3F2-02

知識コンポーネントを基にした学習

Learning Based on Components

沼尾正行　　西川敬之
Masayuki Numao　　Takayuki Nishikawa

大阪大学産業科学研究所
ISIR, Osaka University

It is hard to apply inductive logic programming (ILP) systems to applications, since it has to search a huge search space. This paper proposes to define knowledge components for ILP to make search space smaller, which is shown in modeling human feelings to compose music pieces.

1. はじめに

帰納論理プログラミング (ILP) を用いて楽曲に対する個人感性モデルを獲得する研究において、獲得された知識の解析を行った。その結果、得られた知識を少数の知識コンポーネントに分解することができた。ILP を応用する場合、学習時の探索空間が膨大であることがボトルネックになっているが、各ドメインの知識コンポーネントを明確にすることで、探索空間を飛躍的に縮小することが可能になると考えられる。

現在、実用化されている機械学習手法の多くでは、例題から構造、すなわち、データ要素間の関係 (relation) を学習することができない。構造や関係は、言語やそれに準じる記号記述、たとえば、楽譜や化学式などを獲得する場合に欠かすことができない。しかしながら、関係の数はデータ要素の数に対して、組み合わせ的に増えるために、学習のための探索空間が爆発してしまい、実用的な時間で学習を完了することが困難である。実際、関係を学習するための代表的な手法である帰納論理プログラミングでは、あらかじめ適度な探索空間を想定して設定する必要があり、実用的な場面で、関係の学習能力を発揮させるのに、困難を感じる場合が多い。

筆者らは、音楽に対する被験者の感性を楽譜から獲得する実験を続けている [1]。楽譜にも構造があり、要素間の関係が重要であるため、帰納論理プログラミングの手法である FOIL[3] を使用し、十分な性能を得ている [2]。しかしながら、実験を進めるに従い、FOIL のコード中に楽譜の構造に合わせた特有のコードを挟み込むようになってきた。これにより、飛躍的に学習効率が向上しているが、ノウハウを簡潔に記述することができず、対象が変わった場合には、コードの修正が必要になっていた。本論文では、そうしたノウハウと学習器本体を分離する手法を提案する。

2. ルール学習の手法

ルール学習の手法は数多く提案されているが、その一部である関係学習については、その困難さゆえに、いろいろなバリエーションがある。ここでは基本に戻り、ルール学習のカバーリングアルゴリズム [3] に沿って議論を進めよう。

ルール学習では次の形式のルールの並びを学習する。

if 条件 *1 then* 結論 *1*
else if 条件 *2 then* 結論 *2*
else ...

これは、直線的に条件と結論を配置したものである。命題またはその否定をリテラルと呼ぶとすると、各条件はリテラルの積で記述されている。ルール (規則) は、各行:

if 条件 *then* 結論

に相当する。関係を学習するためには、'条件' 中の各リテラルを述語で記述することになる。なお、論理型言語では、これを

結論 :- 条件.

と表記するので、以降この記法を用いる。

ルール学習では、ルールの学習とルールが被覆 (cover) する例の排除を繰り返す、分離統治法 (separate and conquer) と呼ばれる手法が用いられる。この手法では、例題集合の中から正例を満たし、負例を満たさない条件を見つけ、その条件を使って規則を一つ生成し、その条件が満たす正例を例題集合から分離除去する。さらに、残りの正例について同じ過程を繰り返す。すなわち、残りの正例を満たし、負例を満たさない '条件' を見つけ、規則を一つ生成し、その条件が満たす正例を例題集合から分離除去するという過程を繰り返して、規則を生成してゆき、正例がなくなるまで続ける。

上述の '条件' はリテラルの積で記述される。リテラルに変数が含まれない場合には、あらかじめ用意されたリテラルの候補を並べて、正例のみを満たし、負例を満たさないようにすればよく、比較的簡単に学習を行うことができる。帰納論理プログラミングの場合には、リテラルに変数が含まれるため、リテラルの候補を選んだ後に、変数名を順次当てはめる必要が生じ、リテラルと引数個分の変数の候補の組み合わせとなってしまい、探索空間が爆発してしまう。帰納論理プログラミングでは、探索空間を押さえるための工夫が議論されている。しかしながら、よく考えてみると、プログラミング言語には変数が含まれているが、人がプログラミングを行う場合には、そのような膨大な探索を行うことはなく、基本的なアルゴリズムのコンポーネントをいくつか組み合わせて、はるかに速くプログラミングを行っている。機械学習の場合に、そのようなコンポーネントを利用する方法を考えてみよう。

連絡先: 沼尾正行，大阪大学産業科学研究所，〒 567-0047 大阪府茨木市美穂ヶ丘 8-1，電話: 06-6879-8425，FAX: 06-6879-8428，Email: numao(a)sanken.osaka-u.ac.jp

3. 和音記述のコンポーネント

楽譜の学習においては、楽譜全体の調や楽器など、全体の特徴を記述した frame 述語と、和音の列を記述した chords 述語が用いられている [2]。これらにより、楽譜全体の特徴および和音の列のそれぞれについて、各被験者の感じ方との対応付けを学習する。frame 述語については、述語変数は一つしか用いられておらず、述語変数を削って命題だけで記述することもでき、通常のルール学習、決定木の学習、ニューラルネットワークなどで、学習が可能である。

chords 述語については、複数の述語変数が複雑に絡まりあった和音の列構造を記述しており、記述に述語変数が不可欠である。しかし、目標とする学習結果には一定のパターンがあり、全く予測不可能な構造を学習したいわけではない。学習結果のルールは、次および付録の節の組み合わせ、すなわち、各節をプログラム変換により展開 (unfolding) したものになっている。

```
chords(A) :- chords2(A, B).

chords2(A, B) :-
        has_chord(A, B, C),

        key(C),
        moll-or-dur(C),
        chord(C),
        form(C),
        inversion(C),
        root_omission-or-not(C),
        semi_own-or-not(C),
        vice_dominant(C),
        augment_chord-or-not(C),
        change(C),
        additional(C),
        function(C),

        chords3(A, B).

chords3(A, B) :- next_to(A, B, D), chords2(A,D).
chords3(A, B). % don't care
```

ここで、has_chord(A,B,C) は、曲 A が和音 B を持ち、その和音の構成が C であることを示す。next_to(A,B,D) は、曲 A の和音 B の次の和音が D であることを示す。節中の key(C) から function(C) までの各リテラルは、和音の構成 C に関する特徴を表しており、それぞれ、付録の節と組み合わせて展開することにより、特定の特徴を満たすルールが得られる。たとえば、key(C) は、key_c(C), key_cis(C), key_d(C), ... のどれかに展開されて、それぞれの特徴を満たすルールが得られる。最後の % don't care とコメントされた節には条件部分がないので、これと組み合わせて展開すると、key(C) はルールの適用条件中から消去され、条件の判定時にこの特徴は無視されることになる。

chords(A), chords2(A,B), chords3(A,B) で始まる節を組み合わせて展開することにより、長さ 1,2,3,... の和音の列を表現するルールが枚挙できる。たとえば、それらの中で、chords(A) および chords2(A,B) で始まる節と、上の最後の節: chords3(A, B). % don't care を組み合わせて展開したものは、次のようになる。

```
chords(A) :-
        has_chord(A, B, C),
        key(C),
        moll-or-dur(C),
        chord(C),
        form(C),
        inversion(C),
        root_omission-or-not(C),
        semi_own-or-not(C),
        vice_dominant(C),
        augment_chord-or-not(C),
        change(C),
        additional(C),
        function(C).
```

この節中の key(C) から function(C) までの各リテラルを、付録の節を用いて展開すると、長さ 1 の和音列を表すルールが得られる。

これらの節を組み合わせて学習するには、辺同士を単一化することにより展開の操作を行っている。そういう意味では、これらの節を領域理論とみなせば、説明に基づく学習 (EBL:Explanation-Based Learning)[4] と同じ学習機構になる。しかし、これらの節をたどることにより、何かが説明されるわけではないし、これらの節の組み合わせは無数にあり、かなり大きな探索空間を形成するので、EBL のように一つの例題で学習できるわけではない。chords から出発し、それに chords2 を組み合わせ、その chords2 中の述語に付録のルールを組み合わせるという各ステップでは、FOIL と同様の Gain 評価関数を用いることができ、欲張り探索を行うことなどが考えられる。

注意してほしいのは、その探索の際には、変数の割り当てが不要なため、探索空間はそれほど大きくならず、述語変数のない命題論理レベルのルールの学習とあまり変わらないことである。付録によれば、各特徴の候補数は、key(C) : 15, moll-or-dur(C) : 4, chord(C) : 8, form(C) : 5, inversion(C) : 8, root_omission-or-not(C) : 3, semi_own-or-not(C) : 3, vice_dominant(C) : 8, augment_chord-or-not(C) : 3, change(C) : 6, additional(C) : 5, function(C) : 4 である。組み合わせは 497,664,000 通りしかなく、和音の長さ 1 の候補について全探索することも可能である。もちろん、長い和音列についてのルールを学習するには、何らかの探索制御が必要になる。

探索空間を狭められる反面、そのために、こうした記述を与えなければならないが、この記述は学習の言語空間を決めるものであり、言語の構文規則に相当するものである。言語の形式が定まれば、自然に記述することが可能である。

4. おわりに

知識のコンポーネントを与えることにより、ルールを学習する手法を提案した。これまでの帰納論理プログラミングは、立ちにくい卵を机の上で微妙に調整して、立たせてきたようなものだが、少し机が揺れるとすぐに倒れてしまい、実用化には微妙な調整が必要であった。コロンブスは卵の尻をつぶして立ててみせ、コロンブスの卵と言われたが、立って当たり前であり、卵を食べるには、きわめて実用的である。本手法で知識コンポーネントを定義したのは、卵の尻をつぶしたのに相当し、ルールの学習が容易になるのは、当たり前であり、理論的な面白さはない。しかし、こうした知識コンポーネントを定義する

ことは、実際の応用ではそれほど無理なことではなく、定義さえすれば、微妙な調整は一切不要になる。今後は、各種の応用分野について、知識コンポーネントを定義することを試みていきたい。

参考文献

[1] Toshihito Sugimoto, Roberto Legaspi, Akihiro Ota, Koichi Moriyama, Satoshi Kurihara, and Masayuki Numao. Modelling affective-based music compositional intelligence with the aid of ANS analyses. *Knowledge-Based Systems*, Vol. 21, No. 3, pp. 200–208, 2008.

[2] 西川敬之, 杉本知仁, 沼尾正行. 帰納論理プログラミングによる個人感性獲得とその評価. 人工知能学会 第 4 回データマイニングと統計数理研究会資料, 2007.

[3] 元田浩, 津本周作, 山口高平, 沼尾正行. データマイニングの基礎. オーム社, 2006.

[4] 人工知能学会編. 人工知能学辞典. 共立出版, 2005.

A　各特徴の候補を示すコンポーネント

```
% 調性の根音
key(C) :- key_c(C).
key(C) :- key_cis(C).
key(C) :- key_d(C).
key(C) :- key_e(C).
key(C) :- key_f(C).
key(C) :- key_fis(C).
key(C) :- key_g(C).
key(C) :- key_a(C).
key(C) :- key_h(C).
key(C) :- key_des(C).
key(C) :- key_es(C).
key(C) :- key_ges(C).
key(C) :- key_as(C).
key(C) :- key_b(C).
key(C). % don't care

% 各和音が長調か短調か
moll-or-dur(C) :- dur(C).
moll-or-dur(C) :- moll(C).
moll-or-dur(C) :- moll_dur(C).
moll_dur(C). % don't care

% 根音を何とする和音か
chord(C) :- chord_I(C).
chord(C) :- chord_II(C).
chord(C) :- chord_III(C).
chord(C) :- chord_IV(C).
chord(C) :- chord_V(C).
chord(C) :- chord_VI(C).
chord(C) :- chord_VII(C).
chord(C). % don't care

% 形体指数
form(C) :- form_V(C).
form(C) :- form_VII(C).
form(C) :- form_IX(C).
form(C) :- form_XI(C).
form(C). % don't care

% 転回指数
inversion(C) :- inversion_Zero(C).
inversion(C) :- inversion_I(C).
inversion(C) :- inversion_II(C).
inversion(C) :- inversion_III(C).
inversion(C) :- inversion_IV(C).
inversion(C) :- inversion_V(C).
inversion(C). % don't care

% 根音省略形
root_omission-or-not(C) :- root_omission(C).
root_omission-or-not(C) :- no_root_omission(C).
root_omission-or-not(C). % don't care

% 準固有和音
semi_own-or-not(C) :- semi_own(C).
semi_own-or-not(C) :- no_semi_own(C).
semi_own-or-not(C). % don't care

% 副属和音
vice_dominant(C) :- vice_dominant_II(C).
vice_dominant(C) :- vice_dominant_III(C).
vice_dominant(C) :- vice_dominant_IV(C).
vice_dominant(C) :- double_dominant(C).
vice_dominant(C) :- vice_dominant_VI(C).
vice_dominant(C) :- vice_dominant_VII(C).
vice_dominant(C) :- no_vice_dominant(C).
vice_dominant(C). % don't care

% 増和音
augment_chord-or-not(C) :- augment_chord(C).
augment_chord-or-not(C) :- no_augment_chord(C).
augment_chord-or-not(C). % don't care

% 変化和音
change(C) :- napori_VI(C).
change(C) :- doria_IV(C).
change(C) :- displacement_up(C).
change(C) :- sus4(C).
change(C) :- no_change(C).
change(C). % don't care

% 付加和音
additional(C) :- additional_VI(C).
additional(C) :- additional_IV_VI(C).
additional(C) :- additional_IX(C).
additional(C) :- no_additional(C).
additional(C). % don't care

% 和音機能
function(C) :- tonic(C).
function(C) :- dominant(C).
function(C) :- subdominant(C).
function(C).  % don't care
```