AD-Tree Package User Manual

1 Introduction for Record, AD-Tree and Contingency Table modules

When using this package, each of “Record”, “ADTree” and “ContingencyTable” modules must be imported to program. However, there are more than one implementations for each of these three modules. As a result, please consider which implementations is more appropriate before importing.

Note: The name of the folder where all python files are, must not be “adtree” or “ADTree”, since some AD-Tree modules are imported as this name. A folder with name of “adtree” or “ADTree” will cause this package does not work.

2 Record modules

Record modules are the only form of original dataset the ADTree modules accept. It is a two-dimensional matrix, in which all numbers are greater than 0. In each row, there stores one record.

The reason for using a Record module instead of a list directly, is that the dataset might be too large to put in memory. As a result, the original dataset might be stored in file system or database. Using a Record module to provide query service to build AD-Tree and contingency table could make this package fit different formats of dataset, only if the interface is implemented.

2.1 Interface

There are three functions and three global variables as its interface:

- **initRecord(args)**

  This function initialises the two-dimensional matrix. Parameter “args” passes all the information necessary to initialise dataset. The content of “args” represents could be different in different implementation of Record module. It has no return value.

- **getRecord(row, column)**

  This function returns the number in a particular row and column in the matrix. Parameter “row” and “column” all count from 0.

- **count(query)**

  This function returns the counting of a particular record in the dataset. Parameter “query” represents one record in the type of list.

- **arityList**

  “arityList” is a list which contains aritys of each attributes.

  For example, for a dataset in which each record has three attributes, and they range from 1 to 2, 1 to 4, and 1 to 2 respectively, the arityList will be list: [2, 4, 2]

- **arityLength**

  “arityLength” is an integer which represents the length of arityList. It also equals to how many attributes there are in a record.

- **recordsLength**

"recordsLength" is an integer which represents how many records there are in the dataset.

2.2 Implemented modules

In this package, there is only one implementation of Record, called "ArrayRecord". ArrayRecord stores dataset simply in a list. The parameter of "args" in initiRecord should be a list which contains two elements. args[0] represents the arity list, and args[1] represents the records table, which is actually a list of list.

3 ADTree modules

ADTree modules accept Record module as input, and are used to build contingency tables.

3.1 Interface

There are one global function and two classes as interface:

- **importModules(recordModule)**

  This function is used after importing this module, before creating any ADNode, to declare which Record module the ADTree module is using.

Class ADNode:

- **__init__(startAttributeNum, recordNums)**

  The constructor of ADNode. Parameter startAttributeNum passes starts from which attributes will this AD-node be built for. Usually this parameter is 1, which means the AD-node represents all attributes in arity list. Parameter recordNums passes which records in dataset will this AD-node be built for. Usually this parameter is a list which contains numbers for all records.

- **getCount()**

  This function returns the counting of this AD-node.

- **getVNChild(attributeNum)**

  This function returns the vary node child of this AD-node, which represents attribute number of "attributeNum".

Class VaryNode:

- **__init__(attributeNum, recordNums)**

  The constructor of VaryNode. Parameter attributeNum passes the attribute value this vary node represents. Parameter recordNums passes which records in dataset will this vary node be built for.

- **getMCV()**

  This function returns the MCV value of this vary node.

- **getADNChild(attributeValue)**

  This function returns the AD-node child of this vary node, which represents attribute value of "attributeValue".

3.2 Implemented modules

In this package, there are five ADTree modules:

- **FullADTree**

  FullADTree is the non-sparse AD-Tree without any optimization measures.

- **ZeroADTree**

  ZeroADTree is the non-sparse AD-Tree with nodes of zero counting cut off.

- **SparseADTree**

  SparseADTree is the sparse AD-Tree with nodes of zero counting cut off.

- **LeafFullADTree**

  LeafFullADTree is the non-sparse AD-Tree optimized with leaf-lists.

- **LeafSparseADTree**

  LeafSparseADTree is the sparse AD-Tree optimized with leaf-lists.

Usually SparseADTree and LeafSparseADTree are used for building contingency tables. Other three types of AD-Tree are used for debugging when implementing new type of contingency table.

For AD-Tree that uses leaf-lists, those modules all have global variable “Rmin”, which is used by some ContingencyTable modules.

4 ContingencyTable modules

ContingencyTable modules accept ADTree modules as input, and could be used to make queries.

4.1 Interface

There are one global function and one class as interface:

- **importModules(recordModule, ADTreeModule)**

  This function is used after importing this module, before creating any contingency table, to declare which Record module and ADTree module the ContingencyTable module is using.

Class ContingencyTable

- **__init__(attributeList, ADN)**

  The constructor of class ContingencyTable. Parameter attributeList passes a list of attribute numbers (not attribute values) to build a contingency table. The attributeList should be a subset of arityList, which contains only the numbers of those attributes that user concerns. Parameter ADN is the root AD-node of the AD-Tree which the contingency table is built from.

- **getCount(query)**

  This function return the counting of a particular query. Parameter query is a list of values according to attributeList.

4.2 Implemented Modules

In this package, there are seven ContingencyTable modules:

- **FullContingencyTable**

FullContingencyTable is the original version of contingency table, without any optimization, which represents the table as a two-dimension arry(list).

This version supports to be built from any types of AD-Tree except those that use leaf-lists.

- **ListContingencyTable**

ListContingencyTable is contingency table that represents the table as simply one-dimension array(list). It also has optimization of product list that making it faster to make queries, and optimization on the way of building product list in constructor. Also, function list.append() is totally avoided to optimize.

This contingency table supports to be built from all types of AD-Tree, including LeafSparseADTree, and could recognise AD-Tree that using leaf-lists by checking module name (if there exists sub-string “Leaf” or “leaf” in AD-Tree module name, it will consider that is using leaf-lists).

- **IteratedContingencyTable**

IteratedContingencyTable is optimized based on ListContingencyTable, which uses a loop and a stack to build the table (iteration), instead of the constructor calling itself (recursion).

This contingency table supports to be built from all five types of AD-Tree,.

- **DictContingencyTable**

DictContingencyTable is optimized based on ListContingencyTable, which uses dict instead of list to store numbers.

This contingency table supports to be built from all five types of AD-Tree.

- **IteratedDictContingencyTable**

IteratedDictContingencyTable is optimized based on DictContingencyTable, which uses a loop and a stack to build the table (iteration), instead of the constructor calling itself (recursion).

This contingency table supports to be built from all five types of AD-Tree.

- **TreeContingencyTable**

TreeContingencyTable uses tree structure instead of any linear structure, such like list and dict.

TreeContingencyTable only supports to be built from any types of AD-Tree except those that use leaf-lists.

- **IteratedTreeContingencyTable**

IteratedTreeContingencyTable is optimized based on TreeContingencyTable, which uses a loop and a stack to build the tree (iteration), instead of the constructor calling itself (recursion). And by only using list in python to represent tree structure, “node” class is eliminated. This version is currently tested be the fastest one.

IteratedTreeContingencyTable only supports to be built from any types of AD-Tree except those that use leaf-lists.

5 Example of building AD-Tree and Contingency Table

All the interfaces are shown above, but actually only six functions are necessary for user to build AD-Tree and contingency table to make query:

- **Record.initRecord(args)**
- **ADTree.importModules(recordModule)**
- **ADTree.ADNode(startAttributeNum, recordNums)**
- **ContingencyTable.importModules(recordModule, ADTreeModule)**
- **ContingencyTable.ContingencyTable(attributeList, ADN)**
- **ContingencyTable.getCount(query)**

This simple example shows how to use this package to build a sparse AD-Tree and a contingency table of "IteratedTreeContingencyTable", and then make some queries.

```
import ArrayRecord as Record
import SparseADTree as ADTree
import IteratedTreeContingencyTable as ContingencyTable

arityList = [4, 3, 2, 5]
recordsTable = [[1, 2, 1, 4], [2, 2, 2, 5], [1, 3, 1, 1], [4, 1, 2, 1],
                [2, 2, 1, 4], [4, 3, 2, 5], [3, 1, 1, 1], [1, 1, 2, 5]]

if __name__ == '__main__':
    # import the original dataset to the record module
    Record.initRecord([arityList, recordsTable])

    # declare that the ADTree module uses ArrayRecord module as dataset
    ADTree.importModules('ArrayRecord')

    # declare that the ContingencyTable uses ArrayRecord and SparsADTree modules
    ContingencyTable.importModules('ArrayRecord', 'SparseADTree')

    # initialise recordNums containing all numbers in the dataset
    recordNums = [num for num in range(1, Record.recordsLength+1)]

    # build an AD-Tree with attribute list starts from the first attribute,
    # and for all the records
    adtree = ADTree.ADNode(1, recordNums)

    # build a contingency table for the first and third attributes
    contab = ContingencyTable.ContingencyTable([1, 3], adtree)

    # query for [1, 1], [2, 1], [3, 1] and [4, 1], and print on screen
    for i in range(4):
        query = [i, 1]
        count = contab.getCount(query)
        print('Q:', query, 'C:', count)
```

The result of this program is:

```
Q: [0, 1] C: 0
Q: [1, 1] C: 2
Q: [2, 1] C: 1
Q: [3, 1] C: 1
```