datatable Documentation

Release 0.8.0

Pasha Stetsenko

Mar 10, 2019

Getting started

1 Getting started **3**
- 1.1 Install datatable . . . 3
- 1.2 Loading data . . . 3
- 1.3 Data manipulation . . . 4
- 1.4 What the f.? . . . 4
- 1.5 Groupbys / joins . . . 5
- 1.6 Offloading data . . . 6

2 Using `datatable` **7**
- 2.1 Create Frame . . . 7
- 2.2 Convert a Frame . . . 7
- 2.3 Parse Text (csv) Files . . . 8
- 2.4 Write the Frame . . . 8
- 2.5 Save a Frame . . . 8
- 2.6 Basic Frame Properties . . . 8
- 2.7 Compute Per-Column Summary Stats . . . 8
- 2.8 Select Subsets of Rows/Columns . . . 9
- 2.9 Delete Rows/Columns . . . 9
- 2.10 Filter Rows . . . 9
- 2.11 Compute Columnar Expressions . . . 9
- 2.12 Sort Columns . . . 9
- 2.13 Perform Groupby Calculations . . . 10
- 2.14 Append Rows/Columns . . . 10

3 Installation **11**
- 3.1 Requirements . . . 11
- 3.2 Install on Mac OS X . . . 11
- 3.3 Install on Linux . . . 11
- 3.4 Build from Source . . . 12
- 3.5 Troubleshooting . . . 13

4 Contributing **15**

5 Have Questions? **17**

6 Frame **19**

7 Ftrl **27**

8 FTRL **29**
8.1 FTRL Model Information 29
8.2 Create an FTRL Model 29
8.3 FTRL Model Parameters 29
8.4 Training a Model 30
8.5 Resetting a Model 30
8.6 Making Predictions 30
8.7 Feature Importances 31
8.8 Further Reading 31

Data is everywhere. From the smallest photon interactions to galaxy collisions, from mouse movements on a screen to economic developments of countries, we are surrounded by the sea of information. The human mind cannot comprehend this data in all its complexity; since ancient times people found it much easier to reduce the dimensionality, to impose a strict order, to arrange the data points neatly on a rectangular grid: to make a **data table**.

But once the data has been collected into a table, it has been tamed. It may still need some grooming and exercise, essentially so it is no longer scary. Even if it is really Big Data, with the **right tools** you can approach it, play with it, bend it to your will, *master* it.

Python `datatable` module is the right tool for the task. It is a library that implements a wide (and growing) range of operators for manipulating two-dimensional data frames. It focuses on: big data support, high performance, both in-memory and out-of-memory datasets, and multi-threaded algorithms. In addition, `datatable` strives to achieve good user experience, helpful error messages, and powerful API similar to R `data.table`'s.

CHAPTER 1

Getting started

1.1 Install datatable

Let's begin by installing the latest stable version of `datatable` from PyPI:

```
$ pip install datatable
```

If this didn't work for you, or if you want to install the bleeding edge version of the library, please check the *Installation* page.

Assuming the installation was successful, you can now import the library in a JupyterLab notebook or in a Python console:

```
import datatable as dt
print(dt.__version__)
```

1.2 Loading data

The fundamental unit of analysis in datatable is a data `Frame`. It is the same notion as a pandas DataFrame or SQL table: data arranged in a two-dimensional array with rows and columns.

You can create a `Frame` object from a variety of data sources: from a python list or dictionary, from a numpy array, or from a pandas DataFrame.

```
DT1 = dt.Frame(A=range(5), B=[1.7, 3.4, 0, None, -math.inf],
               stypes={"A": dt.int64})
DT2 = dt.Frame(pandas_dataframe)
DT3 = dt.Frame(numpy_array)
```

You can also load a CSV/text/Excel file, or open a previously saved binary `.jay` file:

```
DT4 = dt.fread("~/Downloads/dataset_01.csv")
DT5 = dt.open("data.jay")
```

The `fread()` function shown above is both powerful and extremely fast. It can automatically detect parse parameters for the majority of text files, load data from .zip archives or URLs, read Excel files, and much more.

1.3 Data manipulation

Once the data is loaded into a Frame, you may want to do certain operations with it: extract/remove/modify subsets of the data, perform calculations, reshape, group, join with other datasets, etc. In datatable, the primary vehicle for all these operations is the square-bracket notation inspired by traditional matrix indexing but overcharged with power (this notation was pioneered in R data.table and is the main axis of intersection between these two libraries).

In short, almost all operations with a Frame can be expressed as

where is the row selector, is the column selector, and `...` indicates that additional modifiers might be added. If this looks familiar to you, that's because it is. Exactly the same `DT[i, j]` notation is used in mathematics when indexing matrices, in C/C++, in R, in pandas, in numpy, etc. The only difference that datatable introduces is that it allows to be anything that can conceivably be interpreted as a row selector: an integer to select just one row, a slice, a range, a list of integers, a list of slices, an expression, a boolean-valued Frame, an integer-valued Frame, an integer numpy array, a generator, and so on.

The column selector is even more versatile. In the simplest case, you can select just a single column by its index or name. But also accepted are a list of columns, a slice, a string slice (of the form `"A":"Z"`), a list of booleans indicating which columns to pick, an expression, a list of expressions, and a dictionary of expressions. (The keys will be used as new names for the columns being selected.) The expression can even be a python type (such as `int` or `dt.float32`), selecting all columns matching that type.

In addition to the selector expression shown above, we support the update and delete statements too:

```
DT[i, j] = r
del DT[i, j]
```

The first expression will replace values in the subset `[i, j]` of Frame `DT` with the values from `r`, which could be either a constant, or a suitably-sized Frame, or an expression that operates on frame `DT`.

The second expression deletes values in the subset `[i, j]`. This is interpreted as follows: if `i` selects all rows, then the columns given by `j` are removed from the Frame; if `j` selects all columns, then the rows given by `i` are removed; if neither `i` nor `j` span all rows/columns of the Frame, then the elements in the subset `[i, j]` are replaced with NAs.

1.4 What the f.?

You may have noticed already that we mentioned several times the possibility of using expressions in or and in other places. In the simplest form an expression looks like

```
f.ColA
```

which indicates a column `ColA` in some Frame. Here `f` is a variable that has to be imported from the datatable module. This variable provides a convenient way to reference any column in a Frame. In addition to the notation above, the following is also supported:

```
f[3]
f["ColB"]
```

denoting the fourth column and the column `ColB` respectively.

These f-expression support arithmetic operations as well as various mathematical and aggregate functions. For example, in order to select the values from column `A` normalized to range `[0; 1]` we can write the following:

```
from datatable import f, min, max
DT[:, (f.A - min(f.A))/(max(f.A) - min(f.A))]
```

This is equivalent to the following SQL query:

```
SELECT (f.A - MIN(f.A))/(MAX(f.A) - MIN(f.A)) FROM DT AS f
```

So, what exactly is `f`? We call it a "frame proxy", as it becomes a simple way to refer to the Frame that we currently operate on. More precisely, whenever `DT[i, j]` is evaluated and we encounter an `f`-expression there, that `f` becomes replaced with the frame `DT`, and the columns are looked up on that Frame. The same expression can later on be applied to a different Frame, and it will refer to the columns in that other Frame.

At some point you may notice that that datatable also exports symbol `g`. This `g` is also a frame proxy; however it already refers to the *second* frame in the evaluated expression. This second frame appears when you are *joining* two or more frames together (more on that later). When that happens, symbol `g` is used to refer to the columns of the joined frame.

1.5 Groupbys / joins

In the *Data Manipulation* section we mentioned that the `DT[i, j, ...]` selector can take zero or more modifiers, which we denoted as `...`. The available modifiers are `by()`, `join()` and `sort()`. Thus, the full form of the square-bracket selector is:

1.5.1 by(...)

This modifier splits the frame into groups by the provided column(s), and then applies and within each group. This mostly affects aggregator functions such as `sum()`, `min()` or `sd()`, but may also apply in other circumstances. For example, if is a slice that takes the first 5 rows of a frame, then in the presence of the `by()` modifier it will take the first 5 rows of each group.

For example, in order to find the total amount of each product sold, write:

```
from datatable import f, by, sum
DT = dt.fread("transactions.csv")

DT[:, sum(f.quantity), by(f.product_id)]
```

1.5.2 sort(...)

This modifier controls the order of the rows in the result, much like SQL clause `ORDER BY`. If used in conjunction with `by()`, it will order the rows within each group.

1.5.3 join(...)

As the name suggests, this operator allows you to join another frame to the current, equivalent to the SQL `JOIN` operator. Currently we support only left outer joins.

In order to join frame `X`, it must be keyed. A keyed frame is conceptually similar to a SQL table with a unique primary key. This key may be either a single column, or several columns:

```
X.key = "id"
```

Once a frame is keyed, it can be joined to another frame `DT`, provided that `DT` has the column(s) with the same name(s) as the key in `X`:

```
DT[:, :, join(X)]
```

This has the semantics of a natural left outer join. The `X` frame can be considered as a dictionary, where the key column contains the keys, and all other columns are the corresponding values. Then during the join each row of `DT` will be matched against the row of `X` with the same value of the key column, and if there are no such value in `X`, with an all-NA row.

The columns of the joined frame can be used in expressions using the `g.` prefix, for example:

```
DT[:, sum(f.quantity * g.price), join(products)]
```

Note: In the future, we will expand the syntax of the join operator to allow other kinds of joins and also to remove the limitation that only keyed frames can be joined.

1.6 Offloading data

Just as our work has started with loading some data into `datatable`, eventually you will want to do the opposite: store or move the data somewhere else. We support multiple mechanisms for this.

First, the data can be converted into a pandas DataFrame or into a numpy array. (Obviously, you have to have pandas or numpy libraries installed.):

```
DT.to_pandas()
DT.to_numpy()
```

A frame can also be converted into python native data structures: a dictionary, keyed by the column names; a list of columns, where each column is itself a list of values; or a list of rows, where each row is a tuple of values:

```
DT.to_dict()
DT.to_list()
DT.to_tuples()
```

You can also save a frame into a CSV file, or into a binary .jay file:

```
DT.to_csv("out.csv")
DT.save("data.jay")
```

CHAPTER 2

Using `datatable`

This section describes common functionality and commands that you can run in `datatable`.

2.1 Create Frame

You can create a Frame from a variety of sources, including `numpy` arrays, `pandas` DataFrames, raw Python objects, etc:

```
import datatable as dt
import numpy as np
np.random.seed(1)
dt.Frame(np.random.randn(1000000))
```

```
import pandas as pd
pf = pd.DataFrame({"A": range(1000)})
dt.Frame(pf)
```

```
dt.Frame({"n": [1, 3], "s": ["foo", "bar"]})
```

2.2 Convert a Frame

Convert an existing Frame into a `numpy` array, a `pandas` DataFrame, or a pure Python object:

```
nparr = df1.to_numpy()
pddfr = df1.to_pandas()
pyobj = df1.to_list()
```

2.3 Parse Text (csv) Files

`datatable` provides fast and convenient parsing of text (csv) files:

```
df = dt.fread("train.csv")
```

The `datatable` parser

- Automatically detects separators, headers, column types, quoting rules, etc.
- Reads from file, URL, shell, raw text, archives, glob
- Provides multi-threaded file reading for maximum speed
- Includes a progress indicator when reading large files
- Reads both RFC4180-compliant and non-compliant files

2.4 Write the Frame

Write the Frame's content into a `csv` file (also multi-threaded):

```
df.to_csv("out.csv")
```

2.5 Save a Frame

Save a Frame into a binary format on disk, then open it later instantly, regardless of the data size:

```
df.save("out.jay")
df2 = dt.open("out.jay")
```

2.6 Basic Frame Properties

Basic Frame properties include:

```
print(df.shape)    # (nrows, ncols)
print(df.names)    # column names
print(df.stypes)   # column types
```

2.7 Compute Per-Column Summary Stats

Compute per-column summary stats using:

```
df.sum()
df.max()
df.min()
df.mean()
df.sd()
df.mode()
```

(continues on next page)

(continued from previous page)

```
df.nmodal()
df.nunique()
```

2.8 Select Subsets of Rows/Columns

Select subsets of rows and/or columns using:

```
df[:, "A"]          # select 1 column
df[:10, :]          # first 10 rows
df[::-1, "A":"D"]   # reverse rows order, columns from A to D
df[27, 3]           # single element in row 27, column 3 (0-based)
```

2.9 Delete Rows/Columns

Delete rows and or columns using:

```
del df[:, "D"]      # delete column D
del df[f.A < 0, :] # delete rows where column A has negative values
```

2.10 Filter Rows

Filter rows via an expression using the following. In this example, `mean`, `sd`, `f` are all symbols imported from `datatable`.

```
df[(f.x > mean(f.y) + 2.5 * sd(f.y)) | (f.x < -mean(f.y) - sd(f.y)), :]
```

2.11 Compute Columnar Expressions

Compute columnar expressions using:

```
df[:, {"x": f.x, "y": f.y, "x+y": f.x + f.y, "x-y": f.x - f.y}]
```

2.12 Sort Columns

Sort columns using:

```
df.sort("A")
df[:, :, sort(f.A)]
```

2.13 Perform Groupby Calculations

Perform groupby calculations using:

```
df[:, mean(f.x), by("y")]
```

2.14 Append Rows/Columns

Append rows / columns to a Frame using:

```
df1.cbind(df2, df3)
df1.rbind(df4, force=True)
```

CHAPTER 3

Installation

This section describes how to install H2O's `datatable`.

3.1 Requirements

- Python 3.5+

3.2 Install on Mac OS X

Run the following command to install `datatable` on Mac OS X.

```
pip install datatable
```

3.3 Install on Linux

Run one of the following commands to retrieve the `datatable` whl file for Linux environments.

```
# Python 3.5
pip install https://s3.amazonaws.com/h2o-release/datatable/stable/datatable-0.3.2/
↪datatable-0.3.2-cp35-cp35m-linux_x86_64.whl

# Python 3.6
pip install https://s3.amazonaws.com/h2o-release/datatable/stable/datatable-0.3.2/
↪datatable-0.3.2-cp36-cp36m-linux_x86_64.whl
```

3.4 Build from Source

The key component needed for building the `datatable` package from source is the Clang/Llvm distribution. The same distribution is also required for building the `llvmlite` package, which is a prerequisite for `datatable`. Note that the `clang` compiler which is shipped with MacOS is too old, and in particular it doesn't have support for the OpenMP technology.

3.4.1 Installing the `Clang/Llvm` distribution

1. Visit https://releases.llvm.org/download.html and **download** the most recent version of Clang/Llvm available for your platform (but no older than version 4.0.0).
2. Extract the downloaded archive into any suitable location on your hard drive.
3. Create one of the environment variables `LLVM4` / `LLVM5` / `LLVM6` (depending on the version of Clang/Llvm that you installed). The variable should point to the directory where you placed the Clang/Llvm distribution.

   For example, on Ubuntu after downloading `clang+llvm-4.0.0-x86_64-linux-gnu-ubuntu-16.10.tar.xz` the sequence of steps might look like:

```
$ mv clang+llvm-4.0.0-x86_64-linux-gnu-ubuntu-16.10.tar.xz  /opt
$ cd /opt
$ sudo tar xvf clang+llvm-4.0.0-x86_64-linux-gnu-ubuntu-16.10.tar.xz
$ export LLVM4=/opt/clang+llvm-4.0.0-x86_64-linux-gnu-ubuntu-16.10
```

You probably also want to put the last `export` line into your `~/.bash_profile`.

3.4.2 Building `datatable`

1. Verify that you have Python 3.5 or above:

```
$ python --V
```

   If you don't have Python 3.5 or later, you may want to download and install the newest version of Python, and then create and activate a virtual environment for that Python. For example:

```
$ virtualenv --python=python3.6 ~/py36
$ source ~/py36/bin/activate
```

2. Build `datatable`:

```
$ make build
$ make install
$ make test
```

3. Additional commands you may find occasionally interesting:

```
# Uninstall previously installed datatable
make uninstall

# Build a debug version of datatable (for example suitable for ``gdb``␣
↪debugging)
make debug
```

(continues on next page)

(continued from previous page)

```
# Generate code coverage report
make coverage
```

3.5 Troubleshooting

- If you get an error like `ImportError: This package should not be accessible on Python 3`, then you may have a `PYTHONPATH` environment variable that causes conflicts. See this SO question for details.
- If you see errors such as `"implicit declaration of function 'PyUnicode_AsUTF8' is invalid in C99"` or `"unknown type name 'PyModuleDef'"` or `"void function 'PyInit__datatable' should not return a value "`, it means your current Python is Python 2. Please revisit step 1 in the build instructions above.
- If you are seeing an error `'Python.h' file not found`, then it means you have an incomplete version of Python installed. This is known to sometimes happen on Ubuntu systems. The solution is to run `apt-get install python-dev` or `apt-get install python3.6-dev`.
- If you run into installation errors with `llvmlite` dependency, then your best bet is to attempt to install it manually before trying to build `datatable`:

  ```
  $ pip install llvmlite
  ```

  Consult the `llvmlite` Installation Guide for additional information.
- On OS X, if you are getting an error `fatal error: 'sys/mman.h' file not found` or similar, this can be fixed by installing the Xcode Command Line Tools:

  ```
  $ xcode-select --install
  ```

CHAPTER 4

Contributing

`datatable` is an open source project released under the Mozilla Public Licence v2. Open Source projects live by their user and developer communities. We welcome and encourage your contributions of any kind!

No matter what your skill set or level of engagement is with `datatable`, you can help others by improving the ecosystem of documentation, bug report and feature request tickets, and code.

We invite anyone who is interested to contribute, whether through pull requests, or tests, or GitHub issues, API suggestions, or generic discussion.

CHAPTER 5

Have Questions?

If you have questions about using `datatable`, post them on Stack Overflow using the `[datatable] [python]` tags at http://stackoverflow.com/questions/tagged/datatable+python.

CHAPTER 6

Frame

class datatable.**Frame**

Two-dimensional column-oriented table of data. Each column has its own name and type. Types may vary across columns (unlike in a Numpy array) but cannot vary within each column (unlike in Pandas DataFrame).

Internally the data is stored as C primitives, and processed using multithreaded native C++ code.

This is a primary data structure for datatable module.

cbind()

Append columns of Frames *frames* to the current Frame.

This is equivalent to *pandas.concat(axis=1)*: the Frames are combined by columns, i.e. cbinding a Frame of shape [n x m] to a Frame of shape [n x k] produces a Frame of shape [n x (m + k)].

As a special case, if you cbind a single-row Frame, then that row will be replicated as many times as there are rows in the current Frame. This makes it easy to create constant columns, or to append reduction results (such as min/max/mean/etc) to the current Frame.

If Frame(s) being appended have different number of rows (with the exception of Frames having 1 row), then the operation will fail by default. You can force cbinding these Frames anyways by providing option *force=True*: this will fill all 'short' Frames with NAs. Thus there is a difference in how Frames with 1 row are treated compared to Frames with any other number of rows.

Parameters

- **frames** (*sequence or list of Frames*) – One or more Frame to append. They should have the same number of rows (unless option *force* is also used).
- **force** (*boolean*) – If True, allows Frames to be appended even if they have unequal number of rows. The resulting Frame will have number of rows equal to the largest among all Frames. Those Frames which have less than the largest number of rows, will be padded with NAs (with the exception of Frames having just 1 row, which will be replicated instead of filling with NAs).

colindex()

Return index of the column `name`.

Parameters **name** – name of the column to find the index for. This can also be an index of a column, in which case the index is checked that it doesn't go out-of-bounds, and negative index is converted into positive.

Raises **ValueError** – if the requested column does not exist.

copy()
Make a copy of this Frame.

This method creates a shallow copy of the current Frame: only references are copied, not the data itself. However, due to copy-on-write semantics any changes made to one of the Frames will not propagate to the other. Thus, for all intents and purposes the copied Frame will behave as if it was deep-copied.

countna()
Get the number of NA values in each column.

Returns

- *A new datatable of shape (1, ncols) containing the counted number of NA*
- *values in each column.*

countna1()

head()
Return the first *n* rows of the Frame, same as `self[:n, :]`.

key
Tuple of column names that serve as a primary key for this Frame.

If the Frame is not keyed, this will return an empty tuple.

Assigning to this property will make the Frame keyed by the specified column(s). The key columns will be moved to the front, and the Frame will be sorted. The values in the key columns must be unique.

ltypes
The tuple of each column's ltypes ("logical types")

materialize()

max()
Get the maximum value of each column.

Returns

- *A new datatable of shape (1, ncols) containing the computed maximum*
- *values for each column (or NA if not applicable).*

max1()

mean()
Get the mean of each column.

Returns

- *A new datatable of shape (1, ncols) containing the computed mean*
- *values for each column (or NA if not applicable).*

mean1()

min()
Get the minimum value of each column.

Returns

- *A new datatable of shape (1, ncols) containing the computed minimum*
- *values for each column (or NA if not applicable).*

min1()

mode()
Get the modal value of each column.

Returns

- *A new datatable of shape (1, ncols) containing the computed count of*
- *most frequent values for each column.*

mode1()

names
Tuple of column names.

You can rename the Frame's columns by assigning a new list/tuple of names to this property. The length of the new list of names must be the same as the number of columns in the Frame.

It is also possible to rename just a few columns by assigning a dictionary `{oldname: newname, ...}`. Any column not listed in the dictionary will retain its name.

Examples

```
>>> d0 = dt.Frame([[1], [2], [3]])
>>> d0.names = ['A', 'B', 'C']
>>> d0.names
('A', 'B', 'C')
>>> d0.names = {'B': 'middle'}
>>> d0.names
('A', 'middle', 'C')
>>> del d0.names
>>> d0.names
('C0', 'C1', 'C2')
```

ncols
Number of columns in the Frame

nmodal()
Get the number of modal values in each column.

Returns

- *A new datatable of shape (1, ncols) containing the counted number of*
- *most frequent values in each column.*

nmodal1()

nrows
Number of rows in the Frame.

Assigning to this property will change the height of the Frame, either by truncating if the new number of rows is smaller than the current, or filling with NAs if the new number of rows is greater.

Increasing the number of rows of a keyed Frame is not allowed.

nunique()
Get the number of unique values in each column.

Returns

- *A new datatable of shape (1, ncols) containing the counted number of*
- *unique values in each column.*

nunique1()

rbind(**frames*, *force=False*, *bynames=True*)

Append rows of *frames* to the current Frame.

This is equivalent to *list.extend()* in Python: the Frames are combined by rows, i.e. rbinding a Frame of shape [n x k] to a Frame of shape [m x k] produces a Frame of shape [(m + n) x k].

This method modifies the current Frame in-place. If you do not want the current Frame modified, then append all Frames to an empty Frame: *dt.Frame().rbind(frame1, frame2)*.

If Frame(s) being appended have columns of types different from the current Frame, then these columns will be promoted to the largest of two types: bool -> int -> float -> string.

If you need to append multiple Frames, then it is more efficient to collect them into an array first and then do a single *rbind()*, than it is to append them one-by-one.

Appending data to a Frame opened from disk will force loading the current Frame into memory, which may fail with an OutOfMemory exception.

Parameters

- **frames** (*sequence or list of Frames*) – One or more Frame to append. These Frames should have the same columnar structure as the current Frame (unless option *force* is used).
- **force** (*boolean, default False*) – If True, then the Frames are allowed to have mismatching set of columns. Any gaps in the data will be filled with NAs.
- **bynames** (*boolean, default True*) – If True, the columns in Frames are matched by their names. For example, if one Frame has columns ["colA", "colB", "colC"] and the other ["colB", "colA", "colC"] then we will swap the order of the first two columns of the appended Frame before performing the append. However if *bynames* is False, then the column names will be ignored, and the columns will be matched according to their order, i.e. i-th column in the current Frame to the i-th column in each appended Frame.

replace()

Replace given value(s) *replace_what* with *replace_with* in the entire Frame.

For each replace value, this method operates only on columns of types appropriate for that value. For example, if *replace_what* is a list *[-1, math.inf, None, "??"]*, then the value *-1* will be replaced in integer columns only, *math.inf* only in real columns, *None* in columns of all types, and finally *"??"* only in string columns.

The replacement value must match the type of the target being replaced, otherwise an exception will be thrown. That is, a bool must be replaced with a bool, an int with an int, a float with a float, and a string with a string. The *None* value (representing NA) matches any column type, and therefore can be used as either replacement target, or replace value for any column. In particular, the following is valid: *DT.replace(None, [-1, -1.0, ""])*. This will replace NA values in int columns with *-1*, in real columns with *-1.0*, and in string columns with an empty string.

The replace operation never causes a column to change its logical type. Thus, an integer column will remain integer, string column remain string, etc. However, replacing may cause a column to change its stype, provided that ltype remains constant. For example, replacing *0* with *-999* within an *int8* column will cause that column to be converted into the *int32* stype.

Parameters

- **replace_what** (*None, bool, int, float, list, or dict*) – Value(s) to search for and replace.
- **replace_with** (*single value, or list*) – The replacement value(s). If *replace_what* is a single value, then this must be a single value too. If *replace_what* is a list, then this could be either a single value, or a list of the same length. If *replace_what* is a dict, then this value should not be passed.

Returns

Return type Nothing, replacement is performed in-place.

Examples

```
>>> df = dt.Frame([1, 2, 3] * 3)
>>> df.replace(1, -1)
>>> df.to_list()
[[-1, 2, 3, -1, 2, 3, -1, 2, 3]]
```

```
>>> df.replace({-1: 100, 2: 200, "foo": None})
>>> df.to_list()
[[100, 200, 3, 100, 200, 3, 100, 200, 3]]
```

save(*dest=None*, *format='jay'*, *_strategy='auto'*)
Save Frame in binary NFF/Jay format.

Parameters

- **dest** – destination where the Frame should be saved.
- **format** – either "nff" or "jay"
- **_strategy** – one of "mmap", "write" or "auto"

sd()
Get the standard deviation of each column.

Returns

- *A new datatable of shape (1, ncols) containing the computed standard*
- *deviation values for each column (or NA if not applicable).*

sd1()

shape
Tuple with (nrows, ncols) dimensions of the Frame

stypes
The tuple of each column's stypes ("storage types")

sum()
Get the sum of each column.

Returns

- *A new datatable of shape (1, ncols) containing the computed sums*
- *for each column (or NA if not applicable).*

sum1()

tail()
Return the last *n* rows of the Frame, same as `self[-n:, :]`.

to_csv(*path=''*, *nthreads=0*, *hex=False*, *verbose=False*, ***kwargs*)
Write the Frame into the provided file in CSV format.

Parameters

- **dt** (*Frame*) – Frame object to write into CSV.
- **path** (*str*) – Path to the output CSV file that will be created. If the file already exists, it will be overwritten. If path is not given, then the Frame will be serialized into a string, and that string will be returned.
- **nthreads** (*int*) – How many threads to use for writing. The value of 0 means to use all available threads. Negative values mean to use that many threads less than the maximum available.
- **hex** (*bool*) – If True, then all floating-point values will be printed in hex format (equivalent to %a format in C *printf*). This format is around 3 times faster to write/read compared to usual decimal representation, so its use is recommended if you need maximum speed.
- **verbose** (*bool*) – If True, some extra information will be printed to the console, which may help to debug the inner workings of the algorithm.

to_dict()
Convert the Frame into a dictionary of lists, by columns.

Returns a dictionary with *ncols* entries, each being the *colname: coldata* pair, where *colname* is a string, and *coldata* is an array of column's data.

Examples

```
>>> DT = dt.Frame(A=[1, 2, 3], B=["aye", "nay", "tain"])
>>> DT.to_dict()
{"A": [1, 2, 3], "B": ["aye", "nay", "tain"]}
```

to_list()
Convert the Frame into a list of lists, by columns.

Returns a list of *ncols* lists, each inner list representing one column of the Frame.

Examples

```
>>> DT = dt.Frame(A=[1, 2, 3], B=["aye", "nay", "tain"])
>>> DT.to_list()
[[1, 2, 3], ["aye", "nay", "tain"]]
```

to_numpy(*stype=None*)
Convert Frame into a numpy array, optionally forcing it into a specific stype/dtype.

Parameters **stype** (*datatable.stype, numpy.dtype or str*) – Cast datatable into this dtype before converting it into a numpy array.

to_pandas()
Convert Frame to a pandas DataFrame, or raise an error if *pandas* module is not installed.

to_tuples()

Convert the Frame into a list of tuples, by rows.

Returns a list having *nrows* tuples, where each tuple has length *ncols* and contains data from each respective row of the Frame.

Examples

```
>>> DT = dt.Frame(A=[1, 2, 3], B=["aye", "nay", "tain"])
>>> DT.to_tuples()
[(1, "aye"), (2, "nay"), (3, "tain")]
```

CHAPTER 7

Ftrl

class datatable.models.**Ftrl**

Follow the Regularized Leader (FTRL) model with hashing trick.

See this reference for more details: https://www.eecs.tufts.edu/~dsculley/papers/ad-click-prediction.pdf

Parameters

- **alpha** (*float*) – *alpha* in per-coordinate learning rate formula.
- **beta** (*float*) – *beta* in per-coordinate learning rate formula.
- **lambda1** (*float*) – L1 regularization parameter.
- **lambda2** (*float*) – L2 regularization parameter.
- **nbins** (*int*) – Number of bins to be used after the hashing trick.
- **nepochs** (*int*) – Number of epochs to train for.
- **interactions** (*bool*) – Switch to enable second order feature interactions.

alpha

alpha in per-coordinate FTRL-Proximal algorithm

beta

beta in per-coordinate FTRL-Proximal algorithm

colname_hashes

Column name hashes

feature_importances

One-column frame with the overall weight contributions calculated feature-wise during training and predicting. It can be interpreted as a feature importance information.

fit()

Train an FTRL model on a dataset.

Parameters

- **X** (Frame) – Frame of shape (nrows, ncols) to be trained on.

- **y** (Frame) – Frame of shape (nrows, 1), i.e. the target column. This column must have a *bool* type.

Returns

Return type None

interactions
Switch to enable second order feature interactions

labels
List of labels for multinomial regression.

lambda1
L1 regularization parameter

lambda2
L2 regularization parameter

model
Tuple of model frames. Each frame has two columns, i.e. *z* and *n*, and *nbins* rows, where *nbins* is a number of bins for the hashing trick. Both column types are *float64*.

nbins
Number of bins to be used for the hashing trick

nepochs
Number of epochs to train a model

params
FTRL model parameters

predict()
Make predictions for a dataset.

Parameters **X** (Frame) – Frame of shape (nrows, ncols) to make predictions for. It must have the same number of columns as the training frame.

Returns

- *A new frame of shape (nrows, 1) with the predicted probability*
- *for each row of frame X.*

reset()
Reset FTRL model and feature importance information, i.e. initialize model and importance frames with zeros.

Parameters **None** –

Returns

Return type None

CHAPTER 8

FTRL

This section describes the FTRL (Follow the Regularized Leader) model as implemented in datatable.

8.1 FTRL Model Information

The Follow the Regularized Leader (FTRL) model is a datatable implementation of the FTRL-Proximal online learning algorithm for binomial logistic regression. It uses a hashing trick for feature vectorization and the Hogwild approach for parallelization. FTRL for multinomial classification and continuous targets are implemented experimentally.

8.2 Create an FTRL Model

The FTRL model is implemented as the `Ftrl` Python class, which is a part of `datatable.models`, so to use the model you should first do

```
from datatable.models import Ftrl
```

and then create a model as

```
ftrl_model = Ftrl()
```

8.3 FTRL Model Parameters

The FTRL model requires a list of parameters for training and making predictions, namely:

- `alpha` – learning rate, defaults to `0.005`.
- `beta` – beta parameter, defaults to `1.0`.
- `lambda1` – L1 regularization parameter, defaults to `0.0`.

- `lambda2` – L2 regularization parameter, defaults to `1.0`.
- `nbins` – the number of bins for the hashing trick, defaults to `1000000`.
- `nepochs` – the number of epochs to train the model for, defaults to `1`.
- `interactions` – whether to enable second order feature interactions, defaults to `False`.

If some parameters need to be changed, this can be done either when creating the model, as

```
ftrl_model = Ftrl(alpha = 0.1, nbins = 100, interactions = False)
```

or, if the model already exists, as

```
ftrl_model.alpha = 0.1
ftrl_model.nbins = 100
ftrl_model.interactions = False
```

If some parameters were not set explicitely, they will be assigned the default values.

8.4 Training a Model

Use the `fit()` method to train a model for a binomial logistic regression problem:

```
ftrl_model.fit(X, y)
```

where `X` is a frame of shape `(nrows, ncols)` to be trained on, and `y` is a frame of shape `(nrows, 1)` having a `bool` type of the target column. The following datatable column types are supported for the `X` frame: `bool`, `int`, `real` and `str`.

8.5 Resetting a Model

Use the `reset()` method to reset a model:

```
ftrl_model.reset()
```

This will reset model weights, but it will not affect learning parameters. To reset parameters to default values, you can do

```
ftrl_model.params = Ftrl().params
```

8.6 Making Predictions

Use the `predict()` method to make predictions:

```
targets = ftrl_model.predict(X)
```

where `X` is a frame of shape `(nrows, ncols)` to make predictions for. `X` should have the same number of columns as the training frame. The `predict()` method returns a new frame of shape `(nrows, 1)` with the predicted probability for each row of frame `X`.

8.7 Feature Importances

To estimate feature importances, the overall weight contributions are calculated feature-wise during training and predicting. Feature importances can be accessed as

```
fi = ftrl_model.feature_importances
```

where `fi` will be a frame of shape `(nfeatures, 2)` containing feature names and their importances, that are normalized to [0; 1] range.

8.8 Further Reading

For detailed help, please also refer to `help(Ftrl)`.

Index

A

alpha (*datatable.models.Ftrl attribute*), 27

B

beta (*datatable.models.Ftrl attribute*), 27

C

cbind() (*datatable.Frame method*), 19
colindex() (*datatable.Frame method*), 19
colname_hashes (*datatable.models.Ftrl attribute*), 27
copy() (*datatable.Frame method*), 20
countna() (*datatable.Frame method*), 20
countna1() (*datatable.Frame method*), 20

F

feature_importances (*datatable.models.Ftrl attribute*), 27
fit() (*datatable.models.Ftrl method*), 27
Frame (*class in datatable*), 19
Ftrl (*class in datatable.models*), 27

H

head() (*datatable.Frame method*), 20

I

interactions (*datatable.models.Ftrl attribute*), 28

K

key (*datatable.Frame attribute*), 20

L

labels (*datatable.models.Ftrl attribute*), 28
lambda1 (*datatable.models.Ftrl attribute*), 28
lambda2 (*datatable.models.Ftrl attribute*), 28
ltypes (*datatable.Frame attribute*), 20

M

materialize() (*datatable.Frame method*), 20
max() (*datatable.Frame method*), 20
max1() (*datatable.Frame method*), 20
mean() (*datatable.Frame method*), 20
mean1() (*datatable.Frame method*), 20
min() (*datatable.Frame method*), 20
min1() (*datatable.Frame method*), 21
mode() (*datatable.Frame method*), 21
mode1() (*datatable.Frame method*), 21
model (*datatable.models.Ftrl attribute*), 28

N

names (*datatable.Frame attribute*), 21
nbins (*datatable.models.Ftrl attribute*), 28
ncols (*datatable.Frame attribute*), 21
nepochs (*datatable.models.Ftrl attribute*), 28
nmodal() (*datatable.Frame method*), 21
nmodal1() (*datatable.Frame method*), 21
nrows (*datatable.Frame attribute*), 21
nunique() (*datatable.Frame method*), 21
nunique1() (*datatable.Frame method*), 22

P

params (*datatable.models.Ftrl attribute*), 28
predict() (*datatable.models.Ftrl method*), 28

R

rbind() (*datatable.Frame method*), 22
replace() (*datatable.Frame method*), 22
reset() (*datatable.models.Ftrl method*), 28

S

save() (*datatable.Frame method*), 23
sd() (*datatable.Frame method*), 23
sd1() (*datatable.Frame method*), 23
shape (*datatable.Frame attribute*), 23
stypes (*datatable.Frame attribute*), 23
sum() (*datatable.Frame method*), 23
sum1() (*datatable.Frame method*), 23

T

tail() (*datatable.Frame method*), 23
to_csv() (*datatable.Frame method*), 24
to_dict() (*datatable.Frame method*), 24
to_list() (*datatable.Frame method*), 24
to_numpy() (*datatable.Frame method*), 24
to_pandas() (*datatable.Frame method*), 24
to_tuples() (*datatable.Frame method*), 24