PyDAAL 超入門: パート 2 数値テーブルの基本操作

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 2 Basic Operations on Numeric Tables」の日本語参考訳です。

前のパート: パート 1: データ構造

インテル® データ・アナリティクス・ライブラリー (インテル® DAAL) では、さまざまなクラスを利用して、各種データレイアウト、データ型、頻繁なアクセスを提供する数値テーブルを作成できます。このシリーズのパート 1 では、異なるシナリオにおける数値テーブルの作成を説明しました。インテル® DAAL は、作成した数値テーブルを可視化および変形するさまざまなメソッドを提供しています。パート 2 では、これらのメソッドの使用方法を説明します。また、このシリーズのパート 3 では、PyDAAL の「アルゴリズム」セクションの概要を説明します。表 1 は、インテル® DAAL の数値テーブルの基本的な操作のクイック・リファレンスです。

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*、インテル® DAAL、mpi4py (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。

表 1. 利用可能なメソッドのクイック・リファレンス

メソッドの説明	メソッドの構文
* メモリーに格納されている数値テーブルのレイアウトを出力します。入力引数として "nT" を指定する必要があります。	`printNumericTable(nT)`
* 複数の数値テーブルを可視化します。	`printNumericTables(nT1,nT2)`

数値テーブルの形状をチェックします。	`# 行の数`
	`nT.getNumberOfRows()`
	`# 列の数`
	`nT.getNumberOfColumns()`

アクセスおよび操作のために数値テーブルのブロックをロードするバッファーを割り当てます。	`block = BlockDescriptor_Float64()` `# double データ型でメモリーブロックを割り当て`

可視化のブロック記述子に数値テーブルから列および行のブロックを取得します。(バッファーへのアクセスを読み取り専用にするには `rwflag` を "readOnly" に設定します。)	`# 列の値のブロック`
	`nT.getBlockOfColumnValues(colIndex, firstRowIndex,lastRowIndex, rwflag, block)`
	`# 行の値のブロック`
	`nT.getBlockOfRows(firstRowIndex,lastRowIndex, rwflag, block)`

値のブロックがロードされている場合、ブロック記述子オブジェクトから NumPy* 配列を抽出します。	`block.getArray()`
バッファーから行のブロックを解放します。	`nT.releaseBlockOfRows(block)`
* 数値テーブルの配列を出力します。入	`printArray(block.getArray() , num_printed_cols, num_printed_rows, num_cols, message)`

メソッドの説明	メソッドの構文
力引数として "np.array" を指定する必要があります。	
数値テーブルデータ辞書の各列の特徴タイプをチェックします。	`dict[colIndex].featureType`

* <install_root>/share/pydaal_examples/examples/python/source/ の utils フォルダーに含まれる関数を表示します。<install_root>

数値テーブルのライフサイクルのフェーズ

1. 初期化

最初に、直接 NumPy* 配列から数値テーブル (nT) を作成します。以降のコード例では、この nT を使用します。

```
import numpy as np
from daal.data_management import HomogenNumericTable
array =np.array([[1,2,3,4],
                 [5,6,7,8]])
nT= HomogenNumericTable(array)
```

2. 操作

初期化の後、数値テーブルは、pandas* DataFrame に類似したデータにアクセスおよびデータを操作する、さまざまなクラスとメンバー関数を提供します。データ辞書と呼ばれるインテル® DAAL のブックキーピング・オブジェクトについて説明した後、インテル® DAAL の操作方法を説明します。

データ辞書

このシリーズのパート 1 で説明したように、インテル® DAAL の数値テーブル作成では、データ辞書を使用してこれらのカスタムデータ構造を操作します。数値テーブル構造に格納するため生データをメモリーにストリームするとき、テーブルのデータ辞書はメタデータを同時に記録します。ユーザーが非割り当てを指定しない限り、辞書は自動的に作成されます。さまざまなデータ辞書のメソッドを、特徴タイプ、データ型などのアクセスおよび操作に利用できます。ユーザーがメモリー割り当てを行わずに数値テーブルを作成した場合、データ辞書の値に特徴タイプを設定する必要があります。重要な点は、インテル® DAAL のデータ辞書は Python* 辞書ではなくカスタムデータ構造であることです。

詳細は、インテル® DAAL データ辞書での作業を参照してください。

2.1 数値テーブルのデータの変形

2.1.1 標準化と正規化

通常、データ分析作業の前に、データ処理、品質チェック、Null 値や外れ値の制御を含む、データ前処理を行います。前処理段階で重要なことは入力データを正規化することです。インテル® DAAL には、数値テーブルで 2 つの一般的な正規化手法 (Z スコア標準化および min-max 正規化) をサポートするルーチンが用意されています。

現在、インテル® DAAL は、記述的解析の再スケーリングのみをサポートしています。今後、新しいデータに適用される "transform()" メソッドの追加を含む予測的解析のサポートを追加する予定です。

- Z スコア標準化: 値がそれぞれ平均値から外れた標準偏差の数になるように数値テーブルの値を特徴方向に再スケーリングします。インテル® DAAL の Z スコア標準化を使用するステップを以下に示します。

```
import daal.algorithms.normalization.zscore as zscore

# Create an algorithm
algorithm = zscore.Batch(method=zscore.sumDense)

# Set input object for the algorithm to nT
algorithm.input.set(zscore.data, nT)

# Compute Z-score normalization function
res = algorithm.compute()

#Retrieve normalized nT
Norm_nT= res.get(zscore.normalizedData)
```

- min-max 正規化: [0, 1] / [-1-1] 範囲に収まるように数値テーブルの値を特徴方向に線形的に再スケーリングします。インテル® DAAL の min-max 正規化を使用するステップを以下に示します。

```
import daal.algorithms.normalization.minmax as minmax

# Create an algorithm
algorithm = minmax.Batch(method=minmax.defaultDense)

# Set lower and upper bounds for the algorithm
algorithm.parameter.lowerBound = -1.0
algorithm.parameter.upperBound = 1.0

# Set input object for the algorithm to nT
algorithm.input.set(minmax.data, nT)

# Compute Min-max normalization function
res = algorithm.compute()

# Get normalized numeric table
Norm_nT = res.get(minmax.normalizedData)
```

2.1.2 可視化および変形のブロック記述子

可視化または操作のため数値テーブルの内容に直接アクセスすることはできません。代わりに、まず、要求されたデータ値のブロックをメモリーバッファーに移動する必要があります。インスタンス化されると、このバッファーは `BlockDescriptor` と呼ばれるオブジェクトに格納されます。インテル® DAAL の数値テーブル・オブジェクトには、行/列のブロックを検索して `BlockDescriptor` に追加するメンバー関数があります。`rwflag` 引数を使用して、ユーザーがブロックを解放するときに数値テーブルの値を更新するかどうかに応じて、"readOnly"/"readWrite" モードに設定します。`BlockDescriptor` クラスは、特定の行または列のデータのブロックを取得できます。注: `BlockDescriptor` バッファーのデータのデータ型はブロックを取得した数値テーブルのデータ型と一致している必要はありません。

アクセスモード

- ブロックがバッファーメモリーから解放されたときに数値テーブルを更新しないようにするには (数値テーブルの内容へのアクセスを読み取り専用にするには)、`rwflag` 引数を "readOnly" に設定します。

  構文と使用方法

  ```
  from daal.data_management import BlockDescriptor_Float64, readOnly
  #Allocate a readOnly memory block with double dtype
  block = BlockDescriptor_Float64()
  nT.getBlockOfRows(0,1, readOnly, block)
  ```

- ブロックがバッファーメモリーから解放されたときにブロック記述子オブジェクトの変更を数値テーブルに書き込むには (ブロック記述子を使用して数値テーブルを変形できるようにするには)、`rwflag` 引数を "readWrite" に設定します。

  構文と使用方法

  ```
  from daal.data_management import BlockDescriptor_Float64, readWrite
  #Allocate a readOnly memory block with double dtype
  block = BlockDescriptor_Float64()
  nT.getBlockOfRows(0,1, readWrite, block)
  ```

2.1.3 "readWrite" モードのブロック記述子

`getBlockOfRows()`/`getBlockOfColumnValues()` で `rwflag` 引数を "readWrite" に設定すると、行のブロックを解放するときに `BlockDecriptor` オブジェクトの内容が数値テーブルに書き込まれ、数値テーブルの既存の行/列で編集が可能になります。

"readWrite" モードの `BlockDecriptor` を詳細に説明するため、基本的な数値テーブルを作成します。

```
import numpy as np
from daal.data_management import HomogenNumericTable, readWrite,
BlockDescriptor
```

```
from utils import printNumericTable
array =np.array([[1,2,3,4],
                 [5,6,7,8]])
nT= HomogenNumericTable(array)
```

- **行方向に数値テーブルを編集**

```
printNumericTable(nT,"Original nT: ")
#Create buffer object with ntype "double"
doubleBlock = BlockDescriptor(ntype=np.float64)

firstRow = 0
lastRow = nT.getNumberOfRows()

#getBlockOfRows() member function in "readWrite" mode to retrieve
numeric table contents and populate "doubleBlock" object
nT.getBlockOfRows(firstRow,lastRow, readWrite, doubleBlock)
#Access array contents from "doubleBlock" object
array = doubleBlock.getArray()
#Mutate 1st row of array to reflect on "doubleBlock" object
array[0] = [0,0,0,0]
#Release buffer object and write changes back to numeric table
nT.releaseBlockOfRows(doubleBlock)
printNumericTable(nT,"Updated nT: ")
```

nT はデータ [[1,2,3,4],[5,6,7,8]] で作成されました。列の変形後、最初の行はバッファーメモリーを使用して置換されます。変形後の nT はデータ [[0,0,0,0],[5,6,7,8]] になります。

- **列方向に数値テーブルを編集**

```
printNumericTable(nT,"Original nT: ")
#Create  buffer object with ntype "double"
doubleBlock = BlockDescriptor(ntype=np.intc)
ColIndex = 2
firstRow = 0
lastRow = nT.getNumberOfRows()

#getBlockOfColumnValues() member function in "readWrite" mode to
retrieve numeric table ColIndex contents and populate "doubleBlock"
object
nT.getBlockOfColumnValues(ColIndex,firstRow,lastRow,readWrite,doubleBlo
ck)

#Access array contents from "doubleBlock" object
array = doubleBlock.getArray()

#Mutate array to reflect on "doubleBlock" object
array[:][:] = 0

#Release buffer object and write changes back to numeric table
nT.releaseBlockOfColumnValues(doubleBlock)
printNumericTable(nT, "Updated nT: ")
```

nT はデータ [[1,2,3,4],[5,6,7,8]] で作成されました。行の変形後、3 つ目の列はバッファーメモリーを使用して [0,0] に置換されます。変形後の nT はデータ [[1,2,0,4],[5,6,0,8]] になります。

2.1.4 数値テーブルのマージ

数値テーブルは行および列に沿って追加でき、マージする軸に沿って同じ配列サイズを共有します。`RowMergedNumericTable()` および `MergedNumericTable()` は、数値テーブルのマージに利用できる 2 つのクラスです。後者は、列インデックスのマージに使用されます。

- **行方向にマージ**

  構文

```
mnT = RowMergedNumericTable()
mnT.addNumericTable(nT1); mnT.addNumericTable(nT2);
mnt.addNumericTable(nT3)
```

  コード例

```
from daal.data_management import HomogenNumericTable,
RowMergedNumericTable
import numpy as np
from utils import printNumericTable


#nT1 and nT2 are 2 numeric tables having equal number of COLUMNS
array =np.array([[1,2,3,4],
                 [5,6,7,8]], dtype = np.intc)
nT1= HomogenNumericTable(array)
array =np.array([[9,10,11,12],
                 [13,14,15,16]],dtype = np.intc)
nT2= HomogenNumericTable(array)

#Create merge numeric table object
mnT = RowMergedNumericTable()

#add numeric tables to merged numeric table object
mnT.addNumericTable(nT1); mnT.addNumericTable(nT2)
printNumericTable(nT1, "Numeric Table nT1: ")
printNumericTable(nT2, "Numeric Table nT2: ")
printNumericTable(mnT, "Merged Numeric Table nT1 and nT2: ")
```

  出力

```
1.000    2.000    3.000    4.000
5.000    6.000    7.000    8.000&
9.000    10.000   11.000   12.000
13.000   14.000   15.000   16.000
```

- **列方向にマージ**

  構文

  ```
  mnT = MergedNumericTable()
  mnT.addNumericTable(nT1); mnT.addNumericTable(nT1);
  mnt.addNumericTable(nT3)
  ```

  コード例

  ```
  from daal.data_management import HomogenNumericTable,
  MergedNumericTable
  import numpy as np
  from utils import printNumericTable

  #nT1 and nT2 are 2 numeric tables having equal number of ROWS
  array =np.array([[1,2,3,4],
                   [5,6,7,8]], dtype = np.intc)
  nT1= HomogenNumericTable(array)

  array =np.array([[9,10,11,12],
                   [13,14,15,16]],dtype = np.intc)
  nT2= HomogenNumericTable(array)

  #Create merge numeric table object
  mnT = MergedNumericTable()

  #add numeric tables to merged numeric table object
  mnT.addNumericTable(nT1); mnT.addNumericTable(nT2)

  printNumericTable(nT1, "Numeric Table nT1: ")
  printNumericTable(nT2, "Numeric Table nT2: ")
  printNumericTable(mnT, "Merged Numeric Table nT1 & nT2: ")
  ```

  出力

  ```
  1.000    2.000    3.000    4.000    9.000    10.000   11.000   12.000
  5.000    6.000    7.000    8.000    13.000   14.000   15.000   16.000
  ```

2.1.5 数値テーブルの分割

行または列の値での数値テーブルのセクションの抽出に使用する `getBlockOfRows()` および `getBlockOfColumnValues()` メソッドの詳細は、表 1 を参照してください。下記の `getBlockOfNumericTable()` ヘルパー関数を使用して、選択した行および列の範囲を含む連続するテーブルのサブセットを抽出する機能を実装します。`getBlockOfNumericTable()` では、行および列の範囲に、従来の Python 0 ベースのインデックスを使用した、int またはリストキーワード引数を指定できます。

構文と使用方法: `getBlockOfNumericTable(nT, Rows = 'All', Columns = 'All')`

ヘルパー関数

```
def getBlockOfNumericTable(nT,Rows = 'All', Columns = 'All'):
    from daal.data_management import HomogenNumericTable_Float64, \
    MergedNumericTable, readOnly, BlockDescriptor
    import numpy as np
#------------------------------------------------------
    # Get First and Last Row indexes
    lastRow = nT.getNumberOfRows()
    if type(Rows)!= str:
        if type(Rows) == list:
            firstRow = Rows[0]
            if len(Rows) == 2: lastRow = min(Rows[1], lastRow)
        else:firstRow = 0; lastRow = Rows
    elif Rows== 'All':firstRow = 0
    else:
        warnings.warn('Type error in "Rows" arguments, Can be only int/list
type')
        raise SystemExit
#------------------------------------------------------
    # Get First and Last Column indexes
    nEndDim = nT.getNumberOfColumns()
    if type(Columns)!= str:
        if type(Columns) == list:
            nStartDim = Columns[0]
            if len(Columns) == 2: nEndDim = min(Columns[1], nEndDim)
        else: nStartDim = 0; nEndDim = Columns
    elif Columns == 'All': nStartDim = 0
    else:
        warnings.warn ('Type error in "Columns" arguments, Can be only
int/list type')
        raise SystemExit
#------------------------------------------------------
    #Retrieve block of Columns Values within First & Last Rows
    #Merge all the retrieved block of Columns Values
    #Return merged numeric table
    mnT = MergedNumericTable()
    for idx in range(nStartDim,nEndDim):
        block = BlockDescriptor()
        nT.getBlockOfColumnValues(idx,firstRow,(lastRow-
firstRow),readOnly,block)
        mnT.addNumericTable(HomogenNumericTable_Float64(block.getArray()))
        nT.releaseBlockOfColumnValues(block)
    block = BlockDescriptor()
    mnT.getBlockOfRows (0, mnT.getNumberOfRows(), readOnly, block)
    mnT = HomogenNumericTable (block.getArray())
    return mnT
```

この関数に引数を渡す方法は 4 つあります。

1. `getBlockOfNumericTable(nT)` - nT のすべての行と列を含む数値テーブルのブロックを抽出します。
2. `getBlockOfNumericTable(nT, Rows = 4, Columns = 5)` - nT の最初の 4 つの行の値および最初の 5 つの列の値を取得します。

3. `getBlockOfNumericTable(nT, Rows=[2,4], Columns = [1,3])` - リストのパラメーターとして渡された下限および上限を使用して、行および列方向に沿って数値テーブルをスライスします。
4. `getBlockOfNumericTable(nT, Rows=[1,], Columns = [1,])` - 最後のインデックスの下限からすべての行と列を抽出します。

2.1.6 特徴タイプの変更

数値テーブル・オブジェクトには、各列のデータ辞書の特徴タイプを取得および設定する辞書処理メソッドが用意されています。インテル® DAAL でサポートされているデータ辞書では、カテゴリカル(0)、順序(1)、連続(2)の特徴タイプを利用できます。

- **nT と関連付ける辞書オブジェクトの取得**

  構文: `nT.getDictionary()`

  コード例

```
dict = nT.getDictionary() # nT is numeric table created in section 1
'''
'dict' object has data dictionary of numeric table nT. This can be used
to update metadata information about the data. Most common use case is
to modify default feature type of nT columns.
'''
# Print default feature type of 3rd feature (example feature is
continuous):
print(dict[2].featureType) #outputs "2" (denotes Continuous feature
type)

# Modify feature type from Continuous to Categorical:
dict[2].featureType = data_feature_utils.DAAL_CATEGORICAL
print(dict[2].featureType) #outputs "0" (denotes Categorical feature
type)
```

- **nT と関連付ける辞書オブジェクトの設定**

  必要に応じて、このメソッドを使用して、現在のデータ辞書の値を置換するか、新しいデータ辞書を作成します。バッチ更新では、`setDictionary()` メソッドを使用して既存のデータ辞書を上書きできます。

  データ辞書にメモリーを割り当てないでテーブルを作成した場合、`setDictionary()` メソッドを使用してテーブルの特徴のメタデータを構築する必要があります。セクション 1 で作成した 4 つの特徴を含む nT について再度考えます。

  構文: `nT.setDictionary()`

コード例

```
nT.releaseBlockOfRows(block)

#Create a dictionary object using Numeric table dictionary class with
the number of features
dict = NumericTableDictionary(nFeatures)
#Allocate a feature type for each feature
dict[0].featureType = data_feature_utils.DAAL_CONTINUOUS
dict[1].featureType = data_feature_utils.DAAL_CATEGORICAL
dict[2].featureType = data_feature_utils.DAAL_CONTINUOUS
dict[3].featureType = data_feature_utils.DAAL_CATEGORICAL

#set the nT numeric table dictionary with "dict"
nT.setDictionary(dict)
```

2.2 数値テーブルのディスクへのエクスポート

数値テーブルは、NumPy* バイナリーファイル (.npy) としてエクスポートしてディスクに保存できます。次の 2 つのセクションでは、タスクをバイナリー形式で保存するヘルパー関数と、ディスクのデータを圧縮するヘルパー関数を説明します。

2.2.1 シリアル化

インテル® DAAL は、数値テーブル・オブジェクトを NumPy* 配列オブジェクトに変換できるデータアーカイブにシリアル化するインターフェイスを提供します。データがシリアル化された NumPy* 配列は、ディスクに保存し、後からリロードして元の数値テーブルを再構築できます。

このプロセスを自動化するには、次のヘルパー関数を使用してシリアル化してディスクに保存します。

ヘルパー関数

```
def Serialize(nT):
#Construct input data archive Object
#Serialize nT contents into data archive Object
#Copy data archive contents to numpy array
#Save numpy array as .npy in the path
   from daal.data_management import InputDataArchive
   import numpy as np

   dataArch = InputDataArchive()

   nT.serialize(dataArch)

   length = dataArch.getSizeOfArchive()
   buffer_array = np.zeros(length, dtype=np.ubyte)
   dataArch.copyArchiveToArray(buffer_array)

   return buffer_array
buffer_array = Serialize(nT) # call helper function
#np.save(<path>, buffer)# This step is optional
```

</path>

2.2.2 圧縮

大規模なデータセットをディスクに格納する必要がある場合、インテル® DAAL の圧縮メソッドを利用してメモリー・フットプリントを縮小できます。インテル® DAAL 数値テーブルのシリアル化された配列表現は、ディスクに保存する前に圧縮できるため、ストレージを効率的に利用できます。

このプロセスを自動化するには、次のヘルパー関数を使用してシリアル化した後、シリアル化された配列を圧縮します。

数値テーブルを圧縮してディスクに書き込むには、`Serialize(nT)` および `CompressToDisk (nT, path)` ヘルパー関数を使用します。

ヘルパー関数

```
def CompressToDisk(nT, path):
    # Serialize nT contents
    # Create a compressor object
    # Create a stream for compression
    # Write numeric table to the compression stream
    # Allocate memory to store the compressed data
    # Store compressed data
    # Save compressed data to disk
    from daal.data_management
    import Compressor_Zlib, level9, CompressionStream
    import numpy as np

    buffer = Serialize (nT)
    compressor = Compressor_Zlib ()
    compressor.parameter.gzHeader = True
    compressor.parameter.level = level9
    comprStream = CompressionStream (compressor)
    comprStream.push_back (buffer)
    compressedData = np.empty (comprStream.getCompressedDataSize (),
dtype=np.uint8)
    comprStream.copyCompressedArray (compressedData)
    np.save (path, compressedData)
    CompressToDisk (nT, < path >)
```

2.3 数値テーブルのディスクからのインポート

前のセクションで述べたように、数値テーブルは、シリアル化または圧縮された NumPy* ファイル形式で格納できます。数値テーブルを再構築するには、展開/逆シリアル化メソッドを利用します。

2.3.1 逆シリアル化

下記のヘルパー関数は、シリアル化された配列オブジェクトから数値テーブルを再構築します。

ヘルパー関数

```
def DeSerialize(buffer_array):
    from daal.data_management import OutputDataArchive, HomogenNumericTable
    #Load serialized contents to construct output data archive object
    #De-serialize into nT object and return nT

    dataArch = OutputDataArchive(buffer_array)
    nT = HomogenNumericTable()
    nT.deserialize(dataArch)
    return nT
#buffer_array = np.load(path) # this step is optional, used only when
serialized contents have to be written to  disk
nT = DeSerialize(buffer_array)
```

2.3.2 展開

圧縮段階は数値テーブル・オブジェクトのシリアル化を含むように、展開段階は逆シリアル化を含みます。数値テーブルを逆シリアル化するには、`DeSerialize` ヘルパー関数を使用します。展開ヘルパー関数については、下記を参照してください。

数値テーブルを展開してディスクから読み取るには、`DeSerialize(buffer_array)` および `DeCompressFromDisk(path)` ヘルパー関数を使用します。

ヘルパー関数

```
def DeCompressFromDisk(path):
    from daal.data_management import  Decompressor_Zlib, DecompressionStream
    # Create a decompressor
    decompressor = Decompressor_Zlib()
    decompressor.parameter.gzHeader = True

    # Create a stream for decompression
    deComprStream = DecompressionStream(decompressor)

    # Write the compressed data to the decompression stream and decompress it
    deComprStream.push_back(np.load(path))

    # Allocate memory to store the decompressed data
    deCompressedData = np.empty(deComprStream.getDecompressedDataSize(),
dtype=np.uint8)

    # Store the decompressed data
    deComprStream.copyDecompressedArray(deCompressedData)

    #Deserialize
    return DeSerialize(deCompressedData)

nT = DeCompressFromDisk(<path>)#path must be ‘.npy’ file
```

インテル® DAAL は、ZLIB、LZO、RLE、BZIP (リファレンス (英語)) を含む、その他の汎用の圧縮/展開メソッドも実装します。

まとめ

インテル® DAAL のデータ管理コンポーネントは、数値テーブルの内容に対して一般的な操作を行うクラスおよびメソッドを提供します。基本的な数値テーブルの操作は、アクセス、変形、ディスクへのエクスポートおよびディスクからのインポートを含みます。この記事で説明しているヘルパー関数は、インテル® DAAL の数値テーブルサブセットの作成、およびシリアル化と圧縮プロセスの自動化に役立ちます。

超入門シリーズの次のパート (パート 3) では、PyDAAL のアルゴリズム・セクションの概要を説明します。パート 3 では、インテル® DAAL で利用可能な、重要な記述アルゴリズムと予測アルゴリズムのワークフローに注目します。ハイパーパラメーターの設定、適切な計算の分散、シリアル化されたオブジェクトとしてのモデルの配備などの高度な機能についても説明します。

その他の関連リンク

- PyDAAL 超入門: パート 1 データ構造
- 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)

次のパート: パート 3: 解析モデルの構築とデプロイメント

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。