データベースからの結合ルール発見アルゴリズム
- アプリオリのビット列を用いた効率的な実装について -

大囿忠親*　新谷虎松
名古屋工業大学 知能情報システム学科
〒 466-8555 名古屋市昭和区御器所町
連絡先： 大囿忠親，〒 466-8555 名古屋市昭和区御器所町
Tel: 052-735-5467, Fax: 052-735-5477
E-mail: chika@ics.nitech.ac.jp
URL: http://www-toralab.ics.nitech.ac.jp

概 要

知識発見分野において，大規模データベースから効率よく結合ルールを発見するためのアルゴリズムとしてアプリオリが研究されてきた．レコード数及び属性数が多い場合に，アプリオリで最も時間がかかるのは，アイテム集合のサポートを計算する部分である．アイテムをサポートするレコードをビット列を用いてメモリ上に保存することで，高速にサポートの計算を実行することが可能になる．しかし，ビット列の保存には，大量のメモリが必要である．特に，データ中に長いパタンが存在すると，メモリ使用量が爆発的に増加することが問題となる．本稿では，ビット列の保存のためのメモリ量を削減するための手法を提案する．本手法は，長いパタンへの対応とビット列の圧縮により，ビット列の保存に必要なメモリ量の削減を達成する．本手法により，ビット列による高速化を達成しながらも，比較的少ないメモリ使用量で結合ルールを発見することが可能になる．

1　はじめに

本研究は，WWW 上の情報から事例ベース推論システムを半自動的に生成するための研究である．そのために現在，大量のテキストから結合ルールを自動的に抽出するための研究を行っている．知識発見分野において，大規模データベースから効率よく結合ルールを発見するためのアルゴリズムとしてアプリオリ [1] が研究されてきた．アプリオリは，サポートが閾値以上（この閾値を最小サポートと呼ぶ）となる結合ルールを発見するためのアルゴリズムである．ここで，データベース中のレコード t を，$t = \{i_1, i_2, ..., i_k\}$（$i_1, i_2, ..., i_k$ はアイテム）とする．例えば，POS のデータの場合，$t =$ “顧客 A のデータ” の場合，アイテムは，$i_1 =$ “パン”，$i_2 =$ “バター” のように，顧客 A が買った商品になる．k 個のアイテムから構成される集合を k-アイテム集合と呼ぶ．データベース中のアイテム i が，s 個のレコードに含まれているとき，i のサポートは s であるという．$\{i_1, i_2, ..., i_k\}$ のサポートは，これらのアイテムを同時に含むレコードの個数である．

レコード数及び属性数が多い場合に，アプリオリで最も時間がかかるのは，アイテム集合のサポートを計算する部分である [2]．アイテムをサポートするレコードをビット列を用いてメモリ上に保存することで，高速にサポートの計算を実行することが可能になる．ここでのビット列は，i 番目のレコードがアイテム集合を含むか否かを，i ビット目の値で表したものである．ビット列同士の AND 演算とビットカウントによって，サポートを高速に計算できる．しかし，ビット列の保存には，大量のメモリが必要である．特に，データ中に長いパタンが存在すると，メモリ使用量が爆発的に増加することが問題となる [3]．長いパタンとは，k が大きい k-アイテム集合のことである．

本稿では，ビット列の保存のためのメモリ量を削減するための手法を提案する．本手法は，長いパタンへ

の対応とビット列の圧縮により，ビット列の保存に必要なメモリ量の削減を達成する．本手法により，ビット列による高速化を達成しながらも，比較的少ないメモリ使用量で結合ルールを発見することが可能になる．

1.1 アプリオリ

以下にアプリオリの手順を示す．

1. 1 つの基本条件からなる論理積の中で閾値以上のサポートをアイテム集合の集合を L_1 とする．L_k のことをラージアイテム集合と呼ぶ．
2. 各 $k = 2, ...$ について L_{k-1} が空集合でない限り以下の手続きを繰り返す．
   (a) L_{k-1} から L_k の元の候補集合 C_k を計算．
   (b) C_k の各論理積の中で閾値以上のサポートをもつアイテム集合の集合を L_k とする．
3. アイテム集合からルールを求め，確信度が閾値以上の場合は，興味深いルールとして出力．ここでの確信度とは，ルールの条件部を満たすレコードが，どれくらいそのルールの結論部も満たしているかを表す度合いとして定義される．

本研究では，上記手順における 2-(a) と 2-(b) の部分の高速化とメモリ使用量の削減を行った．

2 ビット列の使用メモリ量の削減

ビット列の保存に必要なメモリ量を削減するためには，ビット列の数を減らすか，ビット列の大きさを減らせば良い．まず，ビット列の数を減らすために長いパタンによる組合せ爆発を回避することを目指す．本研究では，(1) 候補集合をグループ化しながら計算，(2) 長いパタンとなるグループの発見，(3) 長いパタンを直接計算，の 3 つのステップにより長いパタンによる組合せ爆発を防ぐ．

(1) の候補集合のグループ化は，文献 [3] で提案された手法を用いた．これは，アイテム間の全順序関係を定義し，このとき，k-アイテム集合の先頭から $k-1$ 個目までのアイテムが等しいアイテム集合を同じグループにするという手法である．この手法により，長いパタンを効率よく扱うことが可能になる [3]．

(2) の長いパタンとなるグループの発見は，あるグループに属するすべてのアイテム集合のサポートを表すビット列の AND と OR をとり（それぞれ B_{and} と B_{or} とする），B_{and} と B_{or} が等しいかを調べれば良い．B_{and} と B_{or} が等しいことは，全てのアイテム集合が，同じレコードの集合に含まれていることを意味する．このような場合，このグループに含まれるアイテム集合の和集合もまた興味深いアイテム集合となるので，このようなグループは長いパタンとなるグループである．

(3) の長いパタンを直接計算では，(2) で発見されたグループ内のアイテム集合の和集合を新たなアイテム集合とする．このグループが，k-アイテム集合のグループで，このグループが n 個のアイテム集合を含んでいるとき，$(k+n-1)$-アイテム集合が生成される．このアイテム集合は，最小サポートを満たしているので，サポートを計算する必要はない．

(2) で発見されたグループから本手法により生成される長いパタンのアイテム集合は，そのグループから本手法を用いないときに生成される最も長いパタンのアイテム集合と一致する．本手法を用いないときに生成されるはずのアイテム集合とは，得られた長いパタンのアイテム集合の部分集合の集合である．よって，途中で生成されるはずのアイテム集合は，得られた長いパタンのアイテム集合から容易に生成できる．

2.1 候補集合のグループ化

図 1 は，本アルゴリズムで用いるアイテム集合のデータ構造を表している．本アルゴリズムでは，図 1 のように，アイテム集合を頭部と尾部に分割して保存する．k-アイテム集合の頭部とは，k-アイテム集合に含

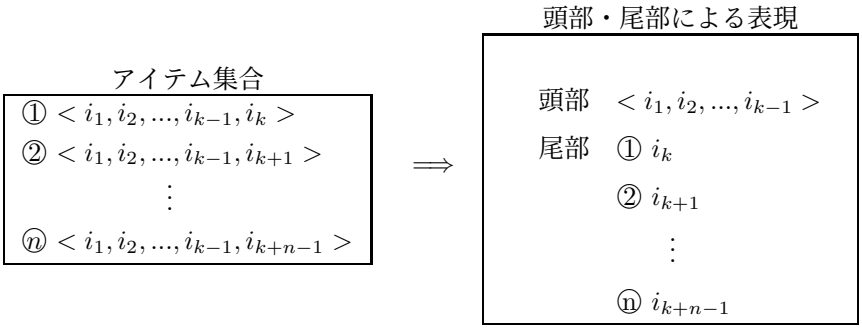

図 1: 頭部・尾部による表現

	実行時間（秒）
無圧縮	0.05
0 圧縮	1.2
zlib	14.2

表 1: 単純に圧縮した場合の実行時間

まれる，$(k-1)$-アイテム集合である．尾部は，k-アイテム集合から頭部を除いたものである．頭部の等しいアイテム集合を一つのグループとして集め，それらのアイテム集合の頭部と，尾部のリストを保存している．これにより，頭部の等しいグループの管理・生成が簡単になる．

2.2 ビット列の圧縮

本研究では，まずビット列を汎用の圧縮アルゴリズムで圧縮することを試みた．ここでは圧縮率の高い汎用圧縮ライブラリ zlib [4] と圧縮率は低いが高速なゼロ圧縮を試した．zlib は，ゼロ圧縮に比べ低速である．実験として，30 ページの Web ページを用いた．形態素解析により，Web ページをキーワードの集合に変換し，キーワードをアイテムとして，アプリオリを適用した．ここでアイテム数は，126 である．すなわち，レコード数 30，アイテム数 126 である．最小サポート値は，3 である．実験環境は，アップル社の PowerBook G3 （CPU: PowerPC 750 400MHz，メモリ: 320MB）を用いた．結果を表 1 に示す．結果として，ゼロ圧縮ですら速度的に問題があり，zlib ですら圧縮できないビット列が頻出することもわかった．すなわち単純に圧縮アルゴリズムを適用するだけでは，不十分であることがわかった．

本研究では，B_{or} を利用してビット列を不可逆圧縮することで高速にビット列を圧縮するための方法を開発した．本手法の特長は，(1) グループごとに圧縮，(2) 伸張が不必要，の 2 点である．(1) のグループごとに圧縮は，同一のグループに属するアイテム集合のビット列が類似しているので，グループごとに圧縮することで効率よく圧縮できる．(2) の伸張が不必要は，ビット列を伸張せずにサポート値の計算が可能であることを表している．本圧縮手法では，B_{or} が 0 であるビットの位置と同じ位置のビットをビット列から除去する．図 2 は，本手法によるビット列の圧縮を示している．図中の矢印の左側が，圧縮前のビット列を表し，右側が圧縮後のビット列を表している．ここでは，B_{or} が 1 の部分（灰色部分）だけが残され，あとは，除去される．B_{or} の n ビット目が 0 であるということは，そのグループに含まれるすべてのアイテム集合が n 番目のレコードに含まれないことを表しており，このグループに関してはこのビットが使われることはない．よって，このビットは除去可能であると判断できる．本手法の特長は，単に未使用のビットを除去するだけなので，圧縮後のビット列を伸張せずにサポートの計算が可能である点である．これにより，全体的な処理の高速化が達成できる．

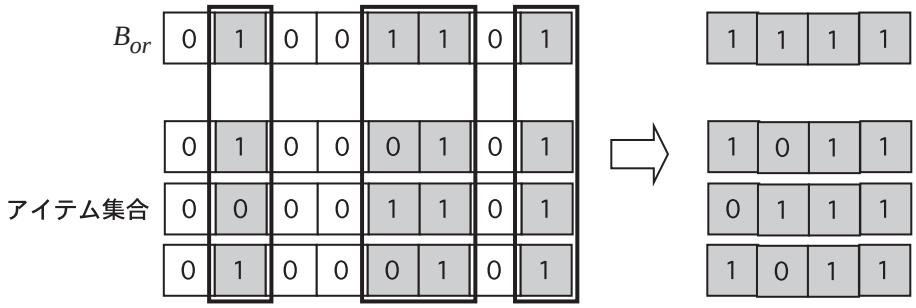

図 2: ビット列の圧縮

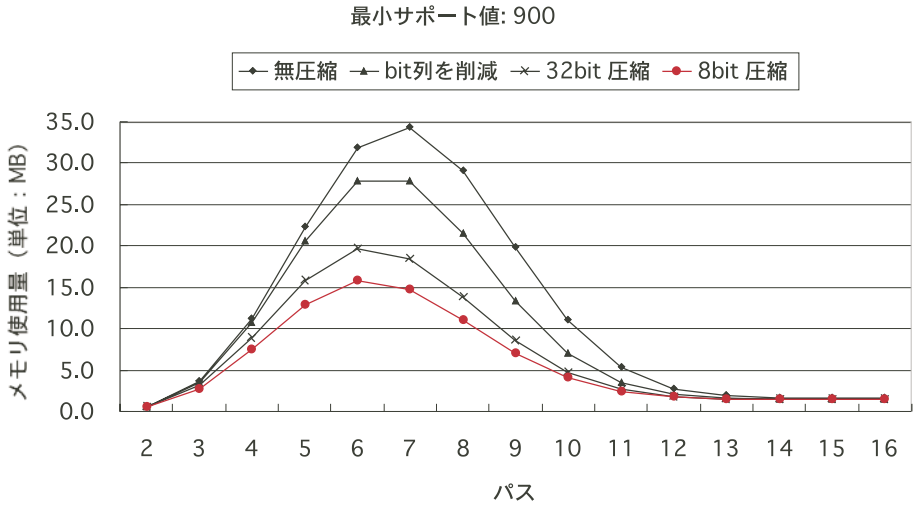

図 3: 実験の結果

2.3 評価

実験では，UCI Machine Learning Repository [5] の mushroom を実験データとして用いた．mushroom は，サンプル数 8124，属性数 22 のデータである．使用した計算機は，アップル社の PowerBook G3（CPU：PowerPC 750（動作周波数：400MHz），メモリ搭載量：320MB）である．実験では，200MB のメモリを使用した．本手法を用いない場合，最小サポートが 900 までしか処理できなかったが，本手法を用いた場合最小サポートが 500 でも成功した．処理速度に関しても，ビット列による高速化を行っただけのときは，42.2 秒，長いパターンの処理を用いただけの場合は 32.4 秒で，さらにビット列の圧縮まで用いた場合は 29.4 秒と，約 30 % 高速化した．現在，より詳細な実験を行っている．

図??は，実験の結果をグラフ化したものである．横軸がパスを表し，縦軸が使用メモリ量を表している．図は，無圧縮の場合，ビット列を削減，32bit ごとに圧縮（32bit 圧縮）した場合，そして 8 ビットごとに圧縮（8bit 圧縮）した場合の，パスとメモリ使用量の関係を表している．ここで，ビット列を削減した場合では，長いパターンを直接計算することでビット列を削減した．32bit 圧縮と 8bit 圧縮では，長いパターンを直接計算することに加えて，ビット列の圧縮も行った．図からわかるように，メモリ使用量が最大になるパス 7 において，無圧縮の場合に比べ，8bit 圧縮は，約 43%のメモリ容量しか必要としない．32bit 圧縮の場合は，8bit 圧縮に比べ 4MB 程メモリ使用量が増加している．図には示されていないが，32bit 圧縮は 8bit 圧縮に比べ処理速度が 1 割程速い．

3　おわりに

本稿では，ビット列を用いたアプリオリの効率的な実装について述べた．ビット列を用いることでアプリオリは，高速な実行が可能であるが，大量のメモリを必要とするという問題がある．本稿では，アイテム集合をグループ化して長いパタンを直接生成することでビット列の数を減らす手法と，グループ化に基づくビット列の圧縮手法を提案した．汎用的な圧縮アルゴリズムによる圧縮は，圧縮・伸張のための処理時間が問題となるが，本圧縮手法は，B_{or} を利用してビット列を不可逆圧縮することで高速にビット列を圧縮する．さらに本手法により圧縮されたビット列は，サポートの計算時に伸張処理を必要としない．これらにより，全体の処理時間やメモリ使用量に関して大幅な改善を達成できた．さらに，本手法とハッシュ木を組み合わせて適用することにより，さらなるメモリ使用量の削減も可能になる．

参考文献

[1] Agrawal, R. and Srikant, R., “Fast algorithms for mining association rules,” Proc. of the 20th VLDB Conference, pp. 207–216 (1994)

[2] 森下真一，中谷明弘, “ データマイニングの理論と実装 ”，コンピュータソフトウェア，17（1），pp. 59–72（2000）

[3] Roberto J. Bayardo Jr., Efficiently Mining Long Patterns from Databases, ACM-SIGMOD, pp. 85–93 (1998)

[4] “zlib Home Page,” http://www.cdrom.com/ pub/infozip/zlib/

[5] Blake, C.L. and Merz, C.J., “UCI Repository of machine learning databases,” http://www.ics.uci.edu/∼mlearn/MLRepository. html, University of California, Irvine, Dept. of Information and Computer Sciences (1998)

付録 参考までに本アルゴリズムの概要を示す

```
function generate_candiate_group(SL_k, min_support) : L_{k+1}
    LS_k =
        {<boolean, basebits: bit sequence, items: IS>}
begin
    rbits_and, rbits_or : bit sequence
    set all bits of rbits to 1
    foreach c_1 ∈ L_k do begin
        C_s = {}
        tail_1 := c_1.tail
        recbits_1 := c_1.rbits
        – グループを生成
        foreach c_2 after c_1 ∈ L_k do begin
            tail_2 := c_2.tail
            recbits_2 := c_2.rbits
            recbits := recbits_1 ∧ recbits_2
            s := count(recbits)
            if s < min_support then continue
            rbits_and := rbits_and ∧ recbits
            rbits_or := rbits_or ∨ recbits
            C_s := C_s ∪ {< tail_2, recbits >}
        end
        if |C_s| > 0 then
            set(basebits, tail_1)
            if |C_s| > 1 ∧ rbits_and = rbits_or then
                <true, basebits, C_s >
                ioLk[v.k - 3].push(v);
            else
                <false, basebits, C_s >
                iL2.push(v);
            end
        end
    end
end
```

図 4: グループ化された候補集合を計算するアルゴリズム