PyDAAL 超入門: パート 1 データ構造

この記事は、インテル® デベロッパー・ゾーンに公開されている「Gentle Introduction to PyDAAL: Vol 1 Data Structures」の日本語参考訳です。

インテル® データ・アナリティクス・アクセラレーション・ライブラリー (インテル® DAAL) は、インテル® アーキテクチャー向けに最適化されたビルディング・ブロックであり、すべてのデータ解析段階をサポートします。インテル® DAAL は、データの取得、前処理、変換、データマイニング、モデリング、検証、データに基づく意思決定を含む、データ解析のすべての段階をカバーします。Python* ユーザーは、インテル® DAAL の Python* API (PyDAAL) を使用して、これらの基本的な機能にアクセスできます。簡単なスクリプト API を介してアクセスされる PyDAAL により、Python* のマシンラーニングは大きく向上します。PyDAAL は、Python* スクリプトのバッチ解析をオンライン (ストリーミング) データ取得や分散演算処理に容易に拡張できる独自の機能も備えています。さまざまなインテル® プロセッサーで最高のパフォーマンスが得られるように、インテル® DAAL はインテル® マス・カーネル・ライブラリー (インテル® MKL) およびインテル® インテグレーテッド・パフォーマンス・プリミティブ (インテル® IPP) の最適化アルゴリズムを利用しています。インテル® DAAL には、C++、Java*、および Python* 向けの API が用意されています。この超入門シリーズでは、PyDAAL の基礎を詳しく説明します。この最初のパートでは、インテル® DAAL のカスタムデータ構造、数値テーブル、PyDAAL におけるデータ管理についてカバーします。

超入門シリーズ

- **パート 1: データ構造** - インテル® DAAL のデータ管理コンポーネントとカスタムデータ構造 (数値テーブルとデータ辞書) をサンプルコードを使用して紹介します。
- **パート 2: 数値テーブルの基本操作** - インテル® DAAL のカスタムデータ構造 (数値テーブルとデータ辞書) で実行可能な操作について、サンプルコードを使用して紹介します。
- **パート 3: 解析モデルの構築とデプロイメント** - バッチ処理でシリアル化されたデプロイメントを利用するインテル® DAAL の解析モデリングと評価プロセスを紹介します。
- **パート 4: 分散処理とオンライン処理** - 大規模なデータやストリーミング・データのデータ分析やモデル・フィッティングをサポートするインテル® DAAL の高度な処理モード (分散とオンライン) を紹介します。

インテル® Distribution for Python* とインテル® DAAL のインストール

この記事のデモには、インテル® Distribution for Python* およびインテル® DAAL (Anaconda* Cloud から無料で入手可能) が必要です (インテル® Parallel Studio XE 2019 ではパッケージに同梱されています)。

1. 必要なパッケージをインストールするため、インテル® Distribution for Python* の完全な環境をインストールします。

   ```
   conda create -n IDP -c intel intelpython3_full python=3.6
   ```

2. インテル® Distribution for Python* 環境をアクティベートします。

   ```
   source activate IDP
   ```

   または

   ```
   activate IDP
   ```

詳細は、インストール・オプションとインテルのパッケージの一覧 (英語) を参照してください。

1. PyDAAL のデータ管理

インテル® DAAL は、次のデータ処理方法をサポートしています。

- バッチ処理
- オンライン処理
- 分散処理
- 複合処理 (オンライン処理と分散処理の組み合わせ)

この記事では、主に**バッチ処理**に注目します。オンラインおよび分散データ管理は、超入門シリーズのパート 4 で説明します。

プログラミングの注意事項

強い型付け: Python* のスクリプトは、動的型付け (ダックタイピング) の概念を多用し、実行時に型を推論する Python* のインタープリターに依存しています。しかし、メモリー・フットプリントに注意が必要な場合や混在コードを配備する場合、問題が発生することがあります。PyDAAL API は、C++ およびアセンブリー言語で記述されたライブラリーを呼び出すため、ユーザーは混在コード環境を利用することになります。そのため、PyDAAL で一貫した型付けが必要になることから、NumPy* のデータ型 "np.float32"、"np.float64"、

"np.intc" をサポートしています。静的/強い型付けにより、適切なメモリー管理のために明示的なデータ型の宣言を行えるだけでなく、コンパイル中に型チェックを行うため、実行時間が大幅に短縮されます。

メモリー・アクセス・パターン: 多次元配列はメモリーに連続するデータとして格納されます。これらのメモリーセグメントは、配列要素を整列するために使用できます。これらの要素を格納する 1 つの方法は、行ベクトルを連続して格納する "行優先" 方式です。列を連続して格納する "列優先" 方式もあります。これらの 2 つのデータ・レイアウト・パターンは、どちらもインテル® DAAL の数値データ構造でサポートされており、記述されているプログラムの想定されるメモリー・アクセス・パターンに基づいて選択します。前者は、C プログラミングおよびインテル® DAAL の標準数値テーブルのデフォルトです。後者は Fortran プログラミングのデフォルトで、インテル® DAAL の配列構造体 (SOA) 数値テーブルを使用します。

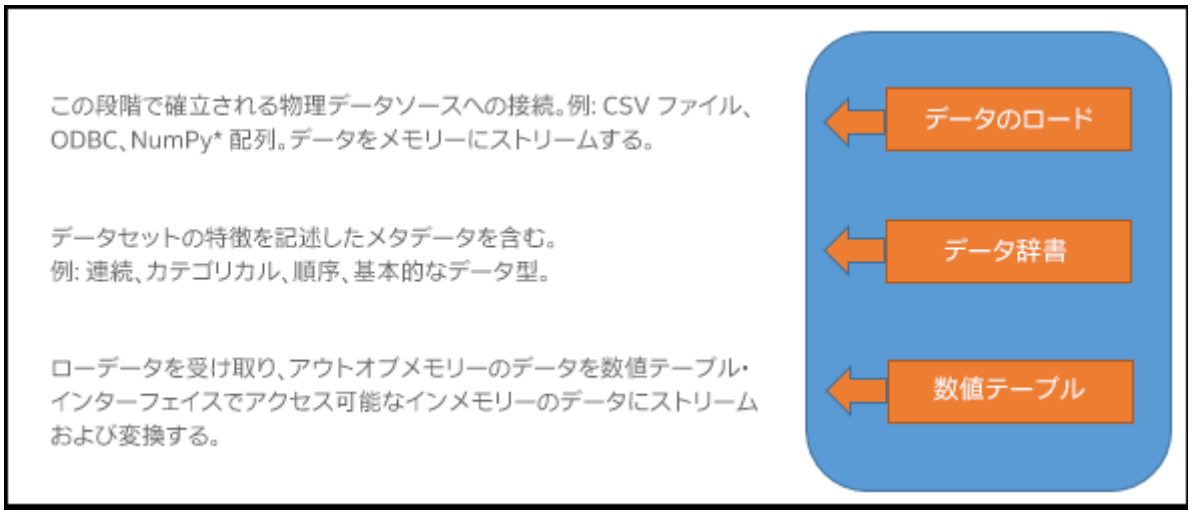
NumPy* には、NumPy* 配列をメモリーの列優先ストレージに変換する **ascontiguousarray ()** メソッドが用意されています。PyDAAL は、入力行列が連続するように自動的に変換を試みます。

SWIG インターフェイス・オブジェクト: インテル® DAAL の重要なコンポーネントである SWIG は、C/C++ プログラム向けの単純化されたラッパーおよびインターフェイス・ジェネレーターです。(ウィキペディアの SWIG ページ)。PyDAAL は、SWIG を使用して Python* スクリプトでインテル® DAAL C++ ライブラリーを制御します。重要な点は、SWIG を使用することにより、PyDAAL で Python* のグローバル・インタープリター・ロック (GIL) を回避して、コンパイルされた C++ コードによる処理/スレッド化をディスパッチできることです。
インテルの PyDAAL API チームは、dir(DAAL_object) 呼び出しにより Python* の対話型コンソールに表示されるクラスメソッドとして、インテル® DAAL の C++ メンバー関数を Python* ユーザー向けに提供しています。

データ構造の概要とフロー

数値テーブル: インテル® DAAL により使用される主要データ構造は数値テーブルです。生データは数値テーブル構造にストリームされ、以降のアクセスで解析モデルおよび適切なマシン・ラーニング・アルゴリズムを構築するため、メモリーに格納されます。インテル® DAAL の数値テーブルは、行が観測を表し、列が特徴を表す、データセットのテーブルを定義します。数値テーブルのデータは、数値テーブル・インターフェイスを使用してアクセスします。

インテル® DAAL のコンポーネントとデータフロー



2. 数値テーブル

2.1 数値テーブルの種類

数値テーブルは基本データ型およびストレージ設定で構築できます。初期化設定はデータ型とデータレイアウトに分けることができます。

- データ型: インテル® DAAL は、数値テーブルの作成に intc (C の整数型 int32/64 と同じ)、float32、float64 の 3 つのデータ型 (dtype) をサポートします。

NumPy* の dtype は "np.float32"、"np.float64"、"np.intc" と呼ばれます。

- データレイアウト: 密行列データは、データパターンに応じて、行/列優先アクセスにレイアウトできます。また、疎行列および三角行列を使用して、メモリー・フットプリントの小さなデータを作成できます。

インテル® DAAL がサポートする数値テーブルの種類 (密および疎データの同次、非同次、メモリー節約数値テーブル) を以下に示します。

1. 同次 nT

- 特徴は同じ基本データ型で要素を連続して格納
- 利用可能なデータ型: intc、float32、float64

クラス名:

HomogenNumericTable(array, ntype)

`array` - 連続する NumPy* 配列

`ntype` - インテル® DAAL がサポートしているデータ型

例: `HomogenNumericTable (inputArray, ntype = np.float32)`

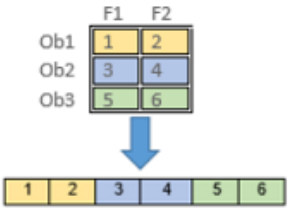
`inputArray` - 宣言されたデータ型で NumPy* を使用して作成された配列

2. 非同次 nT

- 特徴は異なるデータ型

1. **構造体配列 (AOS):** "観測" に沿って格納

- サンプルデータ: F1、F2 は特徴、Ob1、Ob2、Ob3 は観測



NumPy* 配列 (デフォルト) に似た方法でメモリーを使用して要素を格納

クラス名:

AOSNumericTable*(array)*

array - 1D 構造体配列

例: `AOSNumericTable(inputArray)`

`inputArray` - NumPy* 配列を使用して作成された非同次配列

3. メモリー節約 nT

- 特徴は同じ基本データ型
- 疎/三角行列の種類に応じてメモリー格納を最適化

1. パックド三角行列

クラス名:

上パックド float64 データ型行列を作成するには

```
PackedTriangularMatrix(packedLayout =
NumericTableIface.upperPackedTriangularMatrix,
DataType = float64)
```

```
Prereqs: from numpy import float64
from daal.data_management import  NumericTableIface
```

2. パックド対称行列

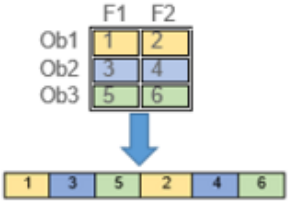
クラス名:

```
PackedSymmetricMatrix(packedLayout =
NumericTableIface.upperPackedSymmetricMatrix,
DataType = float64)
```

2. 非同次 nT

2. **配列構造体 (SOA):** "特徴" に沿って格納

サンプルデータ: F1、F2 は特徴、Ob1、Ob2、Ob3 は観測



pandas* DataFrame に似た方法でメモリーを使用して要素を格納

クラス名:

SOANumericTable*(nFeatures, nObservations)*

`nFeatures`: 特徴の数
`nObservations`: 観測の数
メソッド `setArray(columns, index)`

例: `SOA = SOANumericTable(nFeatures, nObservations)`

```
inputArray の col:
    SOA.setArray(col, idx)
```

3. メモリー節約 nT

3. **圧縮疎行列 (CSR)**

クラス名:

CSRNumericTable*(values, colIndices, rowOffsets, nFeatures, nObservations)*

`values`: NumPy* 1D 配列
`colIndices`: 行インデックスの NumPy* 配列
`rowOffsets`: 列インデックスの NumPy* 配列
`nFeatures`: 特徴の数
`nObservations`: 観測の数

2.2 数値テーブル初期化リファレンス

同次数値テーブル

同次数値テーブル

dtype	クラス名	エイリアス
intc	HomogenNumericTable(ndarray, ntype = intc)	HomogenNumericTable_Intc(numpy_array))
float32	HomogenNumericTable(ndarray, ntype = float32)	HomogenNumericTable_Float32(numpy_array))
float64	HomogenNumericTable(ndarray, ntype = float64)	HomogenNumericTable_Float64(numpy_array))

非同次数値テーブル

1. 構造体配列: heterogen_AOS_nT = AOSNumericTable(ndarray)
2. 配列構造体: heterogen_SOA_nT = SOANumericTable(nRows, nColumns)

メモリー節約数値テーブル

1. 圧縮疎行列
homogenCSR_nT = CSRNumericTable(values, colIndices, rowOffsets, nFeatures, nObservations)
2. パックド行列

2.3 さまざまな種類の数値テーブルへのデータのロード

前述したように、PyDAAL は強い型付けのライブラリーです。インテル® DAAL は、一般的な (同次) 型のデータを効率的に制御し、混在 (非同次) 型のデータを個別に制御します。そのため、インテル® DAAL の数値テーブルには、両方の型条件でデータを格納および供給する複数のバージョン、および疎行列のメモリー節約バージョンが用意されています。

2.3.1 同次数値テーブルと異なるデータのロード方法

同次数値テーブルは、同じ基本データ型の特徴を格納するインテル® DAAL のデータ構造です。特徴の値は、(観測 1、観測 2、... のように) 行優先で隣接するブロックとしてメモリーでレイアウトされます。数値テーブルの作成中に、インテル® DAAL は、特徴タイプ (カテゴリカル、順序、連続) の割り当てを格納するデータ辞書を生成します。データ辞書は、いつでもアクセスして割り当てを変更できます。以下のコード例は、inputArray として *NumPy** 配列、*pandas* DataFrame* および PyDAAL の *FileDataSource* (csv ロード) クラスを使用した数値テーブルの作成を示しています。

i. NumPy* 配列を利用したデータのロードと数値テーブルの作成

PyDAAL は、NumPy* の直接的で容易な統合をサポートします。以下のコード例は、NumPy* 配列から HomogenNumericTable を作成します。

数値テーブルは、intc、float32、float64 の 3 つの dtype をサポートしています。dtype を宣言しないで整数 NumPy* 配列が作成された場合、Python* は dtype を推論して、C と同じ int32/int64 ではなく int32 を整数のデフォルトにします。そのため、整数型数値テーブルを作成する場合、入力 NumPy* 配列の初期化で dtype (np.intc) を宣言しなければなりません。

NumPy* ndarray から同次 nT を作成するステップ

1. 宣言された dtype を含む NumPy* ndarray を作成します。
   `array = np.array([], dtype=type)`

   インテル® DAAL は、データ型 np.float64、np.float32、np.intc をサポートしています。

2. 作成した NumPy* 配列から nT を作成します。
   `nT = HomogenNumericTable( array , ntype=dtype)`

コード例

```
import numpy as np

array = np.array([[0.5, -1.3],
                  [2.5, -3.3],
                  [4.5, -5.3],
                  [6.5, -7.3],
                  [8.5, -9.3]],
                   dtype = np.float32)

# import Available Modules for Homogen numeric table
from daal.data_management import(HomogenNumericTable)

nT = HomogenNumericTable(array, ntype = np.float32)
```

ii. pandas* DataFrame を利用したデータのロードと数値テーブルの作成

pandas* は、スプレッドシート形式でデータセットを準備および操作するため広く利用されているライブラリーです。その使いやすさと汎用性により、Python* マシンラーニング作業のあらゆる場所でライブラリーを利用できます。PyDAAL は、同次 (NumPy* 配列を介して入力) または非同次 (DataFrame から直接入力) の両方で、pandas* DataFrame からのデータ入力をサポートしています。非同次数値テーブル作成の詳細は、「SOA」セクションを参照してください。

pandas* DataFrame から同次 nT を作成するステップ

1. 宣言された dtype を含む pandas* DataFrame を作成します。
   `df = pd.DataFrame(values, dtype=type)`

   インテル® DAAL は、データ型 np.float64、np.float32、np.intc をサポートしています。

2. pandas* df を Numpy* 配列 (ndarray) に変換します。
   `array = df.as_matrix()`
3. 作成した NumPy* 配列から nT を作成します。
   `nT = HomogenNumericTable(array , ntype=dtype)`

コード例

```
import pandas as pd
import numpy as np

#Initialize the columns with values
Col1 = [1,2,3,4,5]
Col2 = [6,7,8,9,10]
# Create a pandas DataFrame of dtype integer
df_int = pd.DataFrame({'Col1':Col1,
                       'Col2':Col2},
                      dtype=np.intc)

array = df_int.as_matrix()

from daal.data_management import(HomogenNumericTable)

nT  = HomogenNumericTable(array, ntype = np.intc)
```

iii. CSV ファイルを利用したデータのロードと数値テーブルの作成

pandas* により提供される優れた機能の 1 つは、CSV ファイルからデータを読み取り、マシン・ラーニング・アルゴリズムで使用される DataFrame にデータをロードすることです。インテル® DAAL には、pandas* が csv ファイルを読み取る方法に類似した "FileDataSource" クラスが用意されています。PyDAAL の FileDataSource は、空のデータ・ソース・オブジェクトを作成し、インテル® DAAL の CSVFeatureManager クラスを使用して csv ファイルから行のブロックをロードした後、getNumericTable() メソッドを使用して数値テーブルを作成します。現在、FileDataSource を使用する場合、float64 dtype のみ利用できます。

CSV ファイルソースから同次 nT を作成するステップ

1. csv パスを引数として使用して FileDataSource オブジェクトを作成し、数値テーブルにメモリーを割り当てて、データからデータ辞書を作成します。
   ```
   dataSource = FileDataSource(path, DataSource.doAllocateNumericTable,
   DataSource.doDictionaryFromContext)
   ```

   `DataSource.doAllocateNumericTable` は、デフォルトで同次数値テーブルを作成します (そのため、AOS 作成では "notAllocateNumericTable" を選択します)。

2. 必要なデータブロック (行) を FileDataSource オブジェクトにロードします。
   `dataSource.loadDataBlock(nRows)`
3. ロードされたデータを使用してデータソースから HomogenNumericTable を作成します。
   `nT = dataSource.getNumericTable()`
4. デフォルトの dtype は float64 です。

コード例

```
from daal.data_management \
import(FileDataSource, DataSource)

dataSource = FileDataSource(
    r'path', DataSource.doAllocateNumericTable,
DataSource.doDictionaryFromContext)

# boilerplate method to load nRows in the csv file, default loads all rows if
no argument passed
dataSource.loadDataBlock(30) # load first 30 rows

#dataSource.loadDataBlock() to load all rows

nT  = dataSource.getNumericTable()
```

2.3.2 非同次数値テーブルと異なるデータのロード方法

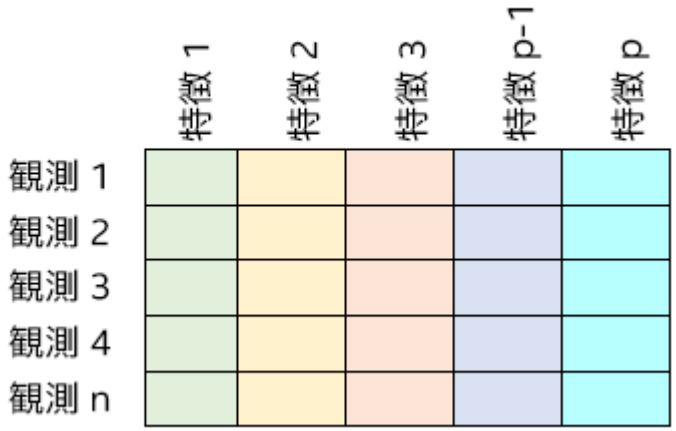
明示的に宣言されていない場合、Python* は動的に型付けされ、実行中に dtype を推論します。このダックタイピングにより、メモリー・フットプリントが考慮されなくなることがあります。データセットが大きくなり、メモリー使用量が重要な要件である場合、dtype の明示的な宣言 (C++ ライブラリーと接続している PyDAAL API によりサポートされている機能) が効果的になります。

入力データの列に異なる数値データ型 (intc、float、または double) が含まれる場合、メモリー・フットプリントを減らすには、各列で dtype を宣言する必要があります。インテル® DAAL の非同次数値テーブルは、配列の個々の列で dtype を宣言する機能により、静的型付けによるメモリー使用を節約します。PyDAAL の現在のバージョンでは、AOSNumericTable (構造体配列) と SOANumericTable (配列構造体) の 2 つの非同次構造を利用できます。

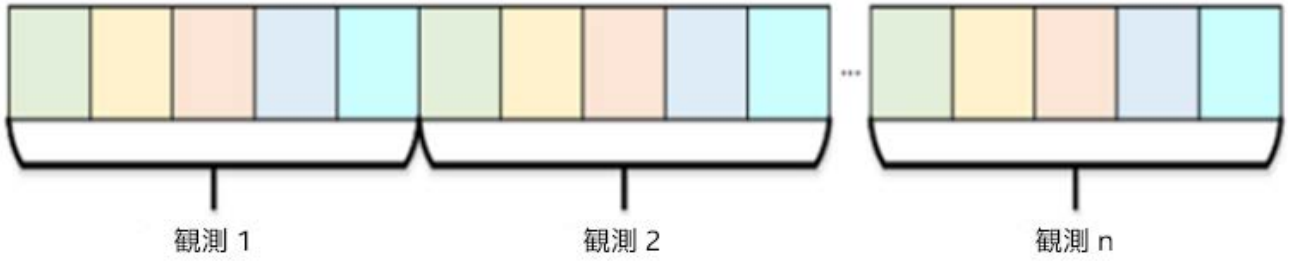
a. メモリー・レイアウト・パターンと AOSNumericTable/SOANumericTable

アクセスパターンに応じて、行優先または列優先形式でメモリーのデータセットのレイアウトを選択します。得られるデータ構造はそれぞれ、構造体配列 (AOS) および配列構造体 (SOA) と呼ばれます。ダウンストリーム・アクセスが行ごと (シーケンシャル観測) の場合は、AOS が最適な選択です。列ごとのアクセス (シーケンシャル特徴) を行う場合は、SOA が最適な選択です。以下の図は、AOS および SOA データレイアウトの違いを示したものです。

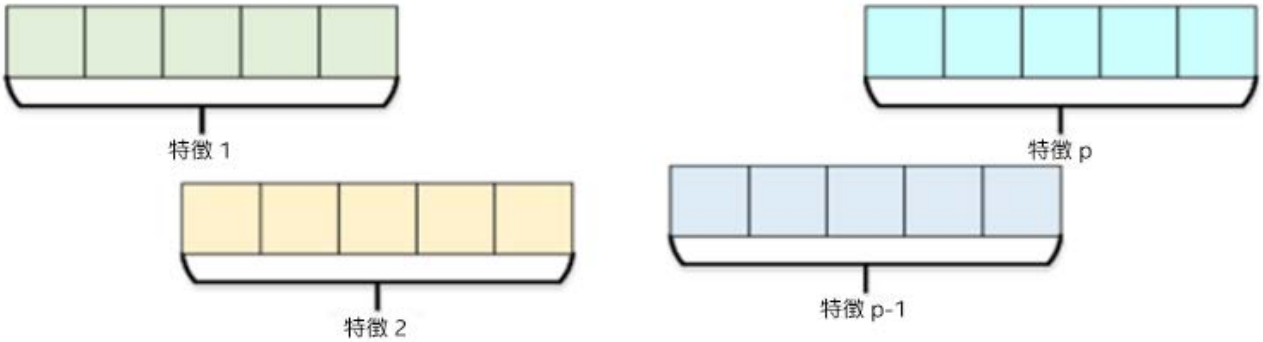
入力データの例



メモリーレイアウト: AOS (構造体配列)



メモリーレイアウト: SOA (配列構造体)



b. AOSNumericTable (構造体配列データ構造)

データアクセスが行優先で入力データ特徴が非同次 dtype を含む場合、インテル® DAAL の AOSNumericTable はアクセスを高速化するため対応するメモリーパターンを提供します。具体的には、"観測" に連続するメモリーを割り当てます。以下のコード例は、*NumPy** 配列、*pandas* DataFrame* および PyDAAL の *FileDataSource* クラスを使用した AOSNumericTable テーブルの作成を示しています。

i. NumPy* 配列を利用したデータのロードと数値テーブルの作成

インテル® DAAL の HomogenNumericTable と異なり、AOSNumericTable は要素のタプルと各タプルで宣言された dtype を含む 1D NumPy* 配列で作成されます。各タプルはデータセットの行です。生成される入力 NumPy* 配列の形状は (nRows,) になります。

NumPy* 配列から AOS nT を作成するステップ

1. 各列で、宣言された dtype を含む NumPy* 1D 配列を作成します。
   `array = np.array([], dtype=[(column, dtype)])`

   インテル® DAAL は、NumPy* データ型 np.float64、np.float32、np.intc をサポートしています。

2. NumPy* 配列から AOS nT を作成します。
   `nT = AOSNumericTable(array)`

コード例

```
import numpy as np

from daal.data_management
import(AOSNumericTable)

array = np.array([(0.5, -1.3, 1),
                  (4.5, -5.3, 2),
                  (6.5, -7.3, 0))],
                  dtype=[('x',                 np.float32),
                         ('categ',             np.intc),
                         ('value',             np.float64)])

nT  = AOSNumericTable(array)
```

ii. pandas* DataFrame を利用したデータのロードと数値テーブルの作成

pandas* DataFrame から AOS nT を作成するステップ

1. pandas* DataFrame を作成します。
   `df = pd.DataFrame(values)`

   インテル® DAAL は、NumPy* データ型 np.float64、np.float32、np.intc をサポートしています。

2. pandas* DataFrame を shape(nRows,) の 1D 構造化 NumPy* 配列に変換します。
   `get_StructArray()` ヘルパー関数を使用して以下の操作を行います。
   - タプルのリストを作成します。
   - dtype で zip します。
   - NumPy* 配列に変換します。

   `array = get_StructArray(df, [dtype1, dtype2, etc.])`

3. 構造化 NumPy* 配列から nT を作成します。
   `nT = ASONumericTable(np.1darray)`

コード例

```
import pandas as pd
import numpy as np
df = pd.DataFrame(columns=['c','f'])

df ['c']=[1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
df ['f']=[3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9, 4.0]

def get_StructArray(df,dtypes):
#*** inputs: df, [dtypes], output: structured Numpy array ***
    dataList = []
    for idx in range(df.shape[0]):
        dataList.append(tuple(df.loc[idx]))
    decDtype = list(zip(df.columns.tolist(),dtypes))
    array =  np.array(dataList,dtype = decDtype)
    return array

array = get_StructArray(df, [np.intc,np.float64] )

from daal.data_management import AOSNumericTable

nT  = AOSNumericTable(array)
```

iii. CSV ファイルを利用したデータのロードと数値テーブルの作成

CSV ファイルソースから AOS nT を作成するステップ

1. csv パスを引数として使用して FileDataSource オブジェクトを作成します。データからデータ辞書を作成します。対応する数値テーブルは割り当てません。
   `dataSource = FileDataSource(path, DataSource.notAllocateNumericTable, DataSource.doDictionaryFromContext)`

   DataSource.doAllocateNumericTable は、デフォルトで同次数値テーブルを作成します (そのため、AOS 作成では "notAllocateNumericTable" を選択します)。

2. 入力データソースのすべての列で、宣言された dtype を含む空の 1D NumPy* 配列を初期化します。
   `array = np.empty([nRows,], dtype=[(column,dtype)])`

   インテル® DAAL は、NumPy* データ型 np.intc、np.float32、np.float64 をサポートしています。

3. AOSNumericTable のメモリーブロックをステップ 2 で初期化した空の配列に割り当てます。
   `nT = AOSNumericTable(array)`
4. データブロック (行) を FileDataSource オブジェクトから AOSNumericTable レイアウトにロードします。
   `dataSource.loadDataBlock(nRows,nT)`

コード例

```
import numpy as np

from daal.data_management import (FileDataSource, AOSNumericTable,
DataSource)

dataSource = FileDataSource(
    r'path', DataSource.notAllocateNumericTable,
DataSource.doDictionaryFromContext)

array = np.empty([10,],dtype=[('x','i4'),('y','f8')])

nT = AOSNumericTable(array)

dataSource.loadDataBlock(10,nT)
```

c. SOANumericTable (配列構造体データ構造)

データアクセスが列優先で入力データ特徴が非同次 dtype を含む場合、インテル® DAAL の AOSNumericTable はアクセスを高速化するため対応するメモリーパターンを提供します。具体的には、"特徴" に連続するメモリーを割り当てます。pandas* はメモリーに同様のパターンでデータを格納するため、このデータ構造は pandas* DataFrame ではより自然な変換になります。

初期化の際に配列の値を設定できるインテル® DAAL の AOSNumericTable とは対照的に、SOANumericTable は初期化の際に最初に行と列の数を定義して (連続するブロックを割り当てて)、その後、一度に 1 つの列に配列の値を設定する必要があります。つまり、適切なデータ次元で SOANumericTable 構造を作成した後、テーブルにデータを設定する必要があります。

以下のコード例は、NumPy* 配列、pandas* DataFrame および PyDAAL の FileDataSource クラスを使用した SOANumericTable テーブルの作成を示しています。

i. NumPy* 配列を利用したデータのロードと数値テーブルの作成

NumPy* 配列から SOA nT を作成するステップ

1. すべての列で、宣言された dtype を含む NumPy* ndarray を作成します。
   `array = np.array([], dtype=type)`

   インテル® DAAL は、NumPy* データ型 np.float64、np.float32、np.intc をサポートしています。

2. nRows および nColumns で SOA nT テンプレートを作成します。
   `nT = SOANumericTable(nColumns, nRows)`
3. SOA nT 配列で一度に 1 つの列を設定します。
   `nT.setArray(array,column index)`

コード例

```
import numpy as np

from daal.data_management import SOANumericTable

Col1 = np.array([1.0, 1.2, 1.4, 1.6, 1.8, 2.0, 2.2, 2.4, 2.6,2.8],
dtype=np.float64)
Col2 = np.array([3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9, 4.0],
dtype=np.float32)
Col3 = np.array([-10, -20, -30, -40, -50, -60, -70, -80, -90, -100],
dtype=np.intc)
nObservations = 10
nFeatures = 4
nT  = SOANumericTable(nFeatures, nObservations)

nT.setArray(Col1, 0)
nT.setArray(Col2, 1)
nT.setArray(Col3, 2)
```

ii. pandas* DataFrame を利用したデータのロードと数値テーブルの作成

pandas* DataFrame から SOA nT を作成するステップ

1. 各列で、異なる dtype で pandas* DataFrame を作成します。
   `df = pd.DataFrame(values)`

   インテル® DAAL は、データ型 np.float64、np.float32、np.intc をサポートしています。

2. df の nRows および nColumns で SOA nT テンプレートを作成します。
   `nT = SOANumericTable(nColumns, nRows)`
3. df のすべての列を NumPy* 配列に変換 (df の列が 3 つの場合、3 つの NumPy* 配列に変換) して、NumPy* 配列で各 SOA nT 列を設定します。
   `nT.setArray(array,column index)`

コード例

```
import pandas as pd
import numpy as np

#Initialize the columns with values
from daal.data_management import SOANumericTable

df = pd.DataFrame()

df['a']=[1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
df['b']=[3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9, 4.0]
df['c']=[1.0, 1.2, 1.4, 1.6, 1.8, 2.0, 2.2, 2.4, 2.6, 2.8]

df = df.astype(dtype={'a' : np.intc,
                      'b' : np.float32,
                      'c' : np.float64})

nT  = SOANumericTable(df.shape[1],df.shape[0])
```

```
for idx in range(len(df.columns)):
        nT.setArray(df[df.columns[idx]].values,idx)
```

SOA nT も、行および列の数なしで初期化し、setNumberOfRows(N) および setNumberOfColumns(N) メソッドを使用して後の段階で数を設定できます。しかし、値を含む既存の SOA nT に行および列の数を設定すると、空の SOA nT が再作成され、前の値は削除されます。

iii. CSV ファイルを利用したデータのロードと数値テーブルの作成

CSV ファイルソースから SOA nT を作成するステップ

1. csv パスを引数として使用して FileDataSource オブジェクトを作成します。データからデータ辞書を作成します。対応する数値テーブルは割り当てません。
   `dataSource = FileDataSource(path, DataSource.notAllocateNumericTable, DataSource.doDictionaryFromContext)`

   `DataSource.doAllocateNumericTable` は、デフォルトで同次数値テーブルを作成します (そのため、SOA 作成では "notAllocateNumericTable" を選択します)。

2. 入力データソースのすべての列で、宣言された dtype を含む空の 1D NumPy* 配列を初期化します。
   `Col1_array = np.empty([nRows,],dtype=dtypes)`

   インテル® DAAL は、NumPy* データ型 np.intc、np.float32、np.float64 をサポートしています。

3. SOANumericTable のメモリーブロックに特徴および観測の数を割り当てます。
   `nT = SOANumericTable(nRows,nObservations)`
4. SOANumericTable のすべての列にステップ 2 で作成した空の 1D NumPy* 配列を設定します。
   `nT.setArray(array,idx)`
5. データブロック (行) を FileDataSource オブジェクトから SOANumericTable レイアウトにロードします。
   `dataSource.loadDataBlock(nRows,nT)`

コード例

```
from daal.data_management import(FileDataSource,SOANumericTable, DataSource)
import numpy as np

# CSV file 'path' with 10 rows and 2 columns
dataSource = FileDataSource(
    r'path', DataSource.notAllocateNumericTable,
DataSource.doDictionaryFromContext)

#if data source has 2 columns
Col1_array = np.empty([10,],dtype=np.float64)
Col2_array = np.empty([10,],dtype=np.float64)

nT  = SOANumericTable(2,10)

nT.setArray(Col1_array,0)
```

```
nT.setArray(Col2_array,1)

dataSource.loadDataBlock(10,nT)
```

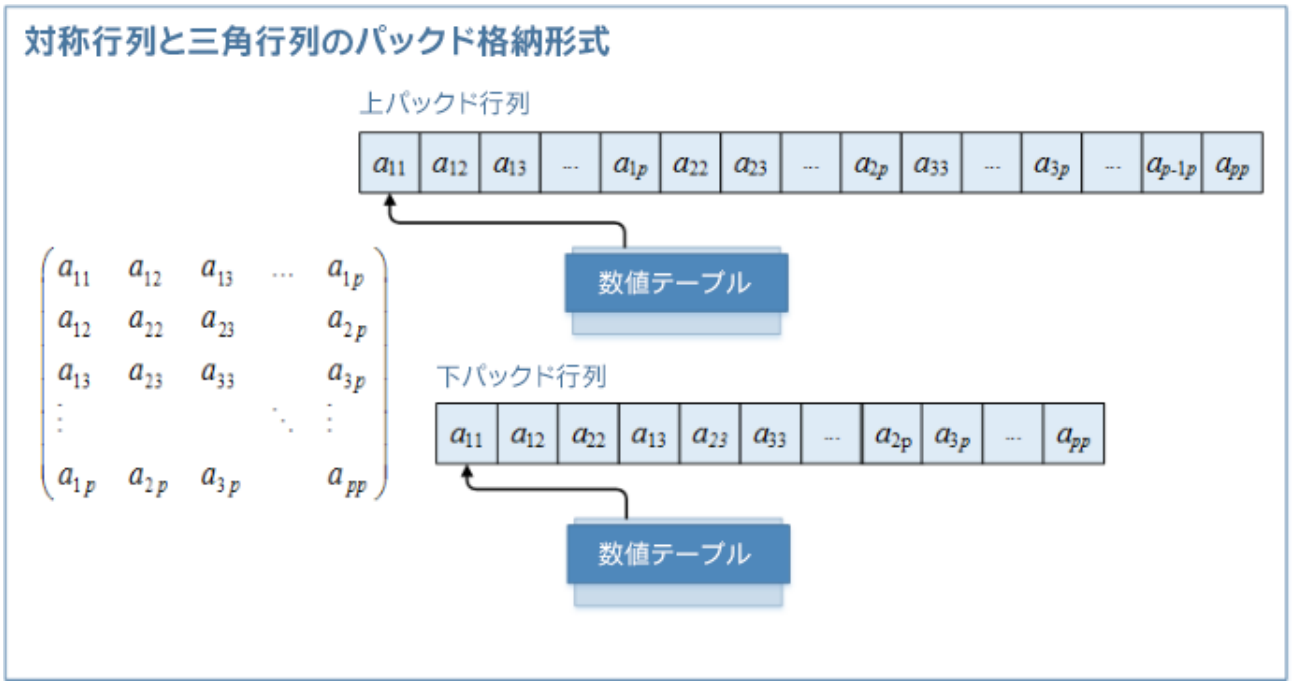
2.3.3 同次メモリー節約数値テーブルと異なるデータのロード方法

この記事の「同次」セクションで説明したように、同次 nT では、メモリー割り当ては同じデータ型の値を保持するすべての観測にわたる 1 つの連続するブロックです。同次メモリー節約 nT は、効率良く値を適切に格納する特別なクラスを表し、入力データ (行列/NumPy* 配列) が疎/疎に近い場合にメモリー・フットプリントを縮小します。

同次メモリー節約 nT は、パックド行列および圧縮疎行列を表す特別なデータレイアウトのクラスを提供します。メモリー節約数値テーブルは、プリミティブ・データ・レイアウトを保持したまま、メモリー・フットプリントの非効率性を除去して、疎/疎に近い/対称行列を格納します。後のセクションで説明する行列の種類の 1 つに基づいたエントリーを提供するため、数値テーブルの値はメモリーに格納されます。

同次メモリー節約数値テーブルは、3 種類の行列を扱うクラスを表します。

a. 対称および三角数値テーブル



対称行列: 対角線の上/下の値が対称の行列。行列の対角線の上と下の両方の値が対称であるため、行列全体を保存するとほぼ倍になります。メモリー節約数値テーブルは、冗長なストレージおよびメモリー・フットプリントを減らすため、対角線の上/下部分のいずれかを保存するオプションを提供します。数値テーブルは、ユーザー設定のレイアウトに応じて上/下パックド対称行列になるように入力対称行列を格納します。

三角行列: 対角線の上/下の値が 0 の行列。通常の同次数値テーブルを作成しているときに、ゼロの上/下を対角線に保存することはメモリーの効率的な利用ではありません。パックド三角行列数値テーブルは、三角行列の種類に応じて上/下の値を対角線に格納することでメモリー・フットプリントを減らします。数値テーブルは、上/下三角行列表現を許可する入力パックド三角行列を格納します。

以下のコード例は、*NumPy** 配列、*pandas* DataFrame* および PyDAAL の *FileDataSource* クラスを使用したパックド三角行列およびパックド対称行列テーブルの作成を示しています。

i. NumPy* 配列を利用したデータのロードと数値テーブルの作成

NumPy* 配列からパックド nT を作成するステップ

1. 宣言された dtype を含むパックド行列を表す NumPy* 1D 配列を作成します。
   `array = np.array([], dtype=type)`

   インテル® DAAL は、NumPy* データ型 np.float64、np.float32、np.intc をサポートしています。

2. 上/下パックド三角行列数値テーブルを作成します。
3. 上/下パックド対称行列数値テーブルを作成します。

コード例

```
from daal.data_management import PackedTriangularMatrix,PackedSymmetricMatrix,
NumericTableIface
import numpy as np
from utils import printArray
array = np.array([1,2,3,4,5,6,7,8,9,10], dtype=np.intc)

nT_LowerPTM =
PackedTriangularMatrix( NumericTableIface.lowerPackedTriangularMatrix,
array, DataType = np.intc)

nT_UpperPTM =
PackedTriangularMatrix( NumericTableIface.upperPackedTriangularMatrix,
array, DataType = np.intc)


nT_LowerPSM =
PackedSymmetricMatrix( NumericTableIface.lowerPackedSymmetricMatrix,
array, DataType = np.intc)

nT_UpperPSM =
PackedSymmetricMatrix( NumericTableIface.upperPackedSymmetricMatrix,
array, DataType = np.intc)
```

ii. pandas* DataFrame を利用したデータのロードと数値テーブルの作成

pandas* DataFrame からパックド nT を作成するステップ

1. 宣言された dtype を含む pandas* DataFrame を作成します。
   `df = pd.DataFrame(values)`

   インテル® DAAL は、データ型 np.float64、np.float32、np.intc をサポートしています。

2. pandas* df を 1D Numpy* 配列に変換します。
3. 上/下パックド三角行列数値テーブルを作成します。
4. 上/下パックド対称行列数値テーブルを作成します。

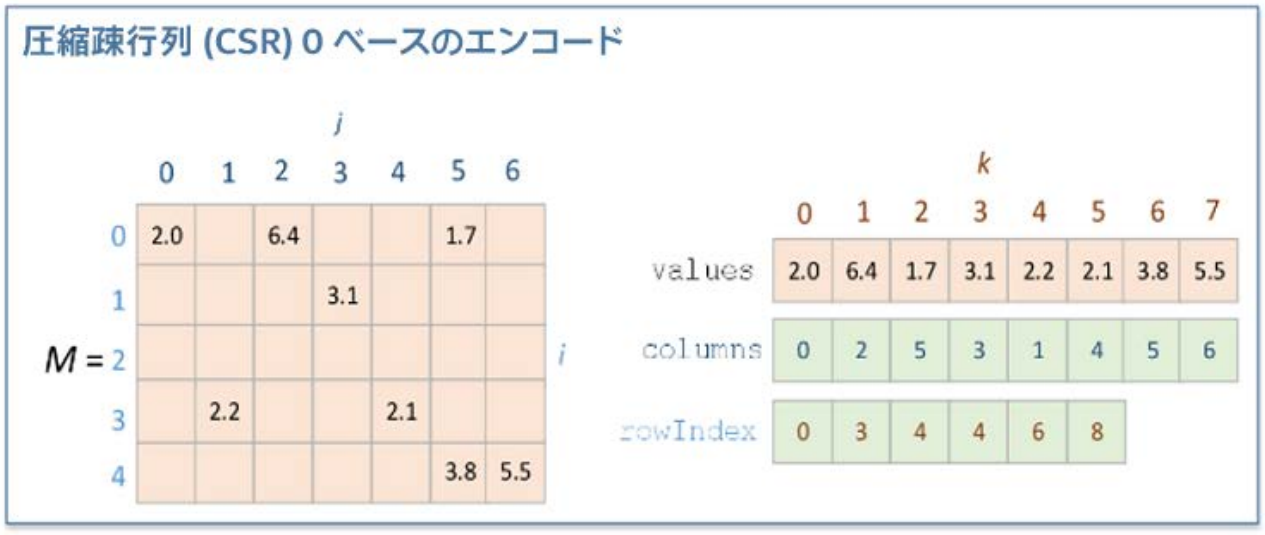
コード例

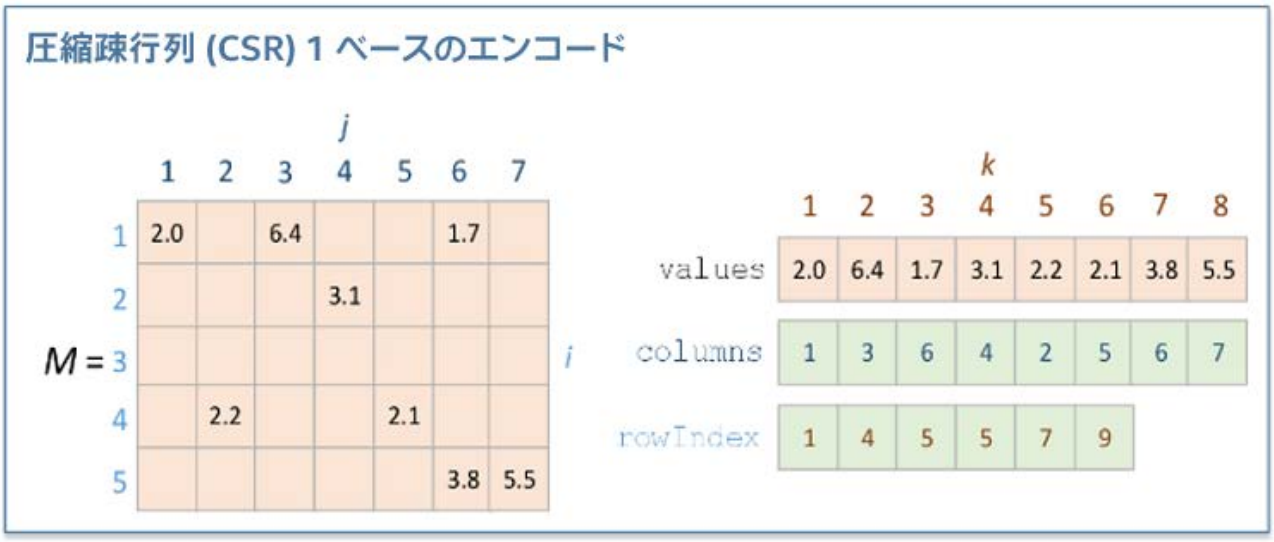
```
from daal.data_management import PackedTriangularMatrix,PackedSymmetricMatrix,
NumericTableIface
import numpy as np
import pandas as pd

df= pd.DataFrame([0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1,
1.2, 1.3, 1.4],
                 dtype= np.float64)
df= pd.DataFrame([1,2,3,4,5,6,7,8,9,0],
                 dtype= np.intc)

array = df.values.ravel()

nT_LowerPTM =
PackedTriangularMatrix( NumericTableIface.lowerPackedTriangularMatrix,
 array, DataType = np.intc)

nT_UpperPTM =
PackedTriangularMatrix( NumericTableIface.upperPackedTriangularMatrix,
 array, DataType = np.intc)

nT_LowerPSM =
PackedSymmetricMatrix( NumericTableIface.lowerPackedSymmetricMatrix,
array, DataType = np.intc)

nT_UpperPSM =
PackedSymmetricMatrix( NumericTableIface.upperPackedSymmetricMatrix,
array, DataType = np.intc)
```

b. 圧縮疎行数値テーブル

基本的な同次数値テーブルが行列データを保存する連続するメモリー位置を割り当てると仮定すると、CSRNumeric テーブル、別のメモリー節約数値テーブルは縮小されたメモリー・フットプリントで配列の値を格納し、疎行列データ形式を保持します。

インテル® DAAL は、疎データ (大量のゼロ要素を含むデータ) をエンコードする特別バージョンの同次数値テーブル用の CSRNumericTable クラスを提供します。ライブラリーは、CSR (Compressed Sparse Row) 形式をエンコーディングに使用します。





CSRNumeric テーブルを作成する前に、3 つの 1D 配列により、以下のように疎行列 "M" を記述しています。

- values: 配列 values は、入力行列の非ゼロの要素を列ごとに含みます。
- columns: 配列 columns の j 番目の要素は、配列 values の j 番目の要素について、行列 M の列インデックスをエンコードします。
- rowIndex: 配列 rowIndex の i 番目の要素は、i 以上でインデックスされた列の最初の非ゼロの要素に対応する、配列 values のインデックスをエンコードします。配列 rowIndex の最後の要素は、行列 M の非ゼロの要素の数をエンコードします。

上記で述べた 3 つの配列を利用して作成された CSRNumeric テーブルは、典型的なゼロの疎データを使用することなく、疎データ形式の数値テーブルを提供します。この表現方法により、行インデックスが圧縮され、行アクセスが高速化されます。

以下のコード例は、NumPy* 配列、pandas* DataFrame および PyDAAL の FileDataSource クラスを使用した数値テーブルの作成を示しています。

i. NumPy* 配列を利用したデータのロードと数値テーブルの作成

NumPy* 配列から CSR nT を作成するステップ

1. 3 つの配列 (非ゼロの値、列インデックス、行オフセット) を作成します。
   `array = np.array([], dtype=type)`

   インテル® DAAL は、数値テーブルにロードされる値として、np.float64、np.float32、np.intc をサポートしています。CSRNumericTable の列インデックスと行オフセットは符号なし int64 にする必要があります。インデックスをほかのデータ型にすると、数値テーブルの作成中に非実装エラーが発生します。

2. 疎行列の行と列の数を宣言します。
3. CSR nT を作成します。
   `nT = CSRNumericTable(non-zero Values, Column indices, Row Offsets, nColumns, nRows)`

コード例

```
import numpy as np
### import Available Modules for CSRNumericTable###
from daal.data_management import CSRNumericTable
# Non zero elements of the matrix
values = np.array([1, -1, -3, -2, 5, 4, 6, 4,-4, 2, 7,8, -5],
                  dtype=np.intc)
# Column indices "colIndices" corresponding to each element in "values" array
colIndices = np.array([1, 2, 4, 1, 2, 3, 4, 5,1, 3, 4,2, 5],
                      dtype=np.uint64)
# Row offsets for every first non zero element encountered in each row
rowOffsets = np.array([1,4,6,9,12,14],
                      dtype=np.uint64)
# Creation of CSR numeric table with the arguments dicussed above

nObservations = 5 # Number of rows in the numpy array
nFeatures = 5# Number of columns in numpy array

CSR_nT = CSRNumericTable(values, colIndices, rowOffsets, nFeatures,
nObservations)
```

ii. pandas* DataFrame を利用したデータのロードと数値テーブルの作成

pandas* DataFrame から CSR nT を作成するステップ

1. pandas* DataFrame を作成して 3 つの配列 (配列の値、列インデックス、行オフセット) の値を初期化します。
   ```
   Df_values= pd.DataFrame([columns = "Values", "ColIndices"
   Df_RowOffsets=pd.DataFrame(columns = ["rowOffsets"])
   ```

行オフセットの値が列インデックス/配列の値と同じサイズでない場合、異なる pandas* df を作成します。

2. DataFrame の値を数値テーブルにロードする NumPy* 配列に変換します。

   インテル® DAAL は、数値テーブルにロードされる値として、np.float64、np.float32、np.intc をサポートしています。CSRNumericTable の列インデックスと行オフセットは符号なし int64 にする必要があります。インデックスをほかのデータ型にすると、数値テーブルの作成中に非実装エラーが発生します。

   ```
   DF [Column name].as_matrix().astype(dtype)
   ```

3. 観測および特徴の数を設定します。
4. CSR 数値テーブルを作成します。
   `CSR_nT = CSRNumericTable(values, colIndices, rowOffsets, nFeatures, nObservations)`

コード例

```
import pandas as pd
import numpy as np
from daal.data_management import CSRNumericTable

#Create DataFrames for values, column indices and rowoffsets
df_Cols_Values = pd.DataFrame(columns = ["values","colIndices"])
df_RowOffsets= pd.DataFrame(columns = ["rowOffsets"])
df_Cols_Values['values'] = [1, -1, -3, -2, 5, 4, 6, 4, -4, 2, 7, 8, -5]
# Column indices "colIndices" corresponding to each element in "values" array
df_Cols_Values['colIndices'] =[1, 2, 4, 1, 2, 3, 4, 5, 1, 3, 4, 2,5]
# Row offsets for every first non zero element encountered in each row
df_RowOffsets['rowOffsets'] = [1, 4, 6, 9, 12,  14]
# Creation of CSR numeric table with the arguments discussed above

#Convert df to numpy arrays with PyDAAL standard dtypes
values= df_Cols_Values['values'].as_matrix().astype(np.intc)
colIndices = df_Cols_Values['colIndices'].as_matrix(). \
                        astype(np.uint64)
rowOffsets = df_RowOffsets['rowOffsets'].as_matrix(). \
                        astype(np.uint64)

nObservations = 5 # Number of rows in the numpy array
nFeatures = 5# Number of columns in numpy array
# Pass the parameters for CSR numeric table creation
CSR_nT = CSRNumericTable(values, colIndices, rowOffsets, nFeatures,
nObservations)
```

3. まとめ

インテル® DAAL は、インテル® MKL とインテル® IPP を利用してデータ分析プロセスを高速化します。この記事で紹介したデータ管理システムは、インテル® DAAL の不可欠な要素であり、さまざまなデータ型とデータレイアウトをサポートする数値テーブルデータ構造で使用されます。インテル® DAAL は、数値テーブルを作成して目的のデータレイアウトを達成するため、NumPy* のようなデータ分析プロセスで使用される一般的なライブラリーと容易に統合できます。その結果、メモリー・フットプリントの縮小や効率的な処理が可能になります。

この記事では、NumPy*、pandas* およびインテル® DAAL データ・ソース・オブジェクトを使用した数値テーブルの作成について説明しました。"PyDAAL 超入門" シリーズの次のパート (パート 2) では、数値テーブルのライフサイクルと数値テーブルの基本的な操作を説明します。データ前処理段階の一部として、数値テーブルは、サニティーチェック、データの取得などの操作、圧縮を使用したシリアル化などを行うさまざまなメソッドを提供します。この記事では、データ処理段階で一般的な数値テーブル操作を実行するさまざまなヘルパー関数についても説明しました。

4. その他の関連リンク

- PyDAAL 超入門: パート 2 数値テーブルの基本操作
- PyDAAL 超入門: パート 3 解析モデルの構築とデプロイメント
- PyDAAL 超入門: パート 4 分散処理とオンライン処理
- インテル® DAAL デベロッパー・ガイド (英語)
- PyDAAL GitHub* チュートリアル (英語)

付録

パックド行列数値テーブルの初期化

パックド三角行列

- **上三角行列**
  - **intc**
    PackedTriangularMatrix(packedLayout=NumericTableIface.upperPackedTriangularMatrix, DataType=intc)
    PackedTriangularMatrix_UpperPackedTriangularMatrixIntc
  - **float32**
    PackedTriangularMatrix(packedLayout=NumericTableIface.upperPackedTriangularMatrix, DataType=float32)
    PackedTriangularMatrix_UpperPackedTriangularMatrixFloat32

    - **float64**
      PackedTriangularMatrix(packedLayout=NumericTableIface.upperPackedTriangular Matrix, DataType=float64)
      PackedTriangularMatrix_UpperPackedTriangularMatrixFloat64
- **下三角行列**
    - **intc**
      PackedTriangularMatrix(packedLayout=NumericTableIface.lowerPackedTriangular Matrix, DataType=intc)
      PackedTriangularMatrix_LowerPackedTriangularMatrixIntc
    - **float32**
      PackedTriangularMatrix(packedLayout=NumericTableIface.lowerPackedTriangular Matrix, DataType=float32)
      PackedTriangularMatrix_LowerPackedTriangularMatrixFloat32
    - **float64**
      PackedTriangularMatrix(packedLayout=NumericTableIface.lowerPackedTriangular Matrix, DataType=float64)
      PackedTriangularMatrix_LowerPackedTriangularMatrixFloat64

パックド対称行列

- **上対称行列**
    - **intc**
      PackedSymmetricMatrix(packedLayout=NumericTableIface.upperPackedSymmetric Matrix, DataType=intc)
      PackedSymmetricMatrix_UpperPackedSymmetricMatrixIntc
    - **float32**
      PackedSymmetricMatrix(packedLayout=NumericTableIface.upperPackedSymmetric Matrix, DataType=float32)
      PackedSymmetricMatrix_UpperPackedSymmetricMatrixFloat32
    - **float64**
      PackedSymmetricMatrix(packedLayout=NumericTableIface.upperPackedSymmetric Matrix, DataType=float64)
      PackedSymmetricMatrix_UpperPackedSymmetricMatrixFloat64
- **下対称行列**
    - **intc**
      PackedSymmetricMatrix(packedLayout=NumericTableIface.lowerPackedSymmetric Matrix, DataType=intc)
      PackedSymmetricMatrix_LowerPackedSymmetricMatrixIntc
    - **float32**
      PackedSymmetricMatrix(packedLayout=NumericTableIface.lowerPackedSymmetric

Matrix, DataType=float32)
PackedSymmetricMatrix_LowerPackedSymmetricMatrixFloat32

- **float64**
  PackedSymmetricMatrix(packedLayout=NumericTableIface.lowerPackedSymmetric
  Matrix, DataType=float64)
  PackedSymmetricMatrix_LowerPackedSymmetricMatrixFloat32 float64
  PackedSymmetricMatrix(packedLayout=NumericTableIface.lowerPackedSymmetric
  Matrix, DataType=float64)

次のパート: パート 2: 数値テーブルの基本操作

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。