MOARLE: 高速・省メモリな行列圧縮計算フレームワーク

小山田昌史† 成田 和世† 劉 健全† 荒木 拓也†

† 日本電気株式会社 グリーンプラットフォーム研究所 〒 211-8666 神奈川県川崎市中原区下沼部 1753
E-mail: †m-oyamada@cq.jp.nec.com, ††{k-narita,j-liu}@ct.jp.nec.com, †††t-araki@dc.jp.nec.com

あらまし 行列計算は，データマイニング処理，機械学習処理，情報検索処理をはじめとするさまざまなデータ処理の中心をなす．行列計算の対象となる行列のサイズは，計算資源の発展やビッグデータの普及に伴い巨大となっている．巨大な行列はコンピュータの記憶領域を逼迫し，またその計算に長大な時間を要する．そのため，巨大な行列の記憶容量と計算時間を削減することは肝要な課題である．この課題に対し，本稿は行列の記憶容量削減，計算高速化，計算省メモリ化を同時に実現する行列の圧縮計算フレームワーク **MOARLE** を提案する．**MOARLE** は，事前に行列を連長圧縮して行列の記憶容量を削減した上で，その後何度もおこなわれる計算を連長圧縮情報の利用により高速かつ省メモリにおこなう．疎行列表現とは異なり，**MOARLE** は密行列に対しても効果的である．実データを用いた実験により **MOARLE** は内積計算やユークリッド距離計算などのコア演算処理におけるメモリ使用量を最大で 98%削減し，また処理速度を最大で 124 倍高速化することが分かった．

キーワード 行列計算，ベクトル計算，機械学習，情報検索，圧縮，組み合わせ最適化

1. 背 景

行列計算は，データマイニング処理，機械学習処理，情報検索処理をはじめとするさまざまなデータ分析処理の中心をなす．行列計算の対象となる行列のサイズは，計算資源の発展やビッグデータの普及に伴い巨大となっている．巨大な行列はコンピュータの記憶領域を逼迫し，またその計算に長大な時間を要する．そのため，巨大な行列の記憶容量と計算時間を削減することは肝要な課題である．

本稿はデータ分析処理の持つ二つの特徴に着目し，巨大な行列の記憶容量と計算時間を削減するという課題に取り組む．一つ目の特徴は，データ分析処理においては同じ行列に対して何度も繰り返し計算がおこなわれるというものである．例えば，機械学習処理におけるハイパーパラメータの調節処理 [4] や，パターン認識や情報検索の分野におけるクエリベクトルとデータベース内のベクトル群とのマッチング処理がこれにあたる．二つ目の特徴は，データ分析処理においては多くの場合に行列の行や列の順序を入れ替えても演算結果が変化しないというものである．例えば機械学習処理は行をサンプルデータ，列を特徴量次元とする計画行列を対象に目的関数の最適化をおこなうが，この際よく用いられる確率的勾配法 [5, 8] や Coordinate Descent 法 [9] では，行や列の順序を入れ替えても最終的に得られる結果は変化しない [8]．また，前述したクエリベクトルとデータベース内のベクトル群とのマッチング処理ではユークリッド距離やコサイン類似度による近傍探索をおこなうが，この際にデータベース内のベクトル群の順序やベクトル次元の順序を入れ替えても，探索結果は変化しない．

我々は，これら二つのデータ分析処理の特徴をうまく利用して行列の記憶容量削減，計算高速化，計算高速化を同時に実現する計算処理フレームワーク**MOARLE** (*Matrix Operation Accelerator based on Run-Length Encoding*) を提案する．**MOARLE** は事前に計算対象となる行列を連長圧縮して行列の記憶容量を削減したうえで，その後の計算を連長圧縮により得られた情報の利用により高速かつ省メモリにおこなう．圧縮処理に必要な初期コストは，何度も計算がおこなわれることで償却される．

MOARLE は行列の圧縮部と計算部からなる．圧縮部に対して，我々は圧縮形式として基本的な**連長圧縮** (*run-length encoding*) を用いる．そして，データ分析処理においては行ないし列の順序が多くの場合演算結果に影響をおよぼさないことに着目し，行ないし列を並び替えることで，連長圧縮の効果を向上させる方式を提案する．また，行列内の要素を少数の代表値で置換することにより精度を犠牲にしつつ圧縮率をさらに向上させる代表値置換処理も提案する．圧縮情報を利用した計算部分では，連長圧縮処理によりベクトル内で同一の値が連続するという事前知識が得られることに着目し，その知識を利用して計算回数を減らす方式を提案する．

本研究の貢献は以下のとおりである:

（1） 行列の行ないし列方向に並ぶ要素を連長圧縮してその情報を計算時に活用することで行列の記憶容量の削減，計算の高速化，計算の省メモリ化を同時に達成する行列の圧縮計算フレームワーク **MOARLE** を提案する．

（2） 行列に対する演算の種類によっては行ないし列の順序が演算結果に影響をおよぼさないことに着目し，行ないし列の並びかえをおこない連長圧縮の効果を最大化させる問題を組み合わせ最適化問題として定式化する．

（3） 連長圧縮の効果を最大化させる問題について，最適解を求めるアルゴリズムが多項式時間では動作しないことをあきらかにしたうえで，行列の辞書ソートにもとづいて近似解を求めるアルゴリズムを提案する．

（4） 行列内の要素を少数の代表値で置き換える方式を提案し，演算精度を犠牲にしつつ連長圧縮の効果をさらに高めることを可能とする．

本稿の構成は以下のとおりである．はじめに，2. 節で，行列に連長圧縮を適用することによる行列計算の省メモリ化・高速化について述べる．次に，3. 節で，行列の行ないし列を並び替えることにより連長圧縮の効果を高める方式について述べる．そして 4. 節で，行列内の要素を少数の代表値で置換することにより連長圧縮の効果をさらに高める方式について述べる．その後 5. 節で提案方式の有効性を実データを用いた実験により確かめ，6. 節で関連研究について紹介したあと，7. 節でまとめる．

2. 行列の圧縮計算

本節では，**MOARLE** における計算部のコア技術である，行列に連長圧縮を適用することによる計算の省メモリ化と高速化について述べる．

2.1 行列の連長圧縮

はじめに，データ圧縮の基礎的な技術である**連長圧縮** (*run-length encoding, RLE*) について説明する．連長圧縮はデータ列を可逆圧縮する技術である．連長圧縮は，データ列を (データが連続する個数, データ) というペアの列として表現する．そのため，同一のデータが頻繁に連続して登場するようなデータ列を，元のデータ列よりもコンパクトに表現することができる．例えば

$$1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1$$

という 17 個の数値からなるデータ列を連長圧縮すると，

$$(5, 1), (9, 2), (3, 1)$$

という 6 個の数値からなるデータ列になり(注1)，データサイズが削減される．なお，連長圧縮の問題として，同一データがあまり連続しないデータ列に対して連長圧縮を適用した場合に圧縮後のデータサイズが元のデータサイズよりも大きくなりうるというものがある．この問題は，連長圧縮のシステム実装に PackBits [2] と呼ばれる技術や γ 符号・δ 符号などの符号化技術 [12] をつかうことにより，緩和可能である．

簡単に PackBits について説明する．PackBits は，連長圧縮において同一の要素が連続しない場合に，同一の要素が連続しない回数を負の値で表現したうえで，連続しない要素をその後に並べる．これにより，異なる要素が並んだ場合にサイズが肥大することを防ぐ．例えば

$$1, 2, 3, 4, 1, 1, 1, 1$$

という 8 個の数値からなるデータ列を単純に連長圧縮すると，

$$(1, 1), (1, 2), (1, 3), (1, 4), (4, 1)$$

という 10 個の数値列となって圧縮前と比べデータサイズが増えてしまうが，PackBits をつかうと

$$(-4, 1, 2, 3, 4), (4, 1)$$

という 7 個の数値列となり，圧縮効果が得られる．本稿では簡単のため，以降の例では単純な連長圧縮を用いて説明するが，提案方式は PackBits を利用した場合にも適用可能である．単純な連長圧縮と PackBits を用いた場合の提案方式の性能差については，5. 節で議論する．

つぎに，行列に対する連長圧縮について説明する．**MOARLE** は行列の行ないし列に対して連長圧縮を適用したうえで，圧縮情報を利用して計算を高速化する．ここで，行列の行（列）に対する連長圧縮とは，行列の各行（列）をそれぞれ独立の数列，すなわちベクトルとみなしたうえで連長圧縮を適用することを意味する．例えば

$$\begin{pmatrix} 1 & 1 & 2 & 2 & 2 \\ 2 & 2 & 1 & 2 & 2 \\ 2 & 3 & 3 & 3 & 3 \\ 2 & 3 & 3 & 2 & 2 \\ 2 & 3 & 1 & 2 & 2 \end{pmatrix} \tag{1}$$

という行列に対して，行を連長圧縮すると

(2, 1)	(3, 2)		
(2, 2)	(1, 1)	(2, 2)	
(1, 2)	(4, 3)		
(1, 2)	(2, 3)	(2, 2)	
(1, 2)	(1, 3)	(1, 1)	(2, 2)

となり，列を連長圧縮すると

(1, 1)	(1, 1)	(1, 2)	(2, 2)	(2, 2)
(4, 2)	(1, 2)	(1, 1)	(1, 3)	(1, 3)
	(3, 3)	(1, 3)	(2, 2)	(2, 2)
		(1, 1)		

となる．行列に対して連長圧縮を適用することで，巨大な行列をコンパクトに表現することができる．

2.2 連長圧縮を利用した計算の高速化

先に述べた行列の連長圧縮は，行列のデータサイズを削減するためだけでなく，行列に対する計算の高速化に利用することができる．連長圧縮はデータ列を (データが連続する個数, データ) というペアの列として表現するものであることを思い返す．これはすなわち，連長圧縮されたデータには「ある値が何度連続して登場するか」という事前知識が含まれていることを意味する．以下で，この事前知識を利用して計算を高速化する方式について例を用いて説明する．なお，本節で説明する方式は圧縮されたデータの伸張を必要としない，すなわちデータを圧縮したまま計算することが可能な方式であるため，計算の高速化と同時に計算の省メモリ化も同時に実現する．

2.2.1 二乗和計算の高速化

はじめに，単純な例として行列の列に関する二乗和計算の高速化について説明する．行列の列に関する二乗和計算は，凸最適化問題の解法としてよく用いられる Coordinate Descent 法 [9] などのさまざまな処理で利用されるため，その高速化は重要である．

(注1)：なお，括弧は読みやすさのために付与しているものである．

例として，式 1 の行列の一列目に対する二乗和計算について考える．単純な二乗和計算は

$$1^2 + 2^2 + 2^2 + 2^2 + 2^2 \tag{2}$$

となり，5 回の積算と 4 回の加算処理を要する．これに対し，圧縮計算ではまず行列を列方向に圧縮し，一列目として

$$(1, 1), (4, 2) \tag{3}$$

を得る．そして，1 が 1 回，2 が 4 回，連続するという知識をつかい，この列に対する二乗和計算を

$$1^2 + 4 \cdot 2^2 \tag{4}$$

としておこなう．元の二乗和計算（式 2）が 5 回の積算と 4 回の加算処理を要したのに対し，圧縮二乗和計算（式 4）は 3 回の積算と 1 回の加算処理でおこなえ，積算と加算の回数が少なくなっている．そのため，計算が高速におこなえる．

2.2.2 内積計算の高速化

つぎに，行列の行に対する内積計算の高速化について説明する．内積計算はユークリッド距離やコサイン類似度の計算をはじめとする機械学習・情報検索分野での計算に頻出するため，その高速化は重要である．

例として，式 1 の行列の一行目と二行目の内積を計算することを考える．単純な内積計算は

$$1 \cdot 2 + 1 \cdot 2 + 2 \cdot 1 + 2 \cdot 2 + 2 \cdot 2 \tag{5}$$

となり，5 回の積算と 4 回の加算処理を要する．これに対し，行列を行方向に連長圧縮すると，一行目と二行目はそれぞれ

$$(2, 1), (3, 2) \tag{6}$$

と

$$(2, 2), (1, 1), (2, 2) \tag{7}$$

のようになる．ここで，二乗和計算と同様に値の連続性の知識を用いると，内積の圧縮計算は

$$1 \times (2 \times 2) + 2 \times (1 \times 1 + 2 \times 2) \tag{8}$$

となる．このとき元の内積計算（式 5）が 5 回の積算と 4 回の加算処理を要したのに対し，圧縮内積計算（式 8）は 2 回の積算と 4 回の加算処理でおこなえ，積算の回数が少なくなっている．そのため，計算が高速におこなえる．

2.3 ユークリッド距離計算の高速化

以上で説明した二乗和計算と内積計算の高速化により，多くの計算処理が高速化できる．ここでは，その例としてユークリッド距離計算の高速化について説明する．ユークリッド距離計算は近傍探索処理やクラスタリング処理など多くのデータ処理で支配的な計算となり，ユークリッド距離計算の高速化によって，多くのデータ処理が高速化される．

ベクトル $\mathbf{a} = (a_1, ..., a_n)$ と $\mathbf{b} = (b_1, ..., b_n)$ のユークリッド距離は

$$\sqrt{\sum_i^n (a_i - b_i)^2} \tag{9}$$

で定義される．このとき式 9 は

$$\sqrt{\sum_i^n (a_i - b_i)^2} = \sqrt{\sum_i^n a_i^2 + \sum_i^n b_i^2 - 2\sum_i^n a_i b_i} \tag{10}$$

と展開でき，展開後の各項は二乗和と内積計算となるため，前述した圧縮計算が適用できる．そのため，圧縮計算によりユークリッド距離計算が高速化できる．圧縮計算が可能な演算には他にコサイン類似度などがあるが，本稿ではこれ以上の例示はしない．

3. 行列の並び替えによる圧縮率の向上

前節で，**MOARLE** における計算部のコア技術として，行列を連長圧縮してデータサイズを減らす方式と，圧縮情報を用いて計算を高速化する方式について説明した．本節では，行列の行ないし列を並び替えることで行列の連長圧縮率が向上することを示し，最良の並びを求める問題が多項式時間で解けないことをあきらかにしたうえで，**MOARLE** における圧縮部のコア技術として，効率的に近似解を得る並び替え方式を提案する．

3.1 行列の並び替えと圧縮計算

本小節では，行列の内容と計算の種類によっては行列の行ないし列を並び替えることで行列全体の連長圧縮率を向上させたうえで演算が可能であることを述べる．

まず，つぎの行列 $X \in \mathbb{R}^{6\times3}$ を考える．

$$X = \begin{pmatrix} 2 & 1 & 2 \\ 1 & 0 & 1 \\ 5 & 1 & 1 \\ 4 & 0 & 2 \\ 3 & 2 & 1 \\ 1 & 2 & 1 \end{pmatrix} \tag{11}$$

行列 X を列方向に連長圧縮すると

(1, 2)	(1, 1)	(1, 2)
(1, 1)	(1, 0)	(2, 1)
(1, 5)	(1, 1)	(1, 2)
(1, 4)	(1, 0)	(2, 1)
(1, 3)	(2, 2)	
(1, 1)		

のように計 15 個のペアによって行列全体を表現することとなる．このとき，行列 X に対して行列の基本変形である行の交換処理をおこない行の並びを変化させると，行列の列方向の連長圧縮率を高めることができる．行列 X の行を並び替えて列方向の圧縮率を最良とした行列を X^* と表記すると，行列 X^* は

$$X^* = \begin{pmatrix} 2 & 1 & 2 \\ 5 & 1 & 1 \\ 3 & 2 & 1 \\ 1 & 2 & 1 \\ 1 & 0 & 1 \\ 4 & 0 & 2 \end{pmatrix}$$

となる．行列 X^* を列方向に連長圧縮すると

(1, 2)	(2, 1)	(1, 2)
(1, 5)	(2, 2)	(4, 1)
(1, 3)	(2, 0)	(1, 2)
(2, 1)		
(1, 4)		

のように計 11 個のペアによって行列全体を表現でき，行列 X と比べ連長圧縮率が向上している．

ここで，行列に対する演算によっては行列の行ないし列の順序を入れ替えても結果に影響しない，という性質に注目する．本稿では，行の順序が入れ替わっても結果が変化しない演算を**行順序不問演算** (*row-order insensitive operation*)，同様に列に対して**列順序不問演算** (*column-order insensitive operation*)，そして行と列どちらの順序を入れ替えても結果に影響しない演算を**行列順序不問演算** (*row-column-order insensitive operation*) と呼ぶ．例えば 2.2 節で例としてあげた行ベクトルないし列ベクトルに対する二乗和計算と内積計算は，行列順序不問演算になる．これらの処理は，パターン認識や情報検索の分野におけるクエリベクトルとデータベース行列内ベクトルのマッチング処理などに頻出する．また，データベース分野におけるリレーショナルモデルではレコードをあらわす行に順序関係はなく，リレーショナルデータを対象とする関係演算は行順序不問演算となる．さらに，機械学習の分野では教師データとしてサンプルを行，特徴量次元を列とする**計画行列** (*design matrix*) を用いるが，計画行列を学習対象するアルゴリズムの多く [8,9] は，サンプルの順序や特徴量次元の順序を入れ替えても最終的に同じ結果へ収束し [8]，行列順序不問演算となる．

3.2 問題定義

MOARLE は，行列に対しておこなわれうる演算が判明しており，その演算が行列順序不問演算であった場合，行列の行ないし列の順序を入れ替えて連長圧縮率を高めたうえで，圧縮計算する．そのため，いかにして行列の行ないし列の順序を入れ替えて連長圧縮を高めるかが，課題となる．本節の残りでは簡単のため議論を行列の行の並び替えに限定したうえで，つぎの問題に取り組む．

問題 1 (圧縮率を高める並び替え探索) m 行 n 列の行列 $X \in \mathbb{R}^{m \times n}$ が与えられ，$Pair_{\#} : \mathbb{R}^{m \times n} \to \mathbb{N}$ を行列を列方向に連長圧縮した際のペア数を算出する関数，$\mathbb{X}$ を行列 X に行の交換作業を繰り返して得られる全ての行列からなる集合としたとき，

$$\begin{aligned} &\textit{minimize:} && Pair_{\#}(X') \\ &\textit{subject to:} && X' \in \mathbb{X},\ Pair_{\#}(X') \leqq Pair_{\#}(X) \end{aligned}$$

となるような行列 X' を得る．

3.3 全探索法

はじめに，問題 1 の最適解を探索する**全探索法**について説明する．全探索法は，行列 X の行を並び替えて得られる全ての行列のパターンを走査し，そのなかから最適な行列を選ぶ．並び替えで得られる行列のパターンは $m!$ 通りあり，それぞれの行列に対してペア数を確認するので，この方式の計算量は $O(m!\,n)$ となる．全探索法を用いると最適な圧縮率を示す並び替えパターンを必ず得ることができるが，計算量が多項式時間ではないため，巨大な行列に対しては適用ができず，実用的ではない．

本節の残りでは，最適解を算出可能だが計算時間が多項式時間でない全探索法に代わり，多項式時間で実行可能解を算出する方式である貪欲法とスコア付き辞書ソート法の二つを紹介する．

3.4 貪欲法

多項式時間で実行可能解を算出する方式である**貪欲法**について説明する．貪欲法は，貪欲的アプローチで行の並び替えをおこない，行列の圧縮率を高める．並び替えは，ある開始行を選択して先頭に並べたうえで，選択された行とハミング距離が最も小さな行をつぎに並べる，という処理を繰り返すことでおこなわれる．完全な処理の流れをアルゴリズム 1 に示す．なお，アルゴリズム中，行列の A の i 行目を行列 B の j 行目で置き換えることを $a_{i*} \leftarrow b_{j*}$ と，m 行 n 列の零行列を $O_{m,n}$ と，それぞれ表記する．また，簡単のため arg min は集合でなく要素を返す，すなわち対象となる関数が最小値をただ一点でとるものとする．

貪欲法の計算量について議論する．貪欲法では，開始行の選び方は m 通りあり，開始行それぞれについてハミング距離が最小となる行を貪欲的に選んでいくパターンは $(m-1) + (m-2) + \ldots + 1 = m(m-1)/2$ 通りある．また，各ハミング距離の計算には n 回の比較が必要である．そのため，貪欲法の計算量は $O(m^3\,n)$ となる．これは多項式時間であるが，行数 m に対して m^3 のオーダーとなっており，巨大な行列への適用は難しい．

3.5 スコア付き辞書ソート法

貪欲法の計算量は行数 m に対して m^3 のオーダーとなり，巨大な行列を並び替えることが困難である．これに対し，我々は $m \log m$ のオーダーで行を並び替えることのできる**スコア付き辞書ソート法**を提案する．はじめに，スコア付き辞書ソート法の基礎となる行列の行の**辞書ソート** (*lexicographical sort*) について説明する．行列の行の辞書ソートとは，行列の行を文字列に見立てて行の大小関係を**辞書式順序** (*lexicographical order*) により定義したうえで，汎用のソートアルゴリズムにより並び替えることである．すなわち行列 X に対し，まず一列目の要素についてソートし，その次に二列目についてソートし，という作業を繰り返す．例えば 3.1 節の式 11 であげた行列 X の行を辞書ソートすると，

Algorithm 1: 貪欲法

Input: $X \in \mathbb{R}^{m \times n}$, 並び替え対象の行列
Output: 並び替えられた行列
Data: P, 行プール; $\mathbb{X}_{\text{cand}}$, 候補解集合; X', 並び替え済行列

foreach $i \in \{1, ..., m\}$ **do**
  $X' \leftarrow O_{m,n}$
  /* 行列 X 内の i 番目の行を開始行として選択し，それ以外の行をプール P に追加 */
  $P \leftarrow \{1, ..., m\} \setminus \{i\}$
  $x'_{1*} \leftarrow x_{i*}$
  for $k \leftarrow 2$ **to** m **do**
    /* 開始行 i に対してハミング距離が最も小さい行 j をプール P から取り出し，並び替え済行列に追加 */
    $j \leftarrow \arg\min_{j \in P} \text{HammingDistance}(X, k-1, j)$
    $P \leftarrow P \setminus \{j\}$
    $x'_{k*} \leftarrow x_{j*}$
  $\mathbb{X}_{\text{cand}} \leftarrow \mathbb{X}_{\text{cand}} \cup \{X'\}$
return $\arg\min_{X' \in \mathbb{X}_{cand}} Pair_{\#}(X')$

$$X_{\text{lex}} = \begin{pmatrix} 1 & 0 & 1 \\ 1 & 2 & 1 \\ 2 & 1 & 2 \\ 3 & 2 & 1 \\ 4 & 0 & 2 \\ 5 & 1 & 1 \end{pmatrix} \tag{12}$$

となる．

行列の行に対する辞書ソートは列番号の若い列を優先的にソートするが，これは場合によって連長圧縮率の向上に効果的でない．例えば式 11 の行列 X を辞書ソートすると一列目が優先的にソートされて式 12 の行列 X_{lex} となるが，この結果の連長圧縮率は低く，三列目を優先してソートした方が連長圧縮率は高くなる．そこで，本稿では辞書ソートを一般化した**スコア付き辞書ソート** (*scored lexicographical sort*) を提案する．スコア付き辞書ソートは，行列の各列についてスコアを計算したうえで，そのスコアの高い順に行をソートする．効果的なスコア定義としては，列内の要素を集合としてみなしたときの**濃度** (*cardinality*) を用いたものがある．濃度は列内に登場する値の種類数をあらわすため，この値が小さな列を優先してソートすると，同じ値が列方向に連続して並ぶ確率が高くなる．その結果，全体として列方向の連長圧縮率が高くなることが期待される．以降，本稿では特に指定しない限りスコア付き辞書ソートのスコアとして濃度の逆数を用いる．

アルゴリズム 2 に，行のスコア付き辞書ソートにおける行の比較関数の処理を示す．アルゴリズム中 x_{ij} は行列 X の (i, j) 成分をあらわす．この比較関数を汎用のソートアルゴリズム内でつかうことで，行列の行をスコア付き辞書ソートすることができる．なお，アルゴリズム 2 では比較関数内でスコア計算をおこなっているが，この処理は事前に一度おこなえばよいため，比較関数の外に出してもよい．

Algorithm 2: スコア付き辞書ソート法の行比較関数

Input: $X \in \mathbb{R}^{m \times n}$, 行列; $p \in \{1, ..., m\}$, 比較対象行番号 1;
  $q \in \{1, ..., m\}$, 比較対象行番号 2;
Output: 行列 X における p 番目の行と q 番目の行の大小関係
Data: Q, 優先度付きキュー

/* 全ての列についてスコアを計算 */
foreach $k \in \{1, ..., n\}$ **do**
  Enqueue(Q, k, ComputeScore(X, k)) /* 行列 X の k 番目の列に対しスコアを算出し，スコアを優先度としキューに追加 */
/* スコアの高い列から順に要素を比較 */
while Q *is not empty* **do**
  $k \leftarrow$ Dequeue(Q)
  if $x_{pk} > x_{qk}$ **then**
    return *1* /* 行列 X の p 番目の行 $>$ q 番目の行 */
  else if $x_{pk} < x_{qk}$ **then**
    return *-1* /* 行列 X の p 番目の行 $<$ q 番目の行 */
return *0* /* 行列 X の p 番目の行 $=$ q 番目の行 */

表 1　圧縮率を向上させる並び替え方式の比較

並び替え方式	計算量	並び替え結果
全探索法	$O(m!\, n)$	最適解
貪欲法	$O(m^3\, n)$	実行可能解
スコア付き辞書ソート法（提案方式）	$O(n\, m \log m)$	実行可能解

スコア付き辞書ソートの計算量について議論する．汎用のソートアルゴリズムは m 個のレコードを $O(m \log m)$ でソートできることが知られている [7]．汎用のソートにおける一度の比較で，スコア付き辞書ソートの比較関数（アルゴリズム 2）は n 回の比較をおこなう．また，濃度をスコアとして利用するスコア計算には $O(nm)$ を要する．そのため，スコア付き辞書ソートの計算量は $O(n\, m \log m)$ となる．これは行列の行数 m に対して $m \log m$ のオーダーであり，巨大な行列にも十分適用可能である．

3.6　並び替え方式の特性

表 3 に，本稿で議論した三つの並び替え方式の特性をまとめる．提案するスコア付き辞書ソート法は，三つの並び替え方式のなかで最も計算量が小さく，巨大な行列に対しても十分適用が可能である．スコア付き辞書ソート法による圧縮率の向上効果は，5. 節で実データを用いた実験により確認する．

4.　代表値置換による不可逆圧縮

前節で，行列の行ないし列を並び替えることで行列の連長圧縮率を向上させる方式について述べた．本節では，行列内の要素を少数の代表値で置換することによりさらに連長圧縮率を向上させ，データサイズの減少と高速化の効果を高める方式について議論する．

本稿でこれまでに議論してきた行列の連長圧縮方式は，行列内に同じ値があまり登場しない場合，圧縮率が低い．例えば実数値を含む下記の行列 X は，同じ値を全く含まないため，連長圧縮による圧縮効果は得られない．

$$X = \begin{pmatrix} 62.5 \\ 63.0 \\ 64.5 \\ 72.5 \\ 64.7 \end{pmatrix} \tag{13}$$

この問題に対し，**MOARLE** は行列内の似た値をグループにまとめ，同一グループに属す値をグループの代表値により置き換える**代表値置換処理** (*representative replacement*) をおこなう．代表値置換処理は一種の不可逆圧縮であり，計算の精度を下げる一方で，さらなる圧縮効果を得る．本節の残りでは **MOARLE** の備える具体的な代表値置換方式二つについて説明する．

4.1 離散化による代表値置換

まず，離散化による代表値置換方式について説明する．この方式は，行列の各列について，格納されている値の値域をある基準点 β とある幅 δ で区切る．すなわち値域を $[\beta,\beta+\delta),[\beta+\delta,\beta+2\delta),[\beta+2\delta,\beta+3\delta),\ldots$ という区間に分割する．各区間が一つのグループとなる．そして，列内の値がそれぞれどの区間に属すかを算出し，同じ区間に属す値を区間の代表値により置き換える．代表値の例としては区間内の要素についての平均値がある．例えば式 13 の行列 X の一列目に対して $\beta = 0, \delta = 5$ で離散化と平均値による代表値置換をおこなうことを考える．このとき $62.5, 63.0, 64.5, 64.7$ は区間 $[60, 65)$ に，72.5 は区間 $[70, 75)$ に入り，それぞれの区間に対する平均値は $(62.5 + 63.0 + 64.5 + 64.7)/4 = 63.675$ と 72.5 となる．そのため，行列 X の第一列目を代表値置換した行列は

$$\widehat{X} = \begin{pmatrix} 63.675 \\ 63.675 \\ 63.675 \\ 72.5 \\ 63.675 \end{pmatrix} \tag{14}$$

となり，連長圧縮の置換前と比べ効果が高くなる．また，代表値置換後の行列 $\widehat{X}$ に行の並び替えをおこなうことでさらに連長圧縮率を高めることができる．

4.2 クラスタリングによる代表値置換

つぎに，クラスタリングによる代表値置換方式について説明する．この方式は，行列の各列について，格納されている値をクラスタリングする．各クラスタがグループとなる．そして，各クラスタの値を，クラスタの代表値により置き換える．例えば k-means 法 [13] をクラスタリングに用いる場合，各列の値があらかじめ設定されたパラメータ k の値に応じ，k 個の代表値へと置換される．クラスタリングによる代表値置換の利点には，値の値域を考慮する必要がないというものがある．これに対し，4.1 節で述べた離散化による代表値置換方式では，列ごとに値域を考慮して適切にパラメータ β, δ の値を設定する必要がある．

4.3 代表値置換による圧縮率の向上と演算誤差

本稿で述べた二つの代表値置換方式では，パラメータの値を変化させることで，行列の圧縮効果を左右することができる．このとき一般に，代表値置換のパラメータを変化させて圧縮率を高めるほど，代表値置換後の行列に対する演算結果と代表値置換前の元行列に対する演算結果の間に生じる演算の誤差は大きくなる．このトレードオフ関係に対しては，代表値置換方式と演算の種類によっては，パラメータの値から代表値置換により生じうる演算誤差の上限値と下限値を見積もる，ないしその逆をおこなうことが可能である．これによって，許容する誤差の上限値を与えたうえで行列の圧縮率を最大にし，省メモリ化・高速化の効果を最大限に高めたうえで計算をおこなうことができる．なお，演算誤差の見積もりの実装と評価は今後の課題とする．

5. 評 価 実 験

本節では提案する行列圧縮計算フレームワーク **MOARLE** の有効性を確かめるため，実データを用いた評価実験をおこなう．

5.1 実 験 環 境

システム: 実験に用いた計算機は，プロセッサが Intel Core i5 680 (4 コア，3.60GHz)，メモリが 16GB，OS が Linux 3.8.0 である．**MOARLE** の実装は C++言語でおこない，コンパイラとしては GNU g++ 4.7.3 を用いた．

データセット: 実験では，行列の圧縮計算対象として疎行列と密行列の両方を用いた．疎行列としては [3] で公開されている Bag of Words 形式の文書行列を用いた．密行列としては [11] で公開されている行列を用いた．各行列の情報を表 2 に記載する．

表 2　実験に用いたデータセット

	Sparse			Dense		
	nips	kos	enron	madelon	arcene	gisette
行数	12,419	6,906	28,102	500	10,000	5,000
列数	1,500	3,430	39,861	2,000	100	6,000
非零要素数	746,316	353,160	3,710,420	999,999	540,941	3,891,362

5.2 行列データサイズの圧縮率

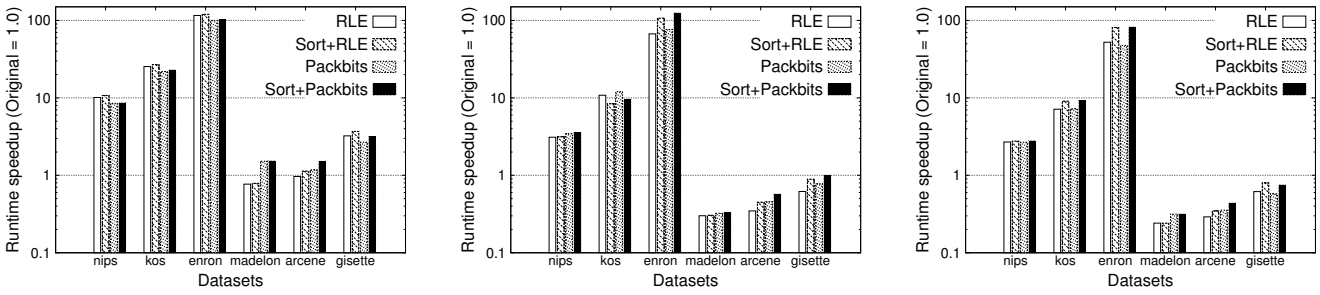
まず，**MOARLE** 圧縮部の行列データサイズの削減効果を評価する．実験では，2.1 節で述べた行列の行に関する連長圧縮を各データセット行列について適用し，圧縮前と比べた圧縮後の行列のデータサイズの比率（単位: %），すなわち圧縮率を比較した．連長圧縮方式には単純な連長圧縮（RLE）と改良方式である PackBits の二つを用いた．また，それぞれの方式について，データセットをそのまま圧縮した場合と，3.5 節で提案したスコア付き辞書ソート法でデータセット行列の列を並び替えたあとに圧縮した場合との計測をおこなった．

表 3 に，各データセットに対して連長圧縮を適用した際のデータ圧縮率を示す．表中，RLE は単純な連長圧縮の利用，Sort は事前の行列並び替えを，それぞれあらわす．表 3 を見ると，以下のようなことが確認できる．

- **連長圧縮は疎行列に対して大きな効果を発揮する．**疎行列である enron に対して並び替えをおこない，PackBits により連長圧縮をした場合に，最高の圧縮率 1.21% を得ている．
- **密行列に対して連長圧縮をおこなうと，行列のデータサ**

表 3　疎行列・密行列に対するデータ圧縮率（単位: MB）

	Sparse			Dense		
	nips	kos	enron	madelon	arcene	gisette
Original	71.06	90.36	4273.12	3.81	3.81	114.44
RLE	10.54 (14.84%)	5.19 (5.74%)	54.95 (1.28%)	7.53 (197.42%)	5.96 (156.45%)	55.16 (48.20%)
Sort+RLE	9.96 (14.02%)	4.89 (5.41%)	53.46 (1.25%)	7.46 (195.67%)	5.12 (134.48%)	46.82 (40.91%)
PackBits	10.0 (14.07%)	5.07 (5.62%)	53.69 (1.25%)	3.86 (101.40%)	3.84 (100.90%)	49.92 (43.62%)
Sort+PackBits	9.12 (12.84%)	4.75 (5.26%)	51.71 (1.21%)	3.88 (101.79%)	3.20 (83.97%)	39.03 (34.10%)



(a) 二乗和計算の高速化比率　(b) 内積計算の高速化比率　(c) ユークリッド距離計算の高速化比率

図 1　計算の高速化比率（速度向上比率は対数）

イズが増えてしまうことがある．密行列である madelon に単純な連長圧縮をした場合，圧縮率は最悪の 197.42% となり，元行列と比べデータサイズが倍近くに膨れ上がってしまう．この問題は，PackBits の適用により緩和される（圧縮率が 101.40% となる）．

- **PackBits は密行列に対して大きな効果を発揮する．**madelon，arcene，gisette の三つで，RLE における圧縮率と比較し，PackBits の利用によりそれぞれ 96%，55%，4.58% データサイズが減少している．
- **並び替えは密行列に対して特に効果を発揮する．**gisette に PackBits を適用した場合，並び替えにより 9.5% データサイズが減少している．
- **並び替えと PackBits の組み合わせは広範に効果的である．**madelon 以外の全てのデータセットで行列のデータサイズを削減することができている．

以上の結果から，スコア付き辞書ソートによる行列の並び替えと PackBits の利用により多様な行列に対して行列の圧縮効果が得られるといえる．

5.3　行列計算の高速化

まず，**MOARLE** による計算高速化の効果を評価する．実験では，行列内の各行ベクトルに関する二乗和計算，行ベクトル同士の内積計算，行ベクトル同士のユークリッド距離計算について，計算に要した時間を各圧縮方式について計測した．実験結果を図 1 に示す．図 1 は，圧縮をおこなわない通常の計算方式と比較した際の，各圧縮方式（RLE, Sort+RLE, PackBits, Sort+PackBits）による圧縮計算の計算時間の向上度合いをあらわす．

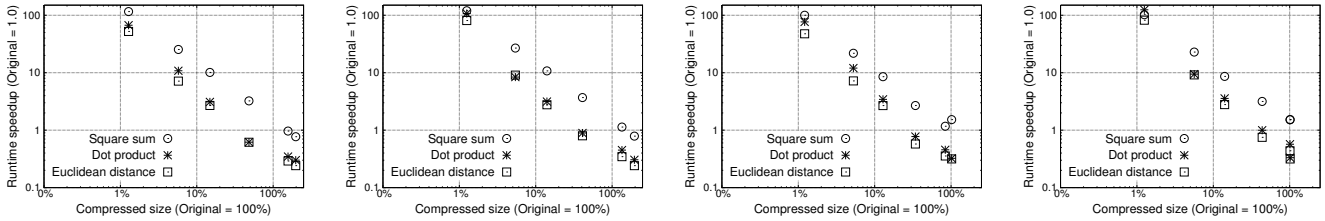
図 1(a) は，二乗和計算の高速化をあらす．疎行列に対しては 10 倍から 120 倍の高速化を達成して，ソートによる速度の向上もみられる．しかし，RLE と比べ PackBits ではデータ圧縮率こそ上がっているものの，高速化比率が下がってしまっている．これは，PackBits における圧縮データ列のスキャン時に長さが正か負か確認し条件分岐をおこなう処理があり，その処理がオーバーヘッドとなったためと推測される．密行列に対しては，単純な RLE を用いた場合に圧縮計算をおこなわない場合よりも速度が遅くなりうることが確認できる．この問題は PackBits と Sort の利用により解決され，最終的に各密行列で 1.5 倍から 3.7 倍の高速化を達成している．

図 1(b) は内積計算の高速化をあらわす．疎行列に対しては 3.5 倍から 124 倍の高速化を達成しており，二乗和計算と同様にソートによる速度の向上もみられる．また，二乗和計算とは異なり PackBits を利用した場合に RLE と比べ速度向上がみられる．これは，内積計算の圧縮計算実装が二乗和計算と比べ複雑であり，長さの正負を確認する処理が支配的ではないためと推測される．密行列に対しては，圧縮をおこなわない場合よりも速度が遅くなってしまっている．これは，現在の圧縮計算実装が複雑で最適化されておらず，実装面でのオーバーヘッドが大きいためと推測される．今後，実装を最適化したうえで，比較を再びおこなうことを予定している．

図 1(c) はユークリッド距離計算の高速化をあらわす．ユークリッド距離の圧縮計算は二乗和計算と内積計算の組み合わせであり，二乗和計算と内積計算の間に位置する実験結果となっているため，考察は省略する．

5.4　圧縮率と高速化比率の関係

最後に，圧縮率と高速化比率に関する考察をおこなう．5.2 節と 5.3 節で，各圧縮方式のデータ圧縮効果と高速化比率について述べた．図 2 は，これら効果の関係をあきらかにするため，各圧縮方式についてデータセットを圧縮した際の圧縮率とその



(a) RLE (b) Sort+RLE (c) PackBits (d) Sort+PackBits

図 2　各圧縮方式における圧縮率と高速化比率の関係（両対数グラフ）

際に得られた高速化比率を散布図にしたものである．各サンプル点がデータセットに対応する．図 2 は両対数グラフであり，全ての圧縮方式・全ての演算でデータセットの圧縮率と高速化比率の間にはベキ分布の関係が成り立つことをあらわしている．すなわち，圧縮後のサイズを線形に減少させると，高速化比率は指数関数的に向上する．この結果を利用すると，**MOARLE** 内でデータ圧縮率から高速化効果を見積もり，高速化が期待できない場合は通常の演算処理をおこなうことが可能となる．その実装と評価に関しては今後の課題とする．

6. 関 連 研 究

MOARLE は，疎行列表現・疎ベクトル表現を利用した行列・ベクトル演算ライブラリに関連する．ベクトル演算ライブラリである Eigen [10] は，疎行列や疎ベクトルをコンパクトに表現し，省メモリかつ高速に処理できる．疎なデータを対象とする Eigen に対し，本稿で提案した **MOARLE** は密なデータに対しても圧縮をおこない，圧縮計算効果を得ることができる．

Rendle らは，行列内のブロック構造を利用することで機械学習アルゴリズムを高速化する方式を提案している [14]．彼らの提案方式は関係代数におけるリレーション結合により生成される構造を利用するものであり，想定する構造がない場合には効果が得られない．これに対し **MOARLE** は行列の構造に対する想定が少なく，より多くの種類の行列に対して効果を得ることができる．

行列の行並び替えによる連長圧縮率の向上に関しては，オートマトンの状態遷移表の圧縮方式に類似した試みがある [6]．彼らは，本稿で述べた貪欲法に相当する方式で連長圧縮の向上を試みている．彼らの対象としている行列は状態遷移表であり，行列が巨大ではないため貪欲法が有効である．一方，本稿のようにデータ分析処理を対象とした場合は行列が巨大となり，計算量が $O(m^3 n)$ である貪欲法が適用できなくなることもある．**MOARLE** におけるスコア付き辞書ソート法の計算量は $O(n\,m \log m)$ であり，巨大な行列にも適用可能である．

4.3 節で述べた代表値置換による行列演算誤差の見積もりに関する試みとしては，BlinkDB [1] がある．BlinkDB は問合せ処理内でサンプリングをおこない近似結果を算出することで，問合せへの応答時間を短縮する．BlinkDB は問合せへの応答時間と近似による誤差のトレードオフ関係をシステマティックに扱い，ユーザの要求する応答時間から，結果に生じる最大誤差を算出することができる．

7. 結　　論

本稿で我々は，行列の圧縮計算フレームワーク **MOARLE** を提案した．**MOARLE** は事前に行列を連長圧縮したうえで，その後何度もおこなわれる計算を連長圧縮情報の利用により高速かつ省メモリにおこなう．**MOARLE** は，行列の圧縮部と圧縮情報を利用した計算部からなる．圧縮部に対して，我々は行列に対する演算の種類によっては行ないし列の順序が演算結果に影響をおよぼさないことに着目し，行ないし列の入れ替えをおこなうことで連長圧縮の効果を向上させる方式を提案した．また，行列内の要素を少数の代表値で置換することによってさらに連長圧縮の効果を向上させる方式も提案した．計算部に関しては，連長圧縮処理によりベクトル内で同一の値が連続するという事前知識が得られることに着目し，その知識を利用して計算回数を減らす方式を提案した．そして，実験により **MOARLE** が疎なデータに対して処理速度を最大で数十から数百倍高速化し，密なデータに対して数倍高速化することを示した．

文　　献

[1] S. Agarwal *et al*. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *EuroSys*, pages 29–42, New York, NY, USA, 2013. ACM.

[2] Apple Inc. Apple Technical Note TN1023, 1996.

[3] K. Bache, M. Lichman. UCI Machine Learning Repository, 2013.

[4] J. Bergstra, Y. Bengio. Random Search for Hyper-parameter Optimization. *J. Mach. Learn. Res.*, 13:281–305, February 2012.

[5] L. Bottou. Large-Scale Machine Learning with Stochastic Gradient Descent. In *COMPSTAT*, pages 177–187, Paris, France, August 2010. Springer.

[6] B. C. Brodie, D. E. Taylor, R. K. Cytron. A Scalable Architecture for High-Throughput Regular-Expression Pattern Matching. In *ISCA*, pages 191–202. IEEE Computer Society, 2006.

[7] T. H. Cormen *et al*. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[8] X. Feng *et al*. Towards a unified architecture for in-RDBMS analytics. In *SIGMOD*, pages 325–336. ACM, 2012.

[9] W. J. Fu. Penalized Regressions: The Bridge versus the Lasso. *Journal of Computational and Graphical Statistics*, 7(3):397–416, 1998.

[10] G. Guennebaud, B. Jacob et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[11] I. Guyon *et al*. Result Analysis of the NIPS 2003 Feature Selection Challenge. In *NIPS*, 2004.

[12] D. J. C. MacKay. *Information Theory, Inference & Learning Algorithms*. Cambridge University Press, New York, NY, USA, 2002.

[13] J. Macqueen. Some Methods for Classification and Analysis of Multivariate Observations. In *5-th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[14] S. Rendle. Scaling Factorization Machines to Relational Data. *PVLDB*, 6(5):337–348, 2013.