# Motivation

Although Pandas has a spot-on interface and it is full of useful functionalities, it lacks performance and scalability. For example, it is hard to decipher intensive intraday data such as Options data or S&P500 constituents tick-by-tick data using Pandas.
Another issue that I have encountered is, often, the research is done using Python, because it has such tools as Pandas, but the execution in production is in C++ for its efficiency, reliability and scalability. Therefore, there is this translation, or sometimes a bridge, between research and executions.
Also, in this day and age, C++ needs a heterogeneous data container. Mainly because of these factors, I implemented the C++ DataFrame.
*This library is still missing a few functionalities compared with Pandas. It needs more statistical and logical functionalities. I welcome all contributions from people with expertise, interest, and time to do it. I will add more functionalities from time to time, but currently I don't have much free time.*

**Views** were recently added. This is a very interesting/useful concept that even Pandas doesn't have it currently. A view is a slice of a DataFrame that is a reference to the original DataFrame. It appears exactly the same as a DataFrame, but if you modify any data in the view, the original DataFrame will also be modified.
There are certain things you cannot do in views. For example, you cannot add to delete columns, extend the index column, …
For more understanding look at the test file

# Code structure

The DataFrame library is a header-only library with one source file exception, *HeteroVector.cc and HeteroView.cc*.
Starting from the root directory;
*DMScu* is a helper module that contains a few objects. One is a stack-based string object and the other is a Linux-only mmap file interface. These objects are used by the DataFrame library and therefore must be compiled beforehand (see build instructions below).
*include* directory contains most of the code. It includes *.h* and *.tcc* files. The later are C++ template code files. The main header file is *DataFrame.h*. It contains the entire DataFrame object and its interface. There are comprehensive comments for each interface call in that file. The rest of the files there will show you how the sausage is made.
*src* directory has the only source file for the library, make-files, and a test program source file. The test source file is *datasci_tester.cc*. It contains test cases for all functionalities of DataFrame. It is not in a very organized structure. I plan to make the test cases more organized.

# Build Instructions

## Using plain make and make-files

Go to the root of the repository, where license file is, and execute *build_all.sh*. This will build the library and test executables for Linux flavors.

### Using cmake

Please see README file. Thanks to [@justinjk007](), you should be able to build this in Linux, Windows, Mac, and more

## Example

This library is based on a heterogenous vector. The heterogeneity is achieved by using static STL or STL-like vectors. Since C++ is a strongly typed language, you still have to know your column types per container at compile time. You can add more columns with different types at any time to your container, but when analyzing the data at any given time you must know the column types.

This is an example of how to create a DataFrame, load data, and run an operation on it:

```
// Defines a DataFrame with unsigned long index type that used std::vector
typedef DataFrame<unsigned long, std::vector>    MyDataFrame;

MyDataFrame                 df;
std::vector<int>            intvec = { 1, 2, 3, 4, 5 };
std::vector<double>         dblvec = { 1.2345, 2.2345, 3.2345, 4.2345, 5.2345 };
std::vector<double>         dblvec2 = { 0.998, 0.3456, 0.056, 0.15678, 0.00345, 0.923,
0.06743, 0.1 };
std::vector<std::string>.  strvec = { "Col_name", "Col_name", "Col_name", "Col_name",
"Col_name" };
std::vector<unsigned long> ulgvec = { 1UL, 2UL, 3UL, 4UL, 5UL, 8UL, 7UL, 6UL }
std::vector<unsigned long> xulgvec = ulgvec;

// This is only one way of loading data into the DataFrame. There are
// many different ways of doing it. Please see DataFrame.h and datasci_tester.cc
int rc = df.load_data(std::move(ulgvec),
                      std::make_pair("int_col", intvec),
                      std::make_pair("dbl_col", dblvec),
                      std::make_pair("dbl_col_2", dblvec2),
                      std::make_pair("str_col", strvec),
                      std::make_pair("ul_col", xulgvec));
// Sort the Frame by index
df.sort<MyDataFrame::TimeStamp, int, double, std::string>();
//Sort the Frame by column "dbl_col_2"
df.sort<double, int, double, std::string>("dbl_col_2");

// A functor to calculate mean, variance, skew, kurtosis, defined in DFVisitors.h file
StatsVisitor<double>  stats_visitor;
// Calculate the stats on column "dbl_col"
df.visit<double>("dbl_col", stats_visitor);
```

**View Example:**

```
std::vector<unsigned long>  idx =
                { 123450, 123451, 123452, 123450, 123455, 123450, 123449 };
std::vector<double> d1 = { 1, 2, 3, 4, 5, 6, 7 };
std::vector<double> d2 = { 8, 9, 10, 11, 12, 13, 14 };
std::vector<double> d3 = { 15, 16, 17, 18, 19, 20, 21 };
std::vector<double> d4 = { 22, 23, 24, 25 };
std::vector<std::string> s1 = { "11", "22", "33", "xx", "yy", "gg", "string" };
MyDataFrame df;

df.load_data(std::move(idx),
             std::make_pair("col_1", d1),
             std::make_pair("col_2", d2),
             std::make_pair("col_3", d3),
```

```
                std::make_pair("col_4", d4),
                std::make_pair("col_str", s1));

typedef DataFrameView<unsigned long> MyDataFrameView;

MyDataFrameView  dfv = df.get_view_by_loc<double, std::string>({ 3, 6 });

dfv.get_column<double>("col_3")[0] = 88.0;
std::cout << "After changing a value on view: "
          << dfv.get_column<double>("col_3")[0]
          << " == " << df.get_column<double>("col_3")[3]
          << std::endl;
```

For more examples see file *datasci_testr.cc*

## Types

using size_type = std::vector<DataVec>::size_type;
size_type is the size type

using TimeStamp = TS;
TimeStamp is the type of the index column

using TSVec = std::vector<TS>;
TSVec is the type of the vector containing the index column

```
enum class nan_policy : bool  {
   pad_with_nans = true,
   dont_pad_with_nans = false
};
```

Enumerated type of Boolean type to specify whether data should be padded with NaN or not

```
enum class sort_state : bool  {
   sorted = true,
   not_sorted = false
};
```

Enumerated type of Boolean type to specify whether data is sorted or not

```
template<typename T>
struct Index2D  {
   T   begin {};
   T   end {};
};
```

It represents a range with begin and end within a continuous memory space

```
template<typename T, typename U>
struct type_declare;
```

```
template<typename U>
```

```
struct type_declare<HeteroVector, U>  { using type = std::vector<U>; };

template<typename U>
struct type_declare<HeteroView, U>  { using type = VectorView<U>; };
```
This is a spoofy way to declare a type at compile time dynamically. Here it is used in declaring a few different data structures depending whether we are a DataFrame or DataFrameView

```
template<typename TS, typename HETERO>
class DataFrame;

template<typename TS>
using StdDataFrame = DataFrame<TS, HeteroVector>;

template<typename TS>
using DataFrameView = DataFrame<TS, HeteroView>;
```
DataFrame is a class that has; An index column of type TS (timestamp, although it doesn't have to be time), and many other columns of different types. The storage used throughout is std::vector.
DataFrames could be instantiated in two different modes:
*StdDataFrame* is the standard fully functional data-frame.
*DataFrameView* is a referenced to a slice of another data-frame. Most of the functionalities of *StdDataFrame* is also available on the *DataFrameView*. But some functionalities such as adding/removing columns etc. are not allowable on views. If you change any of the data in a *DataFrameView* the corresponding data in the original *StdDataFrame* will also be changed.

## Methods

```
template<typename T>
std::vector<T> &create_column(const char *name);
```
It creates an empty column named "name"
T: Type of the column
Returns a reference to the vector for that column

```
void remove_column(const char *name);
```
It removes a column named name.
The actual data vector is not deleted, but the column is dropped from DataFrame

```
void rename_column (const char *from, const char *to);
```
It renames column named from to to. If column from does not exists, it throws an exception

```
template<typename ... Ts>
size_type &load_data(TSVec &&indices, Ts ... args);
```

This is the most generalized load function. It creates and loads an index and a variable number of columns. The index vector and all column vectors are "moved" to DataFrame.

Ts: The list of types for columns in args
indices: A vector of indices (timestamps) of type TimeStamp;
args: A variable list of arguments consisting of
    std::pair(<const char *name, std::vector<T> &&data>).
    Each pair represents a column data and its name
Returns number of items loaded

```
template<typename ITR>
size_type load_index(const ITR &begin, const ITR &end);
```

It copies the data from iterators begin to end into the index column
ITR: Type of the iterator
Returns number of items loaded

```
template<typename ITR>
size_type load_index(const ITR &begin, const ITR &end);
```

It copies the data from iterators begin to end into the index column
ITR: Type of the iterator
Returns number of items loaded

```
size_type load_index(TSVec &&idx);
```

It moves the idx vector into the index column.
Returns number of items loaded

```
template<typename T, typename ITR>
size_type load_column(const char *name,
                      Index2D<const ITR &> range,
                      nan_policy padding = nan_policy::pad_with_nans);
```

It copies the data from iterators begin to end to the named column. If column does not exist, it will be created. If the column exist, it will be over written.
T: Type of data being copied
ITR: Type of the iterator
name: Name of the column
range: The begin and end iterators for data
padding: If true, it pads the data column with nan if it is shorter than the
    index column.
Returns number of items loaded

```
template<typename T>
size_type
load_column(const char *name,
            std::vector<T> &&data,
            nan_policy padding = nan_policy::pad_with_nans);
```

It moves the data to the named column in DataFrame. If column does not exist, it will be created. If the column exist, it will be over written.
T: Type of data being moved
name: Name of the column
padding: If true, it pads the data column with nan,
if it is shorter than the index column.
Returns number of items loaded

```
size_type append_index(const TimeStamp &val);
```
It appends val to the end of the index column.
Returns number of items loaded

```
template<typename T>
size_type append_column(const char *name,
                        const T &val,
                        nan_policy padding = nan_policy::pad_with_nans);
```
It appends val to the end of the named data column. If data column doesn't exist, it throws an exception.
T: Type of the named data column
name: Name of the column
padding: If true, it pads the data column with nan,
if it is shorter than the index column.
Returns number of items loaded

```
template<typename ITR>
size_type append_index(Index2D<const ITR &> range);
```
It appends the range begin to end to the end of the index column
ITR: Type of the iterator
range: The begin and end iterators for data
Returns number of items loaded

```
template<typename T, typename ITR>
size_type append_column(const char *name,
                        Index2D<const ITR &> range,
                        nan_policy padding = nan_policy::pad_with_nans);
```
It appends the range begin to end to the end of the named data column. If data column doesn't exist, it throws an exception.
T: Type of the named data column
ITR: Type of the iterator
name: Name of the column
range: The begin and end iterators for data
padding: If true, it pads the data column with nan,
if it is shorter than the index column.
Returns number of items loaded

```
template<typename ... types>
```

```
void remove_data_by_idx (Index2D<TS> range);
```

It removes the data rows from index begin to index end.
DataFrame must be sorted by index or behavior is undefined.
This function first calls make_consistent() that may add nan values to data columns.
types: List all the types of all data columns.
A type should be specified in the list only once.
range: The begin and end iterators for index specified with index values

```
template<typename ... types>
void remove_data_by_loc (Index2D<int> range);
```

It removes the data rows from location begin to location end within range.
This function supports Python-like negative indexing. That is why the range type is int.
This function first calls make_consistent() that may add nan values to data columns.
types: List all the types of all data columns.
A type should be specified in the list only once.
range: The begin and end iterators for data

```
template<typename ... types>
void make_consistent ();
```

Make all data columns the same length as the index. If any data column is shorter than the index column, it will be padded by nan.
This is also called by sort(), before sorting

```
template<typename T, typename ... types>
void sort(const char *by_name = nullptr);
```

Sort the DataFrame by the named column. By default, it sorts by index (i.e. by_name == nullptr). Sort first calls make_consistent() that may add nan values to data columns. nan values make sorting nondeterministic.
T: Type of the by_name column. You always of the specify this type, even if it is being sorted to the default index
types: List all the types of all data columns.
A type should be specified in the list only once.

```
template<typename T, typename ... types>
std::future<void> sort_async (const char *by_name = nullptr);
```

Same as sort() above, but executed asynchronously

```
template<typename F, typename T, typename ... types>
DataFrame groupby (F &&func,
                   const char *gb_col_name = nullptr,
                   sort_state already_sorted = sort_state::not_sorted) const;
```

Groupby copies the DataFrame into a temp DataFrame and sorts the temp df by gb_col_name before performing groupby. If gb_col_name is null, it groups by index.
F: type functor to be applied to columns to group by
T: type of the groupby column. In case if index, it is type of index
types: List of the types of all data columns.
A type should be specified in the list only once.
func: The functor to do the groupby. Specs for the functor is
in a separate doc.
already_sorted: If the DataFrame is already sorted by gb_col_name,
this will save the expensive sort operation

```
std::future<DataFrame>
groupby_async (F &&func,
               const char *gb_col_name = nullptr,
               sort_state already_sorted = sort_state::not_sorted) const;
```

Same as groupby() above, but executed asynchronously

```
template<typename F, typename ... types>
DataFrame bucketize (F &&func, const TimeStamp &bucket_interval) const;
```

It bucketizes the data and index into bucket_interval's, based on index values and calls the functor for each bucket. The result of each bucket will be stored in a new DataFrame with same shape and returned. Every data bucket is guaranteed to be as wide as bucket_interval. This mean some data items at the end may not be included in the new bucketized DataFrame. The index of each bucket will be the last index in the original DataFrame that is less than bucket_interval away from the previous bucket

NOTE: The DataFrame must already be sorted by index.

F: type functor to be applied to columns to bucketize
types: List of the types of all data columns.
A type should be specified in the list only once.
bucket_interval: Bucket interval is in the index's single value unit.
For example, if index is in minutes, bucket_interval
will be in the unit of minutes and so on.
already_sorted: If the DataFrame is already sorted by index,
this will save the expensive sort operation

```
template<typename F, typename ... types>
std::future<DataFrame>
bucketize_async (F &&func, const TimeStamp &bucket_interval) const;
```

Same as bucketize() above, but executed asynchronously

```
template<typename F, typename ... types>
void self_bucketize (F &&func, const TimeStamp &bucket_interval);
```

This is exactly the same as bucketize() above. The only difference is it stores the result in itself and returns void. So, after the return the original data is lost and replaced with bucketized data

template<typename T, typename V>
DataFrame transpose(TSVec &&indices, const V &col_names) const;
It transposes the data in the DataFrame.
The transpose() is only defined for DataFrame's that have a single data type
T: The single type for all data columns
V: The type of string vector specifying the new names for new columns after transpose
idx: A vector on indices/timestamps for the new transposed DataFrame. Its length must equal the number of rows in this DataFrame. Otherwise an exception is thrown
col_names: A vector of strings, specifying the column names for the new transposed DataFrame. Its length must equal the number of rows in this DataFrame. Otherwise an exception is thrown

template<typename S, typename ... types>
bool write (S &o, bool values_only = false) const;
It outputs the content of DataFrame into the stream o as text in the following format:

INDEX:<Comma delimited list of values>
<Column1 name>:<Column1 type>:<Comma delimited list of values>
<Column2 name>:<Column2 type>:<Comma delimited list of values>

S: Output stream type
types: List all the types of all data columns. A type should be specified in the list only once.
o: Reference to an streamable object (e.g. cout)
values_only: If true, the name and type of each column is not written

template<typename S, typename ... Ts>
std::future<bool> write_async (S &o, bool values_only = false) const;
Same as write() above, but executed asynchronously

bool read (const char *file_name);
It inputs the contents of a text file into itself (i.e. DataFrame). The format of the file must be:

INDEX:<Comma delimited list of values>
<Column1 name>:<Column1 type>:<Comma delimited list of values>
<Column2 name>:<Column2 type>:<Comma delimited list of values>

All empty lines or lines starting with # will be skipped.
file_name: Complete path to the file

```
std::future<bool> read_async (const char *file_name);
```

Same as read() above, but executed asynchronously

```
template<typename T>
typename type_declare<HETERO, T>::type &
get_column (const char *name);
```

It returns a reference to the container of named data column
The return type depends on if we are in standard or view mode
T: Data type of the named column

```
template<typename T>
const typename type_declare<HETERO, T>::type &
get_column (const char *name) const;
```

It returns a const reference to the container of named data column
The return type depends on if we are in standard or view mode
T: Data type of the named column

```
template<typename T>
std::vector<T> get_col_unique_values (const char *name) const;
```

It returns a vector of unique values in the named column in the same order that exists in the column.
For this method to compile and work, 3 conditions must be met:
1) Type T must be hash-able. If this is a user defined type, you must enable and specialize std::hash.
2) The equality operator (==) must be well defined for type T.
3) Type T must match the actual type of the named column.
Of course, if you never call this method in your application, you need not be worried about these conditions.

T: Data type of the named column

```
template<typename ... types>
DataFrame get_data_by_idx (Index2D<TS> range) const;
```

It returns a DataFrame (including the index and data columns) containing the data from index begin to index end. This function assumes the DataFrame is consistent and sorted by index. The behavior is undefined otherwise.
types: List all the types of all data columns.
A type should be specified in the list only once.
range: The begin and end iterators for index specified with index values

```
template<typename ... types>
DataFrameView<TS> get_view_by_idx (Index2D<TS> range) const;
```

It behaves like get_data_by_idx(), but it returns a DataFrameView.
A view is a DataFrame that is a reference to the original DataFrame.
So if you modify anything in the view the original DataFrame will also be modified.

Note: There are certain operations that you cannot do with a view.
For example, you cannot add/delete columns, etc.

types: List all the types of all data columns.
A type should be specified in the list only once.
range: The begin and end iterators for index specified with index values

```
template<typename ... types>
DataFrame get_data_by_loc (Index2D<int> range) const;
```

It returns a DataFrame (including the index and data columns) containing the data from location begin to location end.
This function supports Python-like negative indexing. That is why the range type is int.
This function assumes the DataFrame is consistent and sorted by index. The behavior is undefined otherwise.
types: List all the types of all data columns.
A type should be specified in the list only once.
range: The begin and end iterators for data

```
template<typename ... types>
DataFrameView<TS> get_view_by_loc (Index2D<int> range) const;
```

It behaves like get_data_by_loc(), but it returns a DataFrameView.
A view is a DataFrame that is a reference to the original DataFrame.
So if you modify anything in the view the original DataFrame will also be modified.
Note: There are certain operations that you cannot do with a view.
For example, you cannot add/delete columns, etc.

types: List all the types of all data columns.
A type should be specified in the list only once.
range: The begin and end iterators for data

```
const TSVec &get_index () const  { return (timestamps_); }
```

It returns a const reference to the index container

```
TSVec &get_index ()  { return (timestamps_); }
```

It returns a reference to the index container

```
template<typename ... Ts>
void multi_visit (Ts ... args) const;
```

This is the most generalized visit function. It visits multiple columns with the corresponding function objects sequentially. Each function object is passed every single value of the given column along with its name and the corresponding index value. All functions objects must have this signature

```
bool (const TimeStamp &i, const char *name, const T &col_value)
```

If the function object returns false, the DataFrame will not go on that column.

Ts: The list of types for columns in args
args: A variable list of arguments consisting of
std::pair(<const char *name,
&std::function<bool (const TimeStamp &,
const char *, const T &)>>).
Each pair represents a column name and the functor to run on it.
NOTE: The second member of pair is a _pointer_ to the function or functor object

```
template<typename T, typename V>
V &visit (const char *name, V &visitor) const;
```

It passes the values of each index and each named column to the functor visitor sequentially from beginning to end
T: Type of the named column
V: Type of the visitor functor
name: Name of the data column

```
template<typename T1, typename T2, typename V>
V &&visit (const char *name1, const char *name2, V &&visitor) const;
```

It passes the values of each index and the two named columns to the functor visitor sequentially from beginning to end
T1: Type of the first named column
T2: Type of the second named column
V: Type of the visitor functor
name1: Name of the first data column
name2: Name of the second data column

```
template<typename T1, typename T2, typename T3, typename V>
V &&visit (const char *name1,
           const char *name2,
           const char *name3,
           V &&visitor) const;
```

It passes the values of each index and the three named columns to the functor visitor sequentially from beginning to end
T1: Type of the first named column
T2: Type of the second named column
T3: Type of the third named column
V: Type of the visitor functor
name1: Name of the first data column
name2: Name of the second data column
name3: Name of the third data column

```
template<typename T1, typename T2, typename T3, typename T4, typename V>
V &&visit (const char *name1,
           const char *name2,
           const char *name3,
```

```
const char *name4,
V &&visitor) const;
```

It passes the values of each index and the four named columns to the functor visitor sequentially from beginning to end

T1: Type of the first named column
T2: Type of the second named column
T3: Type of the third named column
T4: Type of the forth named column
V: Type of the visitor functor
name1: Name of the first data column
name2: Name of the second data column
name3: Name of the third data column
name4: Name of the forth data column

```
template<typename T1, typename T2, typename T3, typename T4, typename T5,
         typename V>
V &&visit (const char *name1,
           const char *name2,
           const char *name3,
           const char *name4,
           const char *name5,
           V &&visitor) const;
```

It passes the values of each index and the five named columns to the functor visitor sequentially from beginning to end

T1: Type of the first named column
T2: Type of the second named column
T3: Type of the third named column
T4: Type of the forth named column
T5: Type of the fifth named column
V: Type of the visitor functor
name1: Name of the first data column
name2: Name of the second data column
name3: Name of the third data column
name4: Name of the forth data column
name5: Name of the fifth data column

```
template<typename ... types>
bool is_equal (const DataFrame &rhs) const;
```

It compares self with rhs. If both have the same indices, same number of columns, same names for each column, and all columns are equal, then it returns true. Otherwise it returns false

types: List all the types of all data columns.
    A type should be specified in the list only once.

```
template<typename ... types>
```

```
DataFrame &modify_by_idx (DataFrame &rhs,
                          sort_state already_sorted = sort_state::not_sorted);
```

It iterates over all indices in rhs and modifies all the data columns in self that correspond to the given index value. If not already_sorted, both rhs and self will be sorted by index. It returns a reference to self

types: List all the types of all data columns.
A type should be specified in the list only once.

already_sorted: If the self and rhs are already sorted by index,
this will save the expensive sort operations

## Data Frame built-in Visitors

```
template<typename T, typename TS_T = unsigned long>
struct MeanVisitor;
```

This functor class calculates the mean of a given column. See this document and datasci_tester.cc for examples.

T: Column/data type

TS_T: Index type

```
template<typename T,
         typename TS_T = unsigned long,
         typename = typename std::enable_if<std::is_arithmetic<T>::value, T>::type>
struct SumVisitor;
```

This functor class calculates the sum of a given column. See this document and datasci_tester.cc for examples.

T: Column/data type

TS_T: Index type

T must be an arithmetic-enabled type

```
template<typename T,
         typename TS_T = unsigned long,
         typename = typename std::enable_if<std::is_arithmetic<T>::value, T>::type>
struct MaxVisitor;
```

This functor class calculates the maximum of a given column. See this document and datasci_tester.cc for examples.

T: Column/data type

TS_T: Index type

T must be an arithmetic-enabled type

```
template<typename T,
         typename TS_T = unsigned long,
         typename = typename std::enable_if<std::is_arithmetic<T>::value, T>::type>
struct MinVisitor;
```

This functor class calculates the minimum of a given column. See this document and datasci_tester.cc for examples.

T: Column/data type

TS_T: Index type
T must be an arithmetic-enabled type

```
template<typename T,
         typename TS_T = unsigned long,
         typename = typename std::enable_if<std::is_arithmetic<T>::value, T>::type>
struct CovVisitor;
```

This functor class calculates the covariance of two given columns. In addition, it provides the variances of both columns
See this document and datasci_tester.cc for examples.
T: Column/data type
TS_T: Index type
T must be an arithmetic-enabled type

```
template<typename T,
         typename TS_T = unsigned long,
         typename = typename std::enable_if<std::is_arithmetic<T>::value, T>::type>
struct VarVisitor;
```

This functor class calculates the variance of a given column. See this document and datasci_tester.cc for examples.
T: Column/data type
TS_T: Index type
T must be an arithmetic-enabled type

```
template<typename T,
         typename TS_T = unsigned long,
         typename = typename std::enable_if<std::is_arithmetic<T>::value, T>::type>
struct StdVisitor;
```

This functor class calculates the standard deviation of a given column. See this document and datasci_tester.cc for examples.
T: Column/data type
TS_T: Index type
T must be an arithmetic-enabled type

```
template<typename T,
         typename TS_T = unsigned long,
         typename = typename std::enable_if<std::is_arithmetic<T>::value, T>::type>
struct CorrVisitor;
```

This functor class calculates the correlation of two given columns. See this document and datasci_tester.cc for examples.
T: Column/data type
TS_T: Index type
T must be an arithmetic-enabled type

```
template<typename T,
         typename TS_T = unsigned long,
         typename = typename std::enable_if<std::is_arithmetic<T>::value, T>::type>
```

```
struct DotProdVisitor;
```

This functor class calculates the dot-product of two given columns. See this document and datasci_tester.cc for examples.
T: Column/data type
TS_T: Index type
T must be an arithmetic-enabled type

```
template<typename T,
         typename TS_T = unsigned long,
         typename = typename std::enable_if<std::is_arithmetic<T>::value, T>::type>
struct StatsVisitor;
```

This functor class calculates the following statistics of a given column; mean, variance, standard deviation, skew, and kurtosis. See this document and datasci_tester.cc for examples.
T: Column/data type
TS_T: Index type
T must be an arithmetic-enabled type

```
template<typename T,
         typename TS_T = unsigned long,
         typename = typename std::enable_if<std::is_arithmetic<T>::value, T>::type>
struct SLRRegressionVisitor;
```

This functor class calculates simple linear regression, in one pass, of two given columns (x, y). See this document and datasci_tester.cc for examples.
T: Column/data type
TS_T: Index type
T must be an arithmetic-enabled type