが行うパターン認識は、膨大な計算時間がかかったり、不正確な結果しか得られないことも多い。そのため、コンピュータはパターン認識が不得意だと言われることも多い。

その一方で、コンピュータによるパターン認識の手法は年々進歩しており、今では様々な情報システムに応用されている。例えば指紋や顔写真などを大きなデータベースから検索することはすでに日常的に行われているし、手書き文字や音声によってコンピュータに文章を入力する方法も実用化の域に達している。これらは人間が行っているパターン認識をコンピュータに行わせる例だが、それ以外にも、DNAやタンパクの類似性を判別するような、人間が行うのも難しい問題に使われる場面も増えている。

この章では、パターン認識の特徴と、基本的なアルゴリズムを紹介する。また、その際に重要となるアルゴリズムとして**動的計画法**もあわせて説明する。

7.1　パターン認識の特徴

パターン認識の問題は、音や画像のようなデータの中に隠れているパターンを見つけることだと言える。例えば音声認識では、波形として得られた音声データの中から、母音や子音として知られている音素を見つけることをまず行う。このときは、例えば [æ] などの音素の波形データを用意しておいて、それが現われる場所を探すわけだが、実際の音声データはゆらぎや個人差、雑音があるために用意しておいたデータと完全に一致することはない。従ってパターン認識では、照合するデータと**似ているかどうか**、そしてどのくらい似ているかを判定しなければいけない。

実際の音声認識では波形を直接比較するのではなく、波形を周波数成分ごとの強弱に変換した上で比較を行う。

実際の情報システムでは、パターン認識を何段階も使うことが多い。音声認識であれば、最初に波形データから音素の並びを得るわけだが、次は音素の並びから音節や単語を作り、さらに単語の並びから文を作るという順に認識が行われる。このときも、各単語に対応した音素の並びを用意しておいて、それとの一致を調べることになるが、最初の段階で得た音素の並びには漏れや誤りがあるので、完全に一致することはやはりない。そのため、ここでも

ある程度の誤りを許して**似ている**音素データの並びになっているかを判定しなければいけない。

別の見方をすると、音素や単語に対して 1 つのデータを用意してそれとの一致を調べるかわりに、パターン認識では、音素や単語に対して「似ている」データの集まりを考え、実際のデータがその集まりに含まれるかどうかを判定していると言える。ただし、似たデータの集まりは有限集合とも限らず、また、どれだけ似ているかを比較することも必要となるので、パターン認識のアルゴリズムには様々な工夫が必要となってくる。

7.2　アラインメント

パターン認識の 1 つとして、2 つの文字列を、一致部分が対応するように並べる**アラインメント**問題をとりあげて、そのアルゴリズムを見てみよう。2 つの文字列の一致部分が多ければ多いほど似ている文字列ということになるので、例えば DNA の塩基配列の遺伝的な近さを調べる場合などに登場する問題である。

例えば GACGGATTAG と GATCGGAATAG という 2 つの文字列のアラインメントとは以下のようなものである。

0 1 2 3 4 5 6 7 8 9 0
GA_CGGA$\underline{\text{T}}$TAG
GA$\underline{\text{T}}$CGGA$\underline{\text{A}}$TAG
○ ○ × ○ ○ ○ ○ △ ○ ○ ○

このように並べると、上の文字列は下の文字列と比べて (左端を 0 文字目として)

- 2 文字目が欠けている (そこで**ギャップ**を挿入して位置を揃えてある)、
- 7 文字目が異なっている (**不一致**)、
- その以外の文字目は**一致**している

ことが分かる。文字列の下の記号はその位置が一致 (○)、ギャップ (×)、不一致 (△) のどれであるかを表わしている。

アラインメントはいくつも考えられる。上の例であれば、

0 1 2 3 4 5 6 7 8 9 0
GA$\underline{\text{C}}$_GGA$\underline{\text{T}}$TAG
GA$\underline{\text{T}}$$\underline{\text{C}}GGA\underline{\text{A}}$TAG
○ ○ △ × ○ ○ ○ △ ○ ○ ○

0 1 2 3 4 5 6 7 8 9 0
GA$\underline{\text{C}}$$\underline{\text{G}}G_A\underline{\text{T}}$TAG
GA$\underline{\text{T}}$$\underline{\text{C}}G\underline{\text{G}}A\underline{\text{A}}$TAG
○ ○ △ △ ○ × ○ △ ○ ○ ○

をはじめとして、沢山の可能性がある。

そこで通常は

一致 (○)	+2 点
ギャップ (×)	−2 点
不一致 (△)	−1 点

のように点数を決めておき、その総和が最も大きくなるものを最良のアラインメントとして採用する。(単にアラインメントと言った場合に最良のアラインメントを指すこともある。)

上に示した 3 つのアラインメントの得点は、順に 15, 12, 9 点である。さらに、他のアラインメントを検討しても 15 点以上のものはないことが分かるので、最初に示したものが最良のアラインメントとなる。

7.3　再帰によるアラインメント

アラインメントを求める方法を説明するために、まずは 2 つの文字列 $s =$ "ATAG", $t =$ "AAC" の最良のアラインメントの得点だけを求める問題を考えてみよう。

この問題は再帰的に考えることができる。"ATAG" と "AAC" のアラインメントは、後ろの文字から考えて

1. "ATAG" と "AA" のアラインメントに、ギャップと "C" を追加した場合: $\begin{array}{|c|}\hline \texttt{ATAG-} \\ \hline \texttt{AA C} \\ \hline \end{array}$ と書くことにする。
2. "ATA" と "AA" のアラインメントに、"G" と "C" を追加した場合: $\begin{array}{|c|}\hline \texttt{ATAG} \\ \hline \texttt{AA C} \\ \hline \end{array}$
3. "ATA" と "AAC" のアラインメントに、"G" とギャップを追加した場合: $\begin{array}{|c|}\hline \texttt{ATAG} \\ \hline \texttt{AAC-} \\ \hline \end{array}$

の 3 通りである。このとき、もし $\begin{array}{|c|}\hline \texttt{ATAG} \\ \hline \texttt{AA} \\ \hline \end{array}$, $\begin{array}{|c|}\hline \texttt{ATA} \\ \hline \texttt{AA} \\ \hline \end{array}$, $\begin{array}{|c|}\hline \texttt{ATA} \\ \hline \texttt{AAC} \\ \hline \end{array}$ の最良のアラインメントの得点がそれぞれ 0 点、2 点、0 点だったとすると、上の 1 から 3 の場合の得点は、それぞれに −2 点 (ギャップ) −1 点 (不一致)、−2 点を加えて −2 点、1 点、−2 点となる。従って最良のアラインメントは 2 の場合、つまり最後の文字どうしを「不一致」させるときで、得点は 1 点だと分かる。

$\begin{array}{|c|}\hline \texttt{ATAG} \\ \hline \texttt{AA} \\ \hline \end{array}$, $\begin{array}{|c|}\hline \texttt{ATA} \\ \hline \texttt{AA} \\ \hline \end{array}$, $\begin{array}{|c|}\hline \texttt{ATA} \\ \hline \texttt{AAC} \\ \hline \end{array}$ の最良のアラインメントの得点は、それぞれ再帰的に求めることができる。例えば、$\begin{array}{|c|}\hline \texttt{ATA} \\ \hline \texttt{AA} \\ \hline \end{array}$ であれば、$\begin{array}{|c|}\hline \texttt{ATA-} \\ \hline \texttt{A A} \\ \hline \end{array}$, $\begin{array}{|c|}\hline \texttt{ATA} \\ \hline \texttt{A A} \\ \hline \end{array}$, $\begin{array}{|c|}\hline \texttt{ATA} \\ \hline \texttt{AA-} \\ \hline \end{array}$ の 3 通りを考えて $-2-2=-4$ 点、$0+2=2$ 点、$1-2=-1$ 点となり、その中の最高点 2 点となる。

また、これを続けてゆくといずれ $\begin{array}{|c|}\hline \mathtt{AT} \\ \hline \\ \hline \end{array}$ のように、一方が空文字列となる場合の計算になる。このときはもう一方の文字列の長さだけギャップを入れるしかないので、$-2 \times 2 = -4$ 点のように計算できる。両方が同時に空文字列になる場合もあるが、その場合は 0 点である。

このやり方を一般化して文字列 s の先頭から i 文字と文字列 t の先頭から j 文字の最良のアラインメントの得点を求める関数 $a(i,j)$ は、次のような関係で表わすことができる。

$$a(i,j) = \begin{cases} G \times j & (i = 0) \\ G \times i & (j = 0) \\ \max \left(\begin{array}{l} a(i,j-1) + G, \\ a(i-1,j-1) + q(s[i-1], t[j-1]), \\ a(i-1,j) + G \end{array} \right) & (i > 0 \text{ かつ } j > 0) \end{cases} \tag{7.1}$$

ただし G はギャップの得点 (ここでは -2) を表わす定数、$q(x,y)$ は $x = y$ のときは一致の得点 $(+2)$、$x \neq y$ のときは不一致の得点 (-1) となる関数である。また $s[x]$ は、文字列 s の (先頭を 0 として)x 文字目を表わす。

これを Ruby の関数として定義するとファイル 7.1 のようになる。関数 $a(i,j)$ に対応するのは 15 行目から定義されている `align_sub(s,t,i,j)` である。この定義では、`i`, `j` のどちらかが 0 の場合の計算を 17 行目の式にまとめている。また、19 行目から 21 行目までは (1)`s` にギャップを挿入する場合、(2)`s` と `t` の文字を 1 つ一致または不一致させる場合、(3)`t` にギャップを挿入する場合、の 3 つの得点を計算している。22 行目の `max` を 2 回使った式はこれらの得点の最高点を求められている。

実際に計算させてみると、以下のようになる。

```
1 irb(main):005:0> align_rec("ATAG","AAC")
2 => 1
3 irb(main):006:0> align_rec("GACG","GCAG")
4 => 2
```

練習 7.1 (アラインメントの計算) “GACG” と “GCAG” の、得点が 2 点になるようなアラインメントを手で求めて書け。

また “ACGTA” と “AGCGA” の最良のアラインメントの点数を `align_rec` を用いて求め、その点数になるアラインメントを手で求めて書け。

練習 7.2 (再帰的アラインメントの計算量)　a) `align_rec` を用いて、5 文字の文字列どうしをアラインする時間を測れ。文字列の長さを 6,7,8 と変化させたとき、時間はどのように延びるか。

```
load("./max.rb")

def g()
  -2
end

def q(x,y)
  if x==y
    2
  else
    -1
  end
end

def align_sub(s,t,i,j)
  if i==0 || j==0
    i*g() + j*g()
  else
    v0 = align_sub(s,t,i,  j-1) + g()
    v1 = align_sub(s,t,i-1,j-1) + q(s[i-1],t[j-1])
    v2 = align_sub(s,t,i-1,j)   + g()
    max(v0,max(v1,v2))
  end
end

def align_rec(s,t)
  align_sub(s,t,s.length(),t.length())
end
```

ファイル 7.1: align_rec.rb

b) s が 100 文字の文字列だったとする。s にギャップを 10 個挿入する場合、そのやり方は何通りあるか。

c) s,t がともに 100 文字だったとする。それぞれにギャップを 10 個ずつ挿入して s,t をアラインする場合、アラインメントは何通りあるか。

7.4　動的計画法

再帰によるアラインメントは、文字列が長いときには時間がかかり過ぎて実用にならない。しかし、重複した計算を避けることでより長い文字列どうしのアライメントを現実的な時間で求めることができるようになる。この工夫は**動的計画法**とよばれ、アラインメントに限らず多くの問題を効率よく解くために用いられている。

まずはより簡単な問題を使って動的計画法の考え方を理解しておこう。プロ野球の日本シリーズでは、先に 4 勝したチームが優勝となる。チーム A,B がそれぞれあと i, j 勝すれば優勝するときに、A が優勝する確率を $P(i,j)$ と表わすことにする。1 回の試合で A が勝つ確率は $1/2$ だとすると、P は次のような再帰的な関係によって求めることができる。

$$P(i,j) = \begin{cases} 1 & (i = 0 \text{ かつ } j > 0) \\ 0 & (i > 0 \text{ かつ } i = 0) \\ \dfrac{P(i-1,j) + P(i,j-1)}{2} & (i > 0 \text{ かつ } j > 0) \end{cases} \tag{7.2}$$

これを再帰的に計算する場合、同じ (i,j) の計算を何度も行うことになる。

動的計画法は、このような計算をする際に、表を用意し、表の中身を順に埋めてゆくことで同じ計算の繰り返しを避けながら計算を行う方法である。

この場合であれば、$P(i,j)$ の値を記録する表を用意して、i, j の小さい方から値を埋めてゆく。(0 行目と 0 列目はそれぞれ 0 と 1 で先に埋めておくことができる。)

$j \backslash i$	0	1	2	3	4
0		0	0	0	0
1	1	1/2	1/4	1/8	
2	1	3/4	1/2		
3	1	7/8			
4	1				

上は途中まで表を埋めた様子である。ここで例えば $P(2,3)$ の値 (表で□となっている場所) は、$(P(1,3) + P(2,2))/2$ なので、表に書かれた値を使って $(7/8 + 1/2)/2 = 11/16$ となる。

求めたい値が $P(i,j)$ であれば、$(i+1) \times (j+1)$ の表を埋めればよいので計算量は $O(ij)$ となる。これは再帰的に計算する場合よりも圧倒的に小さい。

7.5　動的計画法によるアラインメント

7.5.1　最良のアラインメントを見つける

式 7.1 は、式 7.2 と似た性質を持っている。つまり、$a(i,j)$ を求めるために $a(i,j-1), a(i-1,j-1), a(i-1,j)$ を再帰的に求めているという点であ

る。また、$i = 0$ または $j = 0$ の場合は直接点数を求めることができる。

そこで、$a(i, j)$ の値を i 行 j 列に置いたような表 A を用意して、表を左上から右下に向かって埋めてゆけば、i 行 j 列の値 (A_{ij} と書く) は、その上、左、左上の 3 つの場所の値を使って求めることができる。長さが m, n であるような文字列 s, t の最良のアラインメントであれば、大きさ $(m+1) \times (n+1)$ の表を埋めることで A_{mn} の値を求められるので、計算は 2 つの文字列の長さの積に比例する時間で終わることになる。

下は $s =$ "ATAG", $t =$ "AAC" の場合に表を作っている様子である。

	t	A	A	C	
s	$i \backslash j$	0	1	2	3
A	0	0	−2	−4	−6
T	1	−2	2	0	2
A	2	−4	□		
G	3	−6			
	4	−8			

□となっている 2 行 1 列の値は、s を先頭から 2 文字 (AT)、t を先頭から 1 文字 (A) アラインさせる場合の得点になる。$s[1] \neq t[0]$ なので、得点は $\max(A_{11} - 2, A_{10} - 1, A_{21} - 2) = 0$ となる。

最後まで表を埋めると下のようになり、A_{43} の値 1 が s, t の最良のアラインメントの得点だと分かる。

	t	A	A	C	
s	$i \backslash j$	0	1	2	3
A	0	0	−2	−4	−6
T	1	−2	2	0	2
A	2	−4	0	1	−1
G	3	−6	−2	2	0
	4	−8	−4	0	1

練習 7.3 (表を用いたアラインメント) "GACG" と "GCAG" についてアラインメントを計算する表を書き、最良のアラインメントの点数を求めよ。

7.5.2　トレースバック

表 A が完成したら、表を右下から左上に逆向きに辿ることでギャップを挿入した文字列 u, v を作ることができる。$s =$ "ATAG", $t =$ "AAC" の場合であれ

ば下の表の 4 行 3 列から出発する。最初は u, v ともに空文字列である。

	t	A	A	C	
s	$i\backslash j$	0	1	2	3
A	0	0	−2	−4	−6
T	1	−2	2	0	2
A	2	−4	0	1	−1
G	3	−6	−2	2	0
	4	−8	−4	0	[1]

4 行 3 列の値を上、左上、左の値と比較する。すると、この値は左上から $A_{32}-1$ として求められたことが分かる。(左や上から求められた場合は −2 点になる。) つまり $s[3], t[2]$ を不一致とするのが最良の場合なので、$u =$ “G”, $v =$ “C” として 3 行 2 列に移動する。

	t	A	A	C	
s	$i\backslash j$	0	1	2	3
A	0	0	−2	−4	−6
T	1	−2	2	0	2
A	2	−4	0	1	−1
G	3	−6	−2	[2]	0
	4	−8	−4	0	[1]

この場合も左上から求めたことが分かるので、u, v の先頭に $s[2], t[1]$ を加えて $u =$ “AG”, $v =$ “AC” として 2 行 1 列に移動する。

	t	A	A	C	
s	$i\backslash j$	0	1	2	3
A	0	0	−2	−4	−6
T	1	−2	2	0	2
A	2	−4	[0]	1	−1
G	3	−6	−2	[2]	0
	4	−8	−4	0	[1]

今度は上から求められた、つまり $A_{11}-2$ として求められたことが分かる。これは $s[1]$ にギャップが対応していることを意味するので、$u =$ “TAG”, $v =$ “-AC” として 1 行 1 列に移動する。

	t	A	A	C	
s	$i\backslash j$	0	1	2	3
A	0	0	−2	−4	−6
T	1	−2	[2]	0	2
A	2	−4	[0]	1	−1
G	3	−6	−2	[2]	0
	4	−8	−4	0	[1]

これは左上から求められたものなので、$u =$ “ATAG”, $v =$ “A-AC” として 0 行 0 列に移動し、終了する。このときの u, v が最良のアラインメントである。

	t	A	A	C	
s	$i\backslash j$	0	1	2	3
A	0	[0]	−2	−4	−6
T	1	−2	[2]	0	2
A	2	−4	[0]	1	−1
G	3	−6	−2	[2]	0
	4	−8	−4	0	[1]

このアルゴリズムを一般化すると、右下から i 行 j 列までのアラインメントが u, v に求められているときに、

1. A_{ij} と左、左上、上の値を比較する。
2. A_{ij} が左から求められた場合は、u の先頭にギャップを、v の先頭に $t[j-1]$ を追加し、左に移動する
3. A_{ij} が左上から求められた場合は、u, v の先頭に $s[i-1], t[j-1]$ をそれぞれ追加し、左上に移動する
4. A_{ij} が上から求められた場合は、u の先頭に $s[i-1]$ を、v の先頭にギャップを追加し、上に移動する

ことを 0 行 0 列に到達するまで続ける。最終的に得られた u, v が最良のアラインメントである。

7.5.3　動的計画法によるアラインメントのプログラム

動的計画法によってアラインメントを求めるプログラムは次の 5 つの関数からなる。

- ギャップ、一致、不一致の点数を定める g, q (ファイル 7.1 にあるものと同じなので省略)
- 表 A を作る関数 align (ファイル 7.2)
- トレースバックを行う関数 traceback (ファイル 7.3)
- 両者を使ってアラインメントを求める関数 align_dp (ファイル 7.4)

長さ m の文字列 s と長さ n の文字列 n をもとに表 A を作る関数 align(s,t) は、まず大きさ (m + 1)(n + 1) の配列を作り、0 行と 0 列を先に埋めておく。残りの部分は 12 行目からの繰り返しによって埋める。

```
1  def align(s,t)
2    m = s.length()
3    n = t.length()
4    a = make2d(m+1,n+1)
5    a[0][0] = 0
6    for j in 1..n
7      a[0][j] = a[0][j-1] + g()
8    end
9    for i in 1..m
10     a[i][0] = a[i-1][0] + g()
11   end
12   for i in 1..m
13     for j in 1..n
14       a[i][j] を計算する
15     end
16   end
17   a
18 end
```

ファイル 7.2: align.rb (関数 align)

練習 7.4 (アラインメント表の作成) ファイル 7.2 の 14 行目には、a[i][j-1], a[i-1][j-1], a[i-1][j] を用いて a[i][j] を求める命令が入る。ここを埋めて関数 align を完成させよ。(ヒント:ファイル 7.1 の 19 行目 から 22 行目)

```
1 irb(main):008:0> align("ATAG","AAC")
2 => [[0, -2, -4, -6], [-2, 2, 0, -2], [-4, 0, 1,
     -1], [-6, -2, 2, 0], [-8, -4, 0, 1]]
```

関数 traceback(a,s,t) はファイル 7.3 のようになる。引数 a は align によって作られた表 A である。この関数は表の右下から出発して、左上に到達するまでの繰り返しをするので、while 命令によって i, j が両方 0 になるまでの繰り返しを行っている。各 i, j の位置では左、左上、上のいずれかに移動するが、ここでは左に移る場合のみを示している (26 行目から 28 行目)。27 行目の式 t[j-1 .. j-1] は、文字列 t の (先頭を 0 文字目として)j-1 文字目を取り出す。

またこの関数は求めたアラインメント u, v の 2 つの文字列を答として返す必要がある。そのため、40 行目で大きさ 2 の配列を作って答としている。

最後に文字列 s, t のアラインメントを求める関数 align_dp(s,t) である。これは、align によって表を作り、traceback を使ってアラインメントを求

```
19 def traceback(a,s,t)
20   u = ""
21   v = ""
22   i = s.length()
23   j = t.length()
24   while i>0 || j>0
25     if j>0 && a[i][j] == a[i][j-1] + g()
26       u = "-"            + u
27       v = t[j-1 .. j-1] + v
28       j = j - 1 # go left
29     else
30       if i>0 && j>0 &&
31           a[i][j] == a[i-1][j-1] + q(s[i-1], t[j-1])
32         左上から求められた場合
33       else
34         if i>0 && a[i][j] == a[i-1][j] + g()
35           上から求められた場合
36         end
37       end
38     end
39   end
40   [u,v]
41 end
```

ファイル 7.3: `align.rb` (関数 `traceback`)

めるだけである。

```
42 def align_dp(s,t)
43   traceback(align(s,t),s,t)
44 end
```

ファイル 7.4: `align.rb` (関数 `align_dp`)

練習 7.5 (アラインメントを求めるプログラムの完成) ファイル 7.3 の 32 行目、35 行目を埋めてアラインメントを求めるプログラムを完成させよ。

```
1 irb(main):010:0> align_dp("ATAG","AAC")
2 => ["ATAG", "A-AC"]
3 irb(main):011:0> align_dp("GACG","GCAG")
```

```
4 => ["GAC-G", "G-CAG"]
```

練習 7.6 (RNA のアラインメント) 配布プログラム RNase_P.rb には 2 つの RNA 塩基配列を与える関数 seq0() および seq1() が定義されている。両者のアラインメントを align_dp を用いて求めよ。

練習 7.7 (スペルチェッカ) 辞書にある単語を並べた辞列 dict と、誤りのある単語 word が与えられたときに、word に最も似ている単語を答える spell(dict,word) を定義せよ。ただしここでの「最も似ている」とは、アラインメントの得点が最も高いものとする。

```
1 irb(main):018:0> spell(["align","airline","engine
    ","arrow"], "arigne")
2 => "align"
```