Tutorial of Frovedis

1. Introduction

This document is a tutorial of Frovedis.

It provides

- Spark-like API (the core functinality of Frovedis)
- Matrix library using above API
- Machine learning algorithm library
- Dataframe for preprocessing

Under the hood, it is an MPI program. Users can use above functionalities with MPI. Unlike Spark, it does not provide out-of-core functionality; the data should fit in memory. However, it should work much faster than Spark because of this.

In addition, Frovedis does not provide fault-tolerant functionality directly. If you need it, please use checkpointing mechanism provided by MPI implementation.

From the following subsections, each functionalities will be explained together with examples.

Please note that some of the functionalities are not vectorized; if you try Frovedis on SX-Aurora TSUBASA or SX-ACE, please check if the functionality you want to use is vectorized or not.

2. Frovedis core

2.1 Simple example

Let's start from a simple program. I assume that Frovedis is successfully installed.

Please go to the ./src/tut2.1 directory and type "make".

(The Makefile is a symblic link to ../Makefile.each. If you are trying the tutorial in the source code tree, please make Makefile.each as a symblic link of Makefile.each.[x86, etc.] according to your architecture.)

If the program is successfully compiled, a binary named "tut" is created. If you run the program like

```
$ mpirun -np 4 ./tut
```

would produce an output like

```
2
4
6
```

```
8
10
12
14
16
```

2.2 Line by line explanation

Then, let's look at the code (tut.cc).

```
#include <frovedis.hpp>
```

you need include `<frovedis.hpp>` to use core.

```
frovedis::use_frovedis use(argc, argv);
```

This is the initialization and finalization of Frovedis, which is implemented in RAII manner: initialization is done as the constructor of the class; finalization is done as the destructor of the class.

It takes argc and argv as the arguments to parse the command line arguments.

```
auto d1 = frovedis::make_dvector_scatter(v);
```

The type of d1 is `frovedis::dvector<int>`. dvector stands for distributed vector. This function takes `vector<T>` and distributes (scatters) the contents into workers. In this case, if number of MPI processes (workers) is 4, {1, 2} is stored in the rank 0 (worker 0), and {3, 4} is stored in the rank 1 (worker 1), and so on.

```
auto d2 = d1.map(two_times);
```

`dvector<T>` has several member functions; "map" is one of them. It takes a function and applies the function to the each item of the dvector in parallel at the workers. Then a new dvector is created from the return value of the function. The type of argument of the function should be the same as the type of dvector. (The return type can be any type; the returned dvector becomes of that type.)

In this case, the function "two_times" takes int, doubles it, and returns the value. Therefore, the system doubles the each element of d1 in parallel and returns new `dvector<int>` d2. If number of workers is 4, as for d2, worker 0 should have {2, 4}, worker 1 should have {6, 8}, and so on.

```
auto r = d2.gather();
```

Another member function "gather" gathers the distributed data and stores it into the `vector<T>`; in this case, the type of "r" is `vector<int>`. Then the program outputs the contents of "r".

2.3 What is happening under the hood?

The program works as an MPI program. Execution of each node branches at the time of initialization.

The initialization is done in the constructor of the variable `use`. After calling `MPI_Init`, rank 0 returns from the constructor and goes through the next statement to continue the execution. On the other hand, other nodes wait for RPC (remote procedure call) requests from rank 0 in the constructor and never return.

In the above example, `map` and `gather` are called only by rank 0. In the functions, rank 0 sends RPC requests to other ranks to process the parts assigned to them.

Finalization is done in the destructor of the variable `use`, which is called at rank 0. Rank 0 sends finalization request to all the other nodes and calls `MPI_Finalize`. When other nodes receive the finalization request, they call `MPI_Finalize` and exit the program. (Underlying RPC library is not exposed to users.)

As you can see in the Makefile, all the include file is in ${INSTALLPATH}/include/frovedis. The file frovedis.hpp includes basic part of Frovedis, including dvector, dunordered_map, and node_local. If you want to use other modules like matrix library or machine learning library, you need to add other include files.

As for the library to link, they are in ${INSTALLPATH}/lib. You need to link at least libfrovedis_core.a (-lfrovedis_core). If you want to use matrix library, you need to link libfrovedis_matrix.a. If you want to use machine learning library, you need to link libfrovedis_ml.a. If you want to use dataframe library, you need to link libfrovedis_dataframe.a.

2.4 dvector in detail

So far, I explained the system using a simple program. But Frovedis supports more (member) functions to write more interesting programs. From now on, I will explain these functionalities. In this section, I will explain functionalities related to dvector.

dvector can be considered as the distributed version of vector. Memory management is similar to vector (RAII): when a dvector is destructed, the related distributed data is deleted at the time. You can copy or construct it from an existing dvector. In this case, distributed data is also copied (If the source variable is rvalue, the system tries to avoid copy).

Since the memory management is based on RAII, the intermediate data created by "method chain" is automatically destructed. For example,

```
d2 = d1.map(some_func1).map(some_func2)
```

The dvector created by map(some_func1) is destructed after calling map(some_func2).

2.4.1 loading / saving data

In the previous example, `make_dvector_scatter` is used to create dvector. You can create dvector by loading a data from a file, and save it to a file. Please go to "src/tut2.4.1-1" and make the program. Running "./tut" will produce "./result" file.

```
auto d1 = frovedis::make_dvector_loadline("./data");
```

Here, make_dvector_loadline("./data") loads data from a file "./data". The contents of the file is delimited by line and stored into `dvector<std::string>`. Here, the contents of "./data" is

```
1
2
...
8
```

So d1 is "1", "2", . . . , "8".

```
auto d2 = d1.map(parse_data).map(two_times);
```

Here, parse_data takes string and converts it to int and returns it. The function "map(two_times)" is applied to the result, which should produce the same dvector as the previous example.

By calling

```
d2.saveline("./result");
```

the result is saved into "./result" file. The contents of the file should be:

```
2
4
...
16
```

Both make_dvector_loadline and saveline are executed on workers in parallel using MPI-IO. So if you run the program in the distributed environment, please confirm that the file can be accessed by the worker.

(If you are using NFS, there might be a problem when saving a file, because it does not support parallel write. If envirnment variable `FROVEDIS_SEQUENTIAL_SAVE` is set to `true`, save is done sequentially to avoid this problem. This is set by default if you are using rpm distribution and using x86env.sh or veenv.sh to set environment.)

In this example, we explicitly parsed the contents using a function `parse_data`. However, we provide a functionality to parse numeric data at the time of loading. In this case,

```
auto d1 = frovedis::make_dvector_loadline<int>("./data");
auto d2 = d1.map(two_times);
```

produces the same result (they are in tut.cc as comments). Please note `<int>` after frovedis::make_dvector_loadline. If there is no template parameter, it loads the lines as string by default. If there is a type like int, double, etc. it parses the line and produces that type of dvector. The format of the data should be one data per each line.

The function "make_dvector_loadline" and the member function "saveline" requires parsing or creating text. On the other hand, you might want to skip them using binary input / output.

For that purpose, we provide load / save for binary numeric data. Please go to "src/tut2.4.1-2".

```
auto d1 = frovedis::make_dvector_loadbinary<int>("./data_binary");
```

This loads binary data "data_binary" and stores into d1. Here, the format is *little endian* for compatibility between x86 and SX-Aurora TSUBASA. (in the older version of Frovedis, it was big endian.) For example, in this case,

```
$ od -t x1 data_binary
0000000 01 00 00 00 02 00 00 00 03 00 00 00 04 00 00 00
0000020 05 00 00 00 06 00 00 00 07 00 00 00 08 00 00 00
```

The option of od `-t x1` means that it shows the result per byte using hexadecimal. Here the stored data is 4 bytes int. So the first data is "01 00 00 00" that means 1.

You always need `<int>` to make_dvector_loadbinary to specifiy the type of the loading data.

```
d2.savebinary("./result_binary");
```

This saves the result, which should be like this:

```
$ od -t x1 result_binary
0000000 02 00 00 00 04 00 00 00 06 00 00 00 08 00 00 00
0000020 0a 00 00 00 0c 00 00 00 0e 00 00 00 10 00 00 00
```

In both make_dvector_loadbinary and savebinary, MPI-IO is used to load or save the data in parallel.

To convert between text and binary data, there are tools in samples/matrix_tools/. You can use `to_binary_vector` or `to_text_vector`. If you want to convert endian, you can use `convert_endian`.

2.4.2 map

In this section, I will explain map in more detail.

So far, we have used a function as the argument of map. You can use a function object instead of a function. Please go to "src/tut2.4.2-1" and look at tut.cc.

```
struct n_times {
  n_times(){}
  n_times(int n_) : n(n_) {}
  int operator()(int i){return i*n;}
  int n;
  SERIALIZE(n)
};
```

We have the above definition of function object instead of a function. It has int n as the internal state, and operator() returns the argument * n.

The special thing of this function object is that it should be possible to serialize it with cereal (or boost::serialization). Since it is tedious to write additional code for serialization, we provide macro for this purpose: in this case, with SERIALIZE(n). This macro can be used for multiple arguments, like SERIALIZE(n, other_variable), etc. If there is nothing to serialize, use SERIALIZE_NONE. Since serialization libraries require default constructor, it is also added to the structure.

You can always use a function object where a function is used, like flat_map, filter and so on, which will be explained later.

Actually, all variables that is used in this framework should be able to be serialized with the serialization libraries. STL containers and PODs like int, double can be serialized by default. But if you define a data structure by yourself, please do not forget to add SERIALIZE or SERIALIZE_NONE.

The map itself is called like this:

```
auto d2 = d1.map<int>(n_times(3));
```

The argument of map is the created function object. Here, please note that `<int>` is added to map. In the previous example, the type of returning value can be inferred from the type of the pointer to the function, but in the case of function object, it is difficult. Therefore, `<int>` needs to be added to map.

(You might experiense weird compilation error when using this kind of template parameter. This happens when the type of the variable (in this case d1) is *dependent type*; that is, it depends on template parameter and cannot be known by the compiler at the time of parsing. In this case, the compiler cannot tell if `<` is less-than operator or not. If this happens, please add `template` before map, like `d1.template map<int>(n_times(3))`)

Using function object is a bit faster than using a function, because it can be inline-expanded. On SX, it might become much faster, because in the case of function pointer, the loop cannot be vectorized, but using function object makes it possible to vectorize the loop.

If you want to vectorize the loop by using function, please use map_partitions or node_local that will be explained later.

In addition, using function object is sometimes convenient to distribute data together with the function.

Then, Please go to "src/tut2.4.2-2" and look at tut.cc. So far, the function took "value" of the type, but it is possible to take the reference of the type.

```
int two_times(int& i) {i*=2; return 0;}
```

In this case, the argument is reference and the function updates the data given as the argument. The returning value is just dummy. You can check that the original data is actually updated by running the program.

However, what you want to do is just updating the original data, returning dummy data is not efficient. We provide another version of map, which is called "mapv", to support void function. Please go to "src/tut2.4.2-3" and look at tut.cc.

```
void two_times(int& i) {i*=2;}
```

is the calling function; "mapv" can be used just like normal map. Here, mapv returns the reference to the original dvector to support method chain. In the example,

```
auto r = d1.mapv(two_times).gather();
```

demonstrates the functionality.

You might wonder if it is possible to use lambda expression with map. It is possible if it does not capture variable. Please see "src/tut2.4.2-4/tut.cc".

```
auto d2 = d1.map(+[](int i){return i*2;});
```

Here, `[](int i){return i*2;}` is lambda expresson that doubles the input. Since it does not capture any variables, it can be converted to function pointer. You can do it by adding `+` before the lambda expression; this is just a unary operator `+`, which can be used to "decay" the type to function pointer. Now it works just like giving a function pointer to map.

2.4.3 flat_map

Sometimes, you might want to output multiple values from one function. For example, in the case of "word count", you would like to output multiple {word, count} pairs for each line. The member function flat_map can be used for this purpose.

Example of flat_map is shown in "src/tut2.4.3". It simply duplicates the input argument.

```
std::vector<int> duplicate(int i) {
  std::vector<int> v;
  v.push_back(i); v.push_back(i);
  return v;
}
```

Output of the function should be `vector<T>`.

```
auto d2 = d1.flat_map(duplicate);
```

Here, the output vectors are "flattened" into a `dvector<int>`; the size of d2 is twice as large as d1.

2.4.4 filter

You might want to select data that have some specific characteristics. You can use filter for this purpose. Please go to "src/tut2.4.4-1" and look at tut.cc.

Filtering function can be written as like this:

```
bool is_even(int i) {return i % 2 == 0;}
```

Then,

```
auto d2 = d1.filter(is_even);
```

selects only even data and stores it into new dvector. The result should be like

```
2
4
6
8
```

If you want to update the original data, instead of creating a new data, you can use inplace_filter. The example can be found in "src/tut2.4.4-2".

2.4.5 reduce

If you want to sum up whole data of dvector, you can use reduce for this purpose. Please go to "src/tut2.4.5" and look at tut.cc.

```
int sum(int a, int b){return a + b;}
```

This is a function used for reduce. In this case, it is just an addition. You can specify any operations that satisfy the associative law, like min, max, multiply.

```
std::cout << d1.reduce(sum) << std::endl;
```

This sums up all the elements of d1. The result should be 36 in this case.

2.4.6 put/get

You might want to put or get each item of the dvector using index. We support it by put and get method.

However, putting or getting one data takes network communication latency, *so it would be very slow if you use put/get in a long running loop.* Please do not use it if performance is critical. On the other hand, it is convenient for debugging purpose, etc. You can find the example in "src/tut2.4.6".

```
std::cout << d1.get(3) << std::endl;
```

Here, d1.get(3) returns 4, because the origin of index is 0.

```
d1.put(3, 40);
```

This changes the value of index 3 as 40.

2.4.7 misc

The size of a dvector can be obtained by size(). If you want to clear the contents, you can use clear(), which also frees the memory. You can find an example of these member functions in "src/tut2.4.7-1".

You can also get the size of each worker's part using sizes(). Please look at the example in "src/tut2.4.7-2".

```
for(auto i: d1.sizes()) std::cout << i << " ";
```

Here, v is evenly scattered, the output should be

```
2 2 2 2
```

if the number of workers is 4. Then,

```
auto d2 = d1.filter(less_than(5));
```

The item 5, 6, 7, and 8 are filtered out in d2. Therefore,

```
for(auto i: d2.sizes()) std::cout << i << " ";
```

output of above should be

```
2 2 0 0
```

Next, we have a method called align_block(), which evenly re-distributes the data.

```
d2.align_block();
for(auto i: d2.sizes()) std::cout << i << " ";
```

This will produce

```
1 1 1 1
```

Before calling align_block, d2 is copied to d3. We have another method called align_to, which align the distribution to the that of the argument (size of both dvectors should be the same).

```
d2.align_to(d3);
for(auto i: d2.sizes()) std::cout << i << " ";
```

This should produce

```
2 2 0 0
```

again.

Or you can directly specify the distribution using align_as.

```
std::vector<size_t> sizes;
sizes.push_back(1);
sizes.push_back(1);
sizes.push_back(2);
sizes.push_back(0);
d2.align_as(sizes);
for(auto i: d2.sizes()) std::cout << i << " ";
```

should produce

```
1 1 2 0
```

Last functionality is `sort`. You can sort the contents of dvector in place. The example is shown in "src/tut2.4.7-3"

2.5 dunordered_map

So far, we explained dvector, which is a distributed version of vector. We have another distributed data structure called dunordered_map, which is a distributed version of unordered_map.

In dunordered_map, each item (Key-Value pair) is distributed according to the hash value of Key. In addition, the Key should be unique just like unordered_map (not multimap).

2.5.1 Creation of dunordered_map from dvector

Usually, dunordered_map is created from dvector, whose actual type should be `dvector<std::pair<K,V>>`. Please go to "src/tut2.5.1-1" and look at tut.cc.

```
auto d1 = frovedis::make_dvector_scatter(v);
auto d2 = d1.group_by_key<int,int>();
```

Here, the type of d1 is `dvector<std::pair<int,int>>`. You can create dunordered_map from d1 by calling `group_by_key<int,int>()`. (Here, you need `<int,int>` for template parameter deduction...)

Here, the type of d2 is `dunordered_map<int,std::vector<int>>`. The method "group_by_key" gathers Key-Value pairs that have the same key, and creates pair of the key and the vector of values that have the same key.

In the above example, there is two Key-Value pairs that have "1" as the key: {1,10} and {1,20}. They are gathered as {1, {10,20}}.

To see the contents of dunordered_map, we need to convert it to dvector, which can be done using a member function as_dvector(). The resulting type is `dvector<int, std::vector<int>>`.

```
auto r = d2.as_dvector().gather();
for(auto i: r) {
  std::cout << i.first << ": ";
  for(auto j: i.second) std::cout << j << " ";
  std::cout << std::endl;
}
```

Here, the type of r is `std::vector<int, std::vector<int>>`. The output should be like

```
4: 10 20
1: 10 20
2: 20 10
3: 20 10
```

The order of keys and values would be different.

The member function group_by_key creates pairs of key and the vector of values. However, it is often required to just create key and value pairs by reducing the vector of values. You can use reduce_by_key for this purpose.

Please go to "src/tut2.5.1-2" and look at tut.cc. You need to specify a function that is used for reduce, which is the same as reduce member function of dvector. In this case,

```
int sum(int a, int b){return a + b;}
```

is used. By calling reduce_by_key like

```
auto d2 = d1.reduce_by_key<int,int>(sum);
```

you can get d2 whose type is `dunordered_map<int,int>`. (Again, you need `<int, int>` here...) In the case of group_by_key, the data was like {1, {10,20}}. Here, reduce_by_key calls sum for the vectors: {10,20}. So the resulting Key-Value should be {1, 30}.

(From the implementation point of view, reduce_by_key is better than calling group_by_key and then reducing the vector, because the system tries to reduce the data locally at first, which reduces the communication.)

```
auto r = d2.as_dvector().gather();
for(auto i: r) std::cout << i.first << ": " << i.second << std::endl;
```

This should output like

```
4: 30
1: 30
2: 30
3: 30
```

2.5.2 Word count

Now we can write the word count program, which is a kind of "hello world" in this field. Please go to "src/tut2.5.2" and have a look at tut.cc .

The core part of word count is as follows:

```
auto d2 =
  d1.flat_map(splitter).map(counter).reduce_by_key<std::string,int>(sum);
```

Here, d1 is `dvector<std::string>` loaded from a file. First, flat_map(splitter) splits the line. The definition of splitter is like

```
std::vector<std::string> splitter(const std::string& line) {
  std::vector<std::string> wordvec;
  boost::split(wordvec, line, boost::is_space());
  return wordvec;
}
```

This splits the line by space and the result is stored into `std::vector<std::string>`, which is returned by this function. Since splitter is called from flat_map, the result is `dvector<std::string>` which consists of each words.

Then, map(counter) is called. Here, counter is defined like this:

```
std::pair<std::string, int> counter(const std::string& v) {
  return std::make_pair(v, 1);
}
```

This returns `std::pair<std::string, int>`, which is word and 1. So the resulting value is `dvector<std::pair<std::string, int>>`.

Then, `reduce_by_key<std::string,int>(sum)` is called. This groups the same words and adds up the occurrence.

The output of this program should be like:

```
test.: 1
it: 1
this: 3
Is: 2
Yes: 2
is!: 1
test?: 2
a: 3
really: 1
is: 1
```

2.5.3 map_values, mapv

The member function map_values is similar to map of dvector. The argument of the input function is Key and Value; returning value is new value. The result of map_values is `dunordered_map<Key, NewValueType>`, where NewValueType is the result type of the input function. Please go to "src/tut2.5.3-1" and see tut.cc.

```
auto d2 = d1.group_by_key<int,int>().map_values(sum_vector);
```

Here, map_values is called using sum_vector. Here, sum_vector is defined like this:

```
int sum_vector(int key, std::vector<int>& v) {
  int sum = 0;
  for(size_t i = 0; i < v.size(); i++) sum += v[i];
  return sum;
}
```

Input of the function is key and value. The key is integer (not used here), and the value is `vector<int>`. This function sum up the contents of the vector and returns it.

This program works just like previous reduce_by_key example.

Like dvector, we also provide non-returning function version of map, mapv. The example is shown in "src/tut2.5.3-2".

2.5.4 filter

Like dvector, we also provide filter. The example is shown in "src/tut2.5.4-1". The program is mostly similar to that of dvector. The difference is that the argument of the input function is key and value, like:

```
bool is_even(int i, std::vector<int>& v) {return i % 2 == 0;}
```

Here, the function only outputs when the key is even (value is not used.)

We also provide inplace_filter. You can find the example in "src/tut2.5.4-2".

2.5.5 put/get

Like dvector, we provide put and get the value using the key. Again, do not use this if performance is critical. But it is convenient for the cases if performance is not important. Please go to "src/tut2.5.5" and look at tut.cc

In this case, `make_dunordered_map_allocate<std::string,int>` is used for creating dunordered_map. Created dunordered_map is empty.

```
d1.put("apple", 10);
d1.put("orange", 20);
```

By this, you can put the key and value pairs.

```
std::cout << d1.get("apple") << std::endl;
```

This should show 10. Then

```
d1.put("apple", 50);
```

modifies the value of apple to 50. So

```
std::cout << d1.get("apple") << std::endl;
```

should show 50.

You can use this as a distributed key value store.

2.6 Zipping the distributed data

So far, we explained using only one distributed data. However, in the practical use cases, combining multiple data is sometimes required. Zipping data is used for this purpose.

2.6.1 Zipping dvector

There is a function called "zip" to create zipped data (whose actual type is `zipped_dvectors<T1,T2>`) Zipped data has the map member function. Please go to "src/tut2.6.1" and look at tut.cc.

```
auto d3 = frovedis::zip(d1, d2).map(sum);
```

Here, d1 and d2 are `dvector<int>`. The map of `zipped_dvectors<int,int>` here takes a function sum. The argument of sum is two int: one is from d1 and the other is from d2. These arguments came from the same index. So this realizes "d3[i] = d1[i] + d2[i]".

Here, the size of d1 and d2 must be the same. If the distribution of d1 and d2 are different, d2.align_to(d1) is automatically called inside of zip.

2.6.2 Zipping dunordered_map

You can also zip dunordered_map. To zip two dunordered_map, the type of the keys should be the same. The type of the value can be different. Please go to "src/tut2.6.2" and look at tut.cc.

```
auto z = zip(d1,d2);
auto d3 = z.map_values(sum);
```

Here, d1, and d2 are `dunordered_map<std::string, int>`. d1 has {"apple", 10}, {"orange", 20}, and d2 has {"apple", 50} and {"grape", 20}.

The type of zipped variable z is `zipped_dunordered_map<std::string, int, int>`. The member function map_values takes a function whose types of argument is Key, the type of the value of d1, and the type of the value of d2.

In the case of map_values, the entries whose key appear in both d1 and d2 are given as the input of the function.

Here, sum is defined as follows:

```
int sum(std::string key, int a, int b) {return a + b;}
```

And only "apple" appears in both d1 and d2. Therefore, the input of sum is "apple", and 10 that appeared in d1, and 50 that appeared in d2. So in this case, map_values outputs a new dunordered_map whose entry is {"apple", 60}.

This is similar to inner join. We have left outer join version of map_values: leftouter_map_values.

```
auto d4 = z.leftouter_map_values(optional_sum);
```

In this case, the third arrgument of the input function is boost::optional. All the entries of d1 is given to the function. If the key does not appear in d2, the third argument becomes empty, which can be tested using if. The function optional_sum is defined as follows:

```
int optional_sum(std::string key, int a, boost::optional<int> b) {
  if(b) return a + *b;
  else return a;
}
```

In this case, the output of leftouter_map_values is a new dunordered_map whose entry is {"apple", 60} and {"orange", 20}. Please note that "grape" is not included.

2.7 Node local variable

So far, I have explained distributed variables that provide "global view"; that is, programmers are not aware of the workers, or actual data distribution.

However, sometimes programmers want to use "local view" of data distribution to write more efficient program.

For example, in the machine learning programs, programmers may want to use some kind of dictionary. If the size of the dictionary is small, they want to just duplicate the data to all workers, instead of using dunordered_map like distributed representation to reduce the latency of looking up the dictionary.

Or in the machine learning programs, programmers may want to locally update the "model" information and then gather the update of all workers. It is difficult to write such kind of programs using only dvectors and/or dunordered_map.

Therefore, we provide other kind of distributed variable: `node_local<T>`, which is a kind of "node local variable". Please look at the example in "src/tut2.7-1".

```
auto l1 = frovedis::make_node_local_broadcast(v);
```

This creates `node_local<vector<int>>` by broadcasting vector v: {1,2}. You can also create node_local with default constructor by using `make_node_local_allocate<T>()`.

The type node_local also have a member function map. The type of argument of the input function is T itself; in this case, `vector<int>`.

```
auto l2 = l1.map(two_times);
```

Here, the function "two_times" takes `vector<int>` and returns `vector<int>` whose items are doubled. On SX, this version of "two_times" would be better than dvector version with function pointer, because the loop can be vectorized.

The type node_local also have a member function gather. The gathered result is `vector<T>`, whose size is the number of workers.

```
auto r = l2.gather();
```

Here, since T is `vector<int>`, the type of r is `vector<vector<int>>` and the result should be

```
2 4
2 4
2 4
2 4
```

If the number of workers is 4.

Like dvector, a member function mapv is also provided for a function that does not return a value.

Next, please go to "src/tut2.7-2". In this case, node_local is created by converting dvector using as_node_local() method. dunordered_map also have the same member function.

The function as_node_local() copies the whole data. There are two ways to avoid copying. If you do not need the original dvector or dunordered_map, you can "move" the variable using moveto_node_local(). After calling this member function, the original data becomes invalid.

Another way is viewas_node_local(). This member function creates node local "view" of the original data. In this case, both original data and created data is valid, but the ownership remains in the original data; therefore, if the original data is destructed, the created data becomes invalid.

In addition, you can make dvector from `node_local<vector<T>>` using `as_dvector<T>()` (or `moveto_dvector<T>()` if you want to move. Note that `<T>` is required for template parameter deduction). On the other hand, we do not provide a member function like asdunordered_map(), because it is not guaranteed that the source unordered_map is properly distributed.

We have a kind of short cut member function called map_partitions for dvector, which does `dvector.as_node_local().map(func).as_dvector<T>()` by dvector.map_partitions(func) without copying. (There is also void returning function version, mapv_partitions.)

The type node_local also has reduce. The example is shown in "src/tut2.7-3". The reduction is done by worker by worker; if the number of workers is 4, the data on worker 0, 1, 2, 3 are summed up using the reduce function.

Therefore, the type of arguments of the input function are T (or const T&), and the type of the return value of it is T.

In this case, the result would be

```
16 20
```

if the number of workers is 4, which is the result of {1,2} + {3,4} + {5,6} + {7,8}.

The most important usage of node_local is that using it as the additional argument of map, etc. The example is shown in "src/tut2.7-4".

Here, d1 is std::vector of std::string, which consists of some items. Then, l1 is `unordered_map<std::string, int>`, which is a kind of dictionary; in this case price of each item, for example.

```
int r = d1.map(convert, l1).reduce(sum);
```

Here, map takes two arguments. First one is a function and the second one is node_local. The function is defined as follows:

```
int convert(std::string item, std::map<std::string,int>& dic) {
  return dic[item];
}
```

The first argument is the item of the dvector as usual. The second argument is the contents of node_local. Using the dictionary the convert function converts the item into the price. Then the price is summed up in the next reduce.

map, mapv, map_values, etc. can take up to 5 node_local variables.

2.8 Using with MPI

Since MPI is SPMD, you need to call the same function on all the workers to use MPI. To do this, we think it would be convenient to utilize map of node_local or map_partitions of dvector. Please look at theexample in "src/tut2.8".

```
d1.mapv_partitions(mpi_func);
```

Here, mpi_func is a function of usual MPI program. In the example, MPI_Bcast is called from the function; therefore, workers communicate with each other inside of the function, before returning to the client program.

The rank of the node can be obtained by frovedis::get_selfid(), and total number of nodes can be obtained by frovedis::get_nodesize(). They are the same as MPI_Comm_rank and MPI_Comm_size.

3 Matrix library

Distributed version and local version of matrix libraries are implemented with the above core functionalities. We will explain them for both dense and sparse matrix.

3.1 Dense matrix

As for dense matrix, we provide rowmajor matrix, columnmajor matirx, and blockcyclic matrix; there are local version of rowmajor matrix and column major matrix.

Here, rowmajor matrix is the most basic structure; loading and saving are provided only for this data structure, and other structure can be converted from rowmajor matrix.

Local version of colmajor matrix supports linear algebra functionality backed by LAPACK and BLAS.

Similarly, distributed version of blockcyclic matrix supports linear algebra functionality backed by ScaLAPACK and PBLAS.

3.1.1 Rowmajor matrix

Please look at "src/tut3.1.1-1/tut.cc". This program multiplies loaded distributed matrix with a vector whose values are all 1. (You do not have to write this kind of program by youserself by using blockcyclic matrix.)

First, you need to include `<frovedis/matrix/rowmajor_matrix.hpp>`.

```
auto m = frovedis::make_rowmajor_matrix_load<double>("./data");
```

This line loads a data to create distributed rowmajor matrix. The type `<double>` is specified to load the data as double. The format of the file is like this:

```
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
```

Each item is separated by space; each row is separated as line.

```
std::vector<double> v(m.num_col);
for(size_t i = 0; i < v.size(); i++) v[i] = 1;
auto lv = frovedis::make_node_local_broadcast(v);
```

These lines create a vector and broadcast it. Here, you can get the number of columns of the loaded matrix as `m.num_col`. The number of rows is `m.num_row`.

```
auto mv = m.data.map(matvec, lv);
```

This calculates the matrix vector multiplication in parallel. Here, the matrix has the member `data` whose type is `node_local<rowmajor_matrix_local<T>>`. The type `rowmajor_matrix_local<T>` is local version of rowmajor matrix; distributed version of rowmajor matrix has node_local of local version of rowmajor matrix. Here, map of the node_local is called to calclate the matrix vector multiplication in parallel. The member num_col, num_row, and data are public.

Then, look at the definition of matvec.

```
std::vector<double>
matvec(frovedis::rowmajor_matrix_local<double>& m, std::vector<double>& v) {
  std::vector<double> ret(m.local_num_row);
  for(size_t r = 0; r < m.local_num_row; r++) {
    for(size_t c = 0; c < m.local_num_col; c++) {
```

```
      ret[r] += m.val[m.local_num_col * r + c] * v[c];
    }
  }
  return ret;
}
```

Here, 1st argument is `rowmajor_matrix_local<double>` because it is the node_local of the map, and 2nd argument is the broadcasted node_local that is given to the 2nd argument of map.

The type `rowmajor_matrix_local<T>` has `local_num_row` and `local_num_col` as the number of rows and number of columns.

It also has the member `val` whose type is `std::vector<T>`. The values of the matrix is stored in the vector in the row major order.

These members are also public, so the user program can access them directly. In the above example,

```
ret[r] += m.val[m.local_num_col * r + c] * v[c];
```

calculates the matrix vector multiply.

Since this function returns `vector<double>`, the returning type of map is `node_local<std::vector<double>>`.

```
auto g = mv.moveto_dvector<double>().gather();
for(auto i: g) std::cout << i << std::endl;
```

These lines convert it to dvector, gather, and print the result. It should be like:

```
15
40
65
90
```

Next, please look at "src/tut3.1.1-2/tut.cc". This program shows binary data load:

```
auto m = frovedis::make_rowmajor_matrix_loadbinary<double>("./data_binary");
```

To load the data as double, `<double>` is given as the template parameter. Here, `data_binary` is a directory that contains three files: nums, val, and type. The contents of nums is two lines of text num_row and num_col. In this case,

```
4
5
```

because the matrix is 4 by 5. Then, `val` contains the values of the matrix in *little endian* binary form, which is similar to dvector case. The file `type` contains the type information of `val`, which is `double` in this case.

```
m.transpose().save("./transposed");
```

This demonstrates the `transpose` functionality that transposes the matrix, and `save` functionality to save the matrix. In this case, the matrix is saved in text format. It should be:

```
1 6 11 16
2 7 12 17
3 8 13 18
4 9 14 19
5 10 15 20
```

You can save the matrix in binary form using `savebinary` member function.

You can convert text matrix and binary matrix using `to_binary_rowmajor_matrix` and `to_text_rowmajor_matrix` in "samples/matrix_tools" directory. If you want to convert endian, you can use `convert_endian_dense_matrix.sh`.

You can also use `operator<<` and `operator>>` to input and output the matrix in text format. You can also locally load `rowmajor_matrix_local` using `make_rowmajor_matrix_local_load[binary]`. You can also save `rowmajor_matrix_local` using save or savebinary member function.

3.1 2 Colmajor matrix

Please look at "src/tut3.1.2-1/tut.cc" as an example of colmajor matrix. This is almost the same as the "src/tut3.1.1-1/tut.cc".

```
auto rm = frovedis::make_rowmajor_matrix_load<double>("./data");
frovedis::colmajor_matrix<double> m(rm);
```

By defining a colmajor matrix by giving a rowmajor matrix as its constructor argument, you can convert a rowmajor matrix to a colmajor matrix. Then,

```
auto mv = m.data.map(matvec, lv);
```

calls the `matvec` function for local version of colmajor matrix. Then please look at matvec function:

```
frovedis::gemv<double>(m, v, ret);
```

Because colmajor_matrix_local is backed by LAPACK/BLAS, you can utilize BLAS routine to calculate matrix vector multiplication. Here, you need `<double>` for template parameter deduction.

To use the BLAS routine, you need to include `<frovedis/matrix/blas_wrapper.hpp>`.

Though we gave colmajor_matrix_local and std::vector as the arguments of gemv, actual arguments of gemv is `sliced_colmajor_matrix_local` and `sliced_colmajor_vector_local`.

Here, "sliced" type contains the pointer to the data and stride or leading dimension information, etc. There are "converting constructors" from `colmajor_matrix_local` or `vector` to `sliced_colmajor_matrix_local` or `sliced_colmajor_vector_local`. With these constructors, `colmajor_matrix_local` or `vector` is implicitly converted to `sliced_colmajor_matrix_local` or `sliced_colmajor_vector_local` and given as the arguments of gemv. You can also explicitly create `sliced_colmajor_matrix_local` or `sliced_colmajor_vector_local`.

In addition, you can use part of a matrix as a vector or a matrix. For example,

```
sliced_colmajor_vector_local<T>
make_row_vector(const sliced_colmajor_matrix_local<T>& matrix,
                size_t row_index)
```

creates a vector as `sliced_colmajor_vector_local` from specified row index of the matrix. Likewise, `make_col_vector` creates `sliced_colmajor_vector_local` from specified column index.

To create part of a matrix,

```
sliced_colmajor_matrix_local<T>
make_sliced_colmajor_matrix_local(const sliced_colmajor_matrix_local<T>& matrix,
                                  size_t start_row_index,
                                  size_t start_col_index,
                                  size_t num_row,
                                  size_t num_col)
```

creates `sliced_colmajor_matrix_local` from start_row_index, start_col_index, and num_row / num_col size.

The program "src/tut3.1.2-2/tut.cc" shows the example of the above function.

```
auto sm1 = frovedis::make_sliced_colmajor_matrix_local<double>(m1, 0, 0, 3, 3);
```

this creates sliced matrix whose size is 3x3, and start from row 0 / column 0.

This can be used as the argument of matmul, like

```
auto mm = sm1 * m2;
```

Then,

```
std::cout << mm.to_rowmajor();
```

shows the result of matmul. The member function `to_rowmajor` converts the matrix into `rowmajor_matrix_local`, which supports `operator<<`. The result should be:

```
46 74 210
131 209 580
216 344 950
```

Please look at the example in "src/tut3.1.2-3/tut.cc". This is an example of using `gesv` that solves Ax = b.

```
frovedis::gesv<double>(m,v);
```

This solves mx = v. The result is overwritten into v. The return value of `gesv` is the same as that of LAPACK.

In this case, you need to include `<frovedis/matrix/lapack_wrapper.hpp>`. The result should be:

```
 1.7
-0.3
-0.2
```

Currently, supported lapack routines are: gesv (solve) gels/gelsd/gelsy/gelss (least square), geev (eigen value), gesvd/gesdd (svd), getrf (LU decomposition), getri (inverse matrix using LU), getrs (solve using LU).

3.1.3 Blockcyclic matrix

In the previous section, we showed routines backed by BLAS and LAPACK, which only works sequentially. Blockcyclic matrix can work similarly with distributed matrix, which is backed by ScaLAPACK and PBLAS.

Please look at "src/tut3.1.3-1/tut.cc", which is similar to rowmajor matrix version in "src/tut3.1.1-1/tut.cc".

```
auto m = frovedis::make_blockcyclic_matrix_load<double>("./data");
```

This line creates blockcyclic matrix by loading a text data from a file. Actually, block cyclic matrix is created from colmajor matrix by providing it as the constructor. But creating colmajor matrix from rowmajor matrix and block cyclic matrix from colmajor matrix is tedious, so we provide a function to directly create it.

```
std::vector<double> v(m.num_col);
for(size_t i = 0; i < v.size(); i++) v[i] = 1;
auto dv = frovedis::make_blockcyclic_matrix_scatter(v);
```

These statements creates distributed vector by scattering a local vector. Actually, ScaLAPACK treats Nx1 matrix as a vector, so the type of `dv` is `blockcyclic_matrix`.

```
auto mv = m * dv;
```

This is actually matrix matrix multiplication. In this case, it creats Nx1 matrix.

```
mv.save("./result");
```

This saves the result as text file. It should be like

```
15
40
65
90
```

Next, please look at "src/tut3.1.3-2/tut.cc", which is similar to colmajor matrix sequential version in "src/tut3.1.2-3/tut.cc".

Again, we can just create blockcyclic matrix by loading or scattering. Then,

```
frovedis::gesv<double>(m,dv);
```

is called to solve mx = dv. The result is stored in dv and saved in "result" file. It should be:

```
 1.7
-0.3
-0.2
```

Supported ScaLAPACK routines are: gesv (solve), gels (least square), gesvd (svd), getrf (LU decomposition), getri (inverse matrix using LU), getrs (solve using LU).

3.2 Sparse matrix

3.2.1 CRS matrix

CRS (Compressed Row Storage) is the most popular format of sparse matrix. Please look at "src/tut3.2.1-1/tut.cc"

It looks like "tut3.1.2/tut.cc"; the data structure is changed to CRS format. You need to include `<frovedis/matrix/crs_matrix.hpp>` to use CRS format.

```
auto m = frovedis::make_crs_matrix_load<double>("./data");
```

This line loads a data to create distributed CRS matrix. The type `<double>` is specified to load the data as double. The format of the file is like this:

```
0:10 1:-3 2:-1 3:1 4:0 5:1
1:9 2:6 3:-2 4:5 5:5
0:3 2:8 3:7 4:5
1:6 2:2 3:7
4:9
4:5 5:-1
```

Each item is separated by space, and each row is separated as line. Each item is like "POS:VAL"; POS is 0-based.

Like dense matrix, there is a local version of sparse matrix. In this case, `crs_matrix_local`.

Operator* is supported for `crs_matrix_local` and std::vector. So `m * v` is returned from `matvec`.

Like dense matrix, `crs_matrix_local` has the member `val` whose type is `std::vector<T>`. It also has a member `idx` that contains the column index of each value, and a member `off` that contains the offset that indicates the starting position of each row; `idx` and `off` are 0-based. Types of idx and off are `size_t` by default, but you can change them using template parameters.

They are public to users, so user program can access them directly.

The result of the program should be:

```
8
23
23
15
9
4
```

Though this is our standard CRS format, we also support loading other formats. One is COO (Coordinate) format. That includes row and column index and its value, like:

```
1 1 1.0
2 1 2.0
2 2 3.0
```

You can use make_crs_matrix_loadcoo for loading this type of format. Each data is separated by space. By default, indices are *1-based*. You can change to use 0-based index by specifying the second argument as `true`. You can use this function for loading data that is created by octave or matlab.

Another format is LibSVM format. This format is used in the famous machine learning library libsvm. It is similar to the above CRS format, but first data of the line is the "label" that is used for machine learning, like:

```
1 1:10 2:1 3:1
-1 2:9 3:6
1 1:3 3:8
```

Also in this case, the indices are *1-based*. You can load the file by using make_crs_matrix_loadlibsvm(const std::string&, dvector&). The label data is returned in the second argument.

Next, please look at "src/tut3.2.1-2/tut.cc". This program shows binary data load, which is quite similar to tut3.1.1-2.

```
auto m = frovedis::make_crs_matrix_loadbinary<double>("./data_binary");
```

To load the data as double, `<double>` is given as the template parameter. Here, `data_binary` is a directory that contains 5 files: nums, val, idx, off, and type. The contents of nums is two lines of text, num_row and num_col. In this case,

```
6
6
```

Then, `val` contains the values of the matrix, `idx` contains the column indices of the values, and `off` contains the offset that shows the starting position of each row. All of them are in *little endian* binary form. The file `type` contains the type information of `val`, `idx`, and `off`, which is `double`, `uint64_t`, and `uint64_t` in this case.

```
m.transpose().save("./transposed");
```

This demonstrates the `transpose` functionality that transposes the matrix, and `save` functionality to save the matrix. In this case, the matrix is saved in text format. It should be:

```
0:10 2:3
0:-3 1:9 3:6
0:-1 1:6 2:8 3:2
0:1 1:-2 2:7 3:7
0:0 1:5 2:5 4:9 5:5
0:1 1:5 5:-1
```

You can save the matrix in binary form using `savebinary` member function.

You can convert text matrix and binary matrix using `to_binary_crs_matrix` and `to_text_crs_matrix` in "samples/matrix_tools" directory. If you want to convert endian, you can use `convert_endian_sparse_matrix.sh`.

You can also use `operator<<` to output the matrix in text format. You can also locally load `crs_matrix_local` using `make_crs_matrix_local_load[binary]`. You can also save `crs_matrix_local` using savebinary member function.

3.2.2 Other sparse matrix formats

You can create other formats of sparse matrixes from CRS matrix. Please look at "src/tut3.2.2/tut.cc".

```
auto crs = frovedis::make_crs_matrix_load<double>("./data");
frovedis::ccs_matrix<double> m(crs);
```

By providing CRS matrix as the input of constructor of ccs_matrix, you can create CCS format of the matrix.

You can also create ELL, JDS, hybrid of JDS and CRS format of sparse matrix likewise, which are commented out in tut.cc.

Operaot* is also defined for these formats.

The purpose of different formats is mostly for performance. For example, if number of non-zero of each row is small, using JDS would improve performance of SpMV (operator*) on vector a architecture. If the distribution of number of non-zero follows power law, hybrid of JDS and CRS would be a better choice.

3.2.3 Singular value decomposition on sparse matrix

We provide singular value decomposition on sparse matrix. Please look at "src/tut3.2.3/tut.cc".

```
auto crs = frovedis::make_crs_matrix_load<double>("./data");
frovedis::colmajor_matrix<double> u;
frovedis::colmajor_matrix<double> v;
frovedis::diag_matrix_local<double> s;
frovedis::sparse_svd(crs, u, s, v, 3);
```

The result is stored in `colmajor_matrix` u, v, and `diag_matrix_local` s. The `diag_matrix_local` is a diagonal matrix, which has `std::vector` as a member. The colmajor_matrix u and v are singular vectors in the form of matrix; each column represents the vectors. So in this case, crs can be approximated by u * s * v^t.

By calling `frovedis::sparse_svd(crs, u, s, v, 3)`, SVD is calculated. The last argument is the number of singular values/vectors to compute (from larger singular values). This function is backed by Parallel ARPACK.

```
std::cout << u.to_rowmajor();
std::cout << s;
std::cout << v.to_rowmajor();
```

These statements prints the calculated values. The result would be like:

```
-0.0838495 0.754634 -0.0435034
-0.107839 -0.430217 0.640159
0.180889 0.487546 0.576771
0.608964 -0.00833374 0.368304
-0.665538 0.0720435 0.307198
-0.367208 0.0498016 0.160064
10.0444
12.1621
16.814
-0.0294522 0.740742 0.0770359
0.292182 -0.508617 0.481848
```

```
0.209257 0.045039 0.54926
0.563579 0.40861 0.314721
-0.742768 0.0973555 0.573913
-0.0254708 -0.118915 0.178258
```

(Note that positive or negative might be different because of the characteristics of SVD.)

You can use different formats of sparse matrix. In this case, the interface is a bit different, like commented out part.

The "samples/svd" directory contains sample program that does sparse singular value decomposition.

4 Machine learning algorithms

We are actively implementing machine learning algorithms. Currently, following algorithms are implemented, but interface might change in the future.

4.1 Generalized linear model

This class of algorithms includes classification with logistic regression and linear support vector machine, and linear regression.

Supported data is currently sparse data (supporting dense data is not so difficult, though...).

Please look at "src/tut4.1-1/tut.cc".

```
auto samples = frovedis::make_crs_matrix_load<double>("./train.mat");
auto label = frovedis::make_dvector_loadline("./train.label").
  map(+[](const std::string& s){return boost::lexical_cast<double>(s);});
```

These statements load training data. The matrix "train.mat" only contains the features of the samples. The vector "./train.label" contains label of the samples. (these data are created from the "iris data" of UCI machine learning repository.)

As mentioned before, the matrix is a sparse matrix. Each row corresponds to each sample. The label contains "1" or "-1" for each sample. Because `make_dvector_loadline` creates vector of string, it is converted to double using boost::lexical_cast.

```
auto model = frovedis::logistic_regression_with_sgd::
  train(samples, label, num_iteration, alpha, minibatch_fraction, regParam,
        rt, intercept);
```

This function does training of logistic regression using stochastic gradient descent; num_iteration is number of iteration, alpha is learning rate, minibatch_fraction is fraction of dividing the data into minibatch, regParam is regularization rate, rt is regularization type, and intercept is if intercept (bias) is to be used or not.

```
model.save("./model");
frovedis::logistic_regression_model<double> lm;
lm.load("./model");
```

Here, the trained model is saved into "./model", and loaded into a variable `lm` again to demonstrate how to save or load the model. You can save or load in binary form using savebinary or load_binary.

```
auto test = frovedis::make_crs_matrix_local_load<double>("./test.mat");
auto result = lm.predict(test);
```

Using the loaded model, prediction is performed. The matrix "./test.mat" contains features for prediction. Here, prediction is not parallelized, so note that the data is `crs_matrix_local`.

The vector `result` contains predicted label. It should be same as (or similar to) the data in "./test.label".

If you want to use linear SVM, use the commented out part, whose interface is mostly the same as logistic regression.

Next, please look at "src/tut4.1-2/tut.cc". It includes regression case.

Most of the part is the same as classification case. In this case, `frovedis::linear_regression_with_sgd::train` is called for regression. For L2 or L1 regularization, `frovedis::ridge_regression_with_sgd::train` or `frovedis::lasso_with_sgd::train` can be used.

The test data in this directory contains data created from housing data of USI machine learning repository. The result of the program would be like:

```
24.6685
24.3606
24.117
24.2339
24.3711
...
```

(The prediction is not so accurate, though...)

The "samples/glm" directory contains programs of these algorithms. You can use them as command line applications.

4.2 Matrix factorization

You can use SVD for matrix factorization, but we also implemented another version of matrix factorization for recommender system. The algorithm is based on a paper "Collaborative Filtering for Implicit Feedback Datasets" by Hu, et al.

Please look at "src/tut4.2/tut.cc". The structure is mostly the same as previous example; it only takes the matrix dataset, not label.

By calling `frovedis::matrix_factorization_using_als::train`, the matrix is factorized into two dense small matrices, which are in the model structure.

Using the model, `predict` member function can be used to predict if a user like an item or not. Several kinds of predict interface is provided; here we used `predict(userid, itemid)` version, which returns some value that indicates if the user likes it or not: higher the better (not the rating value itself).

The matrix is created from MovieLens. (http://files.grouplens.org/datasets/movielens/ml-100k.zip, F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. ACM Transactions on Interactive Intelligent Systems (TiiS) 5, 4, Article 19 (December 2015)) We used data 100K (used u1.base).

The result would be like:

```
user = 1, movie = 6: 0.201767
user = 1, movie = 10: 0.552775
user = 1, movie = 12: 1.02576
user = 1, movie = 14: 0.621925
```

```
user = 1, movie = 17: 0.0876253
user = 1, movie = 20: 0.616092
user = 1, movie = 23: 0.603632
user = 1, movie = 24: 0.504314
user = 1, movie = 27: 0.0290654
user = 1, movie = 31: 0.387138
```

By looking at u1.test, the rating was:

```
1   6   5   887431973
1   10  3   875693118
1   12  5   878542960
1   14  5   874965706
1   17  3   875073198
1   20  4   887431883
1   23  4   875072895
1   24  3   875071713
1   27  2   876892946
1   31  3   875072144
```

Movie id 12 and 14 shows high rating and predicted value is also high. The rating of movie id 27 is low and the predicted value is also low.

The "samples/matrixfactorization" directory contains programs of this algorithm. You can use them as command line applications.

4.3 Clustering

4.3.1 K-means

We also provide clustering algorithm using k-means.

Please look at "src/tut4.3-1/tut.cc". Again, The structure is mostly the same as previous example; it only takes the matrix dataset, not label. The data "./train.mat" is synthesized data.

By calling `frovedis::kmeans(samples, k, num_iteration, eps)`, you can get the centroids of the cluster. Here, k is the number of centroids. The type of the centroids is `rowmajor_matrix_local.`, where each column shows each centroid vector. So in the program, the matrix is transposed and saved.

The centroids can be used to assign data to the closest centroid by calling `kmeans_assign_cluster`. The result should be like:

```
2
1
0
```

4.3.2 DBSCAN

DBSCAN is another clustering method. Since it is based on the density of the samples, it might produce better clustering method. However, it might take longer time than K-means, especially if the number of dimension of the sample is large.

Please look at "src/tut4.3-2/tut.cc", which looks like this:

```
auto dbscan = frovedis::dbscan(0.5, 5);
dbscan.fit(m);
auto labels = dbscan.labels();
```

First, you need to create `frovedis::dbscan` object; first argument of the constructor is epsilon, which is the maximum distance between two samples that can be considered as neighborhood. The second argument is the minimum number of samples in a neighborhood that is needed to be considered as a core point.

After creating the object, you can call `fit` method that takes `rowmajor_matrix` as the argument; each row of the matrix represents a sample.

The fit method assigns each point a label. That can be obtained by `labels()` method. If a sample is considered as noise, the label becomes -1.

In this example, the sample data is created by a python script `gendata.py`. You can find the label of the generated data as `train.label`. It should be mostly the same as the clustering result.

4.4 Factorization Machine

We provide factorization machine algorithm.

Please look at "src/tut4.4/tut.cc". By calling below function, you can train the factorization machine model for classification/regression task.

```
auto trained = frovedis::fm_train(dim_0, dim_1, dim_2,
    init_stdev, iteration, init_learn_rate, optimizer,
    regular_0, regular_1, regular_2, is_regression,
    data, label, batchsize_pernode);
```

`dim_0` switches to use bias or not.
`dim_1` switches to use 1-way interation or not.
`dim_2` is the dimension of 2-way interation.
`init_stdev` is tdev for initialization of 2-way factors.
`iteration` is number of iteration.
`init_learn_rate` is initial learning rate for SGD.
`optimizer` is kind of optimizer.
`regular_0` is bias regularization parameter.
`regular_1` is 1-way regularization parameter.
`regular_2` is 2-way regularization parameter.
`is_regression` switches regression or classification.
`data` is training samples of crs matrix.
`label` is training labels.
`batchsize_pernode` is the size of minibatch processed by one node (actual size is this value multiplied by the number of nodes).

Our implementation followed to libFM, so it is helpfull to refer that to understand the detail. You can acquire predictions from trained model by calling `fm_model::predict` method and calculate RMSE based on trained model and test data and labels with calling `fm_test` method.

4.5 Decision Trees

We provide decision tree and tree ensemble algorithms for classification and regression. Currently, a dense data format is supported.

4.5.1 Decision Tree

Please look at "src/tut4.5.1-1/tut.cc". This is a classification tree example with data created from the iris dataset. Unlike "tut4.1-1", the label is "0", "1", or "2" for each sample. The following function builds a classification tree model.

```
auto model = frovedis::decision_tree::train_classifier(
  dataset, labels,
  num_classes, categorical_features_info,
  impurity, max_depth, max_bins
);
```

There are problem specification parameters and tunable parameters.

Please specify `num_classes` and `categorical_features_info` depending on the problem you want to solve and your dataset. The number of classes is specified by `num_classes`. Here, this example is a 3-class classification problem. You can specify categorical features' information, which gives column indices of categorical features and the number of categories for those features (but here, `categorical_features_info` is leaved empty because the iris dataset has no categorical feature). For example, an unordered_map { {0, 2}, {4, 5} } means that the feature[0] takes values 0 or 1 (binary) and the feature[4] has five categories (values 0, 1, 2, 3 or 4). Note that feature indices and category assignments are 0-based.

The following parameters may be tuned. An impurity function is specified by `impurity`. Several types of impurity functions are available: typical one is "impurity_type::Gini" or "impurity_type::Entropy" for classification problems. The default impurity function is the Gini impurity. The maximum tree depth (i.e. the tree height) is limited by `max_depth`. If `max_depth` is 0, you get a tree which has only a single node. Continuous features are divided up into histogram bins, and the maximum number of its bins is tuned by `max_bins`.

Then, a constructed tree model can be dumped in a simple text format which would be like:

```
<1> Split: feature[2] < 1.92188, IG: 0.333333
 \_ (2) Predict: 0 (100%)
 \_ <3> Split: feature[3] < 1.70312, IG: 0.378547
    \_ <6> Split: feature[2] < 4.94062, IG: 0.0889681
    |  \_ <12> Split: feature[3] < 1.6125, IG: 0.04543
    |  |  \_ (24) Predict: 1 (100%)
    |  |  \_ (25) Predict: 2 (100%)
    |  \_ <13> Split: feature[3] < 1.55, IG: 0.222222
    |     \_ (26) Predict: 2 (100%)
    |     \_ <27> Split: feature[2] < 5.4, IG: 0.444444
    |        \_ (54) Predict: 1 (100%)
    |        \_ (55) Predict: 2 (100%)
    \_ <7> Split: feature[2] < 4.86562, IG: 0.0150704
       \_ <14> Split: feature[0] < 6.05, IG: 0.111111
       |  \_ <28> Split: feature[0] < 5.95, IG: 0.5
       |  |  \_ (56) Predict: 1 (100%)
       |  |  \_ (57) Predict: 2 (100%)
       |  \_ (29) Predict: 2 (100%)
       \_ (15) Predict: 2 (100%)
```

If the "Split" condition is true, the first child node is the next place, otherwise the second child node. For example, when you look at the node <1> and if the condition "feature[2] < 1.92188" is false, the next place is the node <3>. The "IG" indicates the information gain with the spliting.

A `predict` method of a tree model returns predicted results. If you need a probability of the prediction, please use a `predict_with_probability` method. For example, the following statement prints predicted classes and their probabilities.

```
for (auto result: results) {
  std::cout << result.get_predict() << " (" <<
            << result.get_probability() * 100 << "%)" << std::endl;
}
```

Next, please look at "src/tut4.5.1-2/tut.cc". This is a regression tree example with data created from the Boston housing dataset. Most of the part is the same as the classification case. The following function builds a regression tree model.

```
auto model = frovedis::decision_tree::train_regressor(
  dataset, labels,
  categorical_features_info,
  impurity, max_depth, max_bins
);
```

There is no `num_classes` parameter. About an impurity function, typical one is "impurity_type::Variance" for regression problems. A constructed tree model has common methods for classification/regression, but probabilities which are predicted by a regression tree model make no sense (those values are always set to 0).

4.5.2 Random Forest

Random forest is a tree ensemble algorithm based on decision trees, and many of parameters are common with the decision tree. Here, we describe some additional parameters. Please look at "src/tut4.5.2-1/tut.cc". This is a classification example with the same iris dataset as "tut4.5.1-1". The following function builds a classification forest model.

```
auto model = frovedis::random_forest::train_classifier(
  dataset, labels,
  num_classes, categorical_features_info,
  num_trees, feature_subset_strategy,
  impurity, max_depth, max_bins,
  seed
);
```

There are additional parameters `num_trees`, `feature_subset_strategy`, and `seed`. Please specify `num_trees` as the number of trees. In this example `num_trees` is set to 3, but please set a more large number in practice. It depends on the dataset size and the feature dimension, but generally speaking, hundreds or thousands of trees are effective.

Random forest does subsampling (sampling from dataset rows) for each tree and does feature-sampling (sampling from dataset columns) for each tree node. The default subsampling strategy is bootstrapping. You can specify a feature-sampling strategy by `feature_subset_strategy`, and typical one is "feature_subset_strategy::Sqrt" for classification problems. This means, when the feature dimension is `m`, only `sqrt(m)`-pieces of features are used for each tree node construction. At last, `seed` is a random seed. In this example, `seed` is set to current time, so you may get different results on every execution.

A regression example with Boston housing dataset is "src/tut4.5.2-2/tut.cc", and its parameters are almost the same as the classification case. For regression forest, a typical feature-sampling strategy is "feature_subset_strategy::OneThird". This means only `m/3`-pieces of features are used when the feature dimension is `m`.

4.5.3 Gradient Boosted Trees

Gradient Boosted Trees (GBTs) algorithm, a.k.a. Gradient Boosting Decision Trees (GBDT) algorithm, is provided for {1, -1} binary classification and for regression. Multi-class classification is not supported, so please use decision tree or random forest for multi-class classification problems.

Please look at "src/tut4.5.3-1/tut.cc". This is a {1, -1} binary classification example with data created from the iris dataset (but the label is different from "tut4.1-1"; "Iris-virginica" is assigned to 1, otherwise -1). Again, please note that labels must be {1, -1}, not {0, 1}, for GBTs. The following function builds a classification GBTs model.

```
auto model = frovedis::gradient_boosted_trees::train_classifier(
  dataset, labels,
  categorical_features_info,
  impurity, max_depth, max_bins,
  num_iterations, loss, learning_rate
);
```

There are additional parameters `num_iterations`, `loss`, and `learning_rate`. But please take care of `impurity` as well. Even if classification GBTs, it is internally based on *regression* trees, so `impurity` must be set to an impurity function for *regression*. Again, a typical impurity function for regression is "impurity_type::Variance".

Let's look into parameters for GBTs. Like as random forest, please specify `num_iterations` as the number of trees. A loss function is specified by `loss` and "loss_type::LogLoss" is available for classification problems. It works like logistic regression. You can tune a learning rate by `learning_rate`.

A regression example with Boston housing dataset is "src/tut4.5.3-2/tut.cc". For regression GBTs, several types of loss functions are available: "loss_type::LeastSquaresError" and "loss_type::LeastAbsoluteError".

4.6 Word2Vec

We provide word2vec algorithm. It is based on Intel pWord2Vec implementation. We only support a variation of negative-sampling and skip-gram.

There are 3 steps to make word embeddings from raw text. First is to create vocabulary by calling `w2v_build_vocab_and_dump(train_file, stream_file, vocab_file, vocab_count_file, min_count)` method. `train_file` is path for raw text in which words are separated by space, and sentences by newline. 2nd, 3rd, 4th arguements specifies paths for output file. `stream_file` is sequence of one-hot integers made by input text, `vocab_file` is mapping between one-hot id and words. `vocab_countf_file` is counts of each word appeared in input text. Words with total frequency lower than `min_count` is not included in vocabulary. We suggest to run this step on x86 due to inefficiency with vector engine.

Second step is to train word2vec model. Take care the computation is performed with OpenMP, so appropriate numbers of MPI process and OpenMP should be determined for fast computation. The method for training is,

```
auto weight = frovedis::w2v_train(
    nl_train_data, vocab_count, hidden_size, window,
    sample, negative, iter, alpha, model_sync_period,
    min_sync_words, full_sync_times, message_size, num_threads);
```

`nl_train_data`, `vacab_count` are data structure constructed before. `hidden_size` is dimension of embedding vector. `window` is the maximum lenght of current and predicted words in a sentence. `sample` is threshold to donwsample for high-frequency words. `negative` is the number for negative samples. `alpha` is initial learning

rate. `num_threads` specifies the number of OpenMP threads used for inner-loop computation. Return value of this method is row major matrix in which each row specifies embedding vector of a word.

Third step is save result to file of specific format. It is done by calling `w2v_save_model(weight, vocab_file, word_vectors_file, binary, min_count)` method. `vocab_file` is path of file made in first step. `binary` switches binary of output, `word_vectors_file`.

4.7 Deep Learning

We provide preliminary support of deep neural network. It is based on third party library "tiny-dnn", which is very cleanly written C++ deep learning library. We modified it and added some modules to support distributed training and to improve performance on vector architecture. It also depends on a library called "vednn", which provides optimized kernel operation of layers (e.g. convolutional layer) on vector architecture.

Basic idea of how to use it is the same as tiny-dnn. Distributed training is implemented by splitting the training data into the nodes, which is so called data parallel style. Each node train the local data; at the time of back propagation, the gradient calculated at each node is all-gathered, which takes place in the optimizer.

Please look at "src/tut4.7-1/tut.cc", which uses MLP (multy layer perceptron) to recognize hand written digit (MNIST).

To run the code, you need MNIST dataset. If you installed Frovedis from rpm, you can find it in ${INSTALL-PATH}/data/mnist.tar.gz. Please extract data from the file.

First, you need to include `<frovedis/ml/dnn/dnn.hpp>` in addition to `<frovedis.hpp>`. Since it uses mnist data, it also includes `<tiny_dnn/io/mnist_parser.h>`. The file `<tiny_dnn/io/display.h>` is used to display the progress. (Unlike the original tiny-dnn, we tried to separate including files.)

In the main function, arguments are parsed and the function `train` is called. Please go to the function train.

The function `parse_mnist_labels` and `parse_mnist_images` loads the images and labels of the MNIST data. The type of loaded label is `std::vecctor<label_t>`, where `label_t` is defined as `size_t`. The type of loaded image is `std::vector<vec_t>`, where `vec_t` is defined as `std::vector<float_t>`. The type of `float_t` is defined as `float`.

The label and image is distributed using `make_dvector_scatter`.

Next, `backend_type` is created. On x86, it should be internal. On VE, it should be ve.

The network is defined as `network<sequential> nn`. It is constructed in function `construct_net`. It is exactly the same as tiny-dnn. As you can see, you can add layers using `<<` operator to the network.

As for optimizer, `dist_RMSprop optimizer` is defined. This is distributed version of optimizer and all-gathers the gradient calculated at each node internally. At this moment, we provide distributed version of RMSprop and adagrad.

Tiny-dnn supports to call any function at the end of training of mini batch or epoch. The function object is defined as `on_enumerate_epoch` and `on_enumerate_minibatch`. Here, these function objects need to be sent to other nodes, so they need to be serialized. Therefore, we need to use function objects instead of lambda function of C++.

Usually, the usage of these functions is to see the progress of the training. Therefore, the pointer to the network, etc. is passed to the function object. Please note that the pointer is valid only on rank 0 of the nodes. Therefore, execution of the functions is guarded by `if(get_selfid() == 0)`.

Then you can just call `dnn_train` to train the network. It is similar to `train` method of network of tiny-dnn.

After training, all nodes have the same weight of network. So you can test and save the model on rank 0.

Currently, MLP and CNN with some other layers (e.g. dropout, batch normalization) are supported. As for CNN, please see "src/tut4.7-2/tut.cc".

It is mostly the same as the previous example. Only the network construction is different. (In this example, testing during the training is comment out'ed, because it causes a problem during training with batch normalization...)

5. Dataframe

Dataframe for preprocessing is also supported. It provides functionalities like filter, sort, join, and group by of tables. A table consists of columns; types of the columns may be different.

Internally, table is represented as collections of dvectors, which is more or less similar to columnar store databases.

5.1 Table creation

You can create tables by various ways. The most basic way is to append dvectors to a table. Please look at "src/tut5.1-1/tut.cc".

To use dataframe, you need to include `<frovedis/dataframe.hpp>`. After creating dvectors of type int, double, and std::string, they are appended to a dftable t.

```
frovedis::dftable t;
t.append_column("c1", d1);
t.append_column("c2", d2);
t.append_column("c3", d3);
```

The first argument of `append_column` is the name of the column, which will be used to manipulate the table.

The created table can be saved by the `save` member function. Here, the table is saved into directory "./t". It contains the saved dvectors of the columns (c1, c2, and c3). They are saved as binary format unless the type is std::string. The file `columns` contains the names of the columns. The file `types` contains the type names of the columns. The files like `c1_null` contains the information of the null data, which might be created by outer join. In this case, they contain nothing.

Then, the saved dftable can be loaded.

```
auto t2 = frovedis::make_dftable_load("./t");
```

Here, the saved dftable is loaded to another dftable t2. You can see if the table is correctly loaded by the show member function:

```
t2.show();
```

It should ouptut the result to standard out like

```
c1    c2      c3
1     10      a
2     20      b
3     30      c
4     40      d
5     50      e
6     60      f
7     70      g
8     80      h
```

If you want to limit the output size, you can specify the number of rows as the argument of show, like show(3).

The appended dvector can be extracted by as_dvector. In this case, you need to give the type of the column because it cannot be known statically.

```
auto dc1 = t.as_dvector<int>("c1");
for(auto i: dc1.gather()) std::cout << i << " ";
std::cout << std::endl;
```

It should produce output like

```
1 2 3 4 5 6 7 8
```

The types that can be appended to dftable is currently dvector of int, unsigned int, long, unsigned long, long long, unsigned long long, float, double, and std::string.

Internally, std::string is treated differently to get better performance. That is, a dictionary is created and the column contains the index to the dictionary. It might take some time to create the dictionary especially on a vector machine.

You can create tables by loading from a text file. Please look at "src/tut5.1-2/tut.cc".

```
auto t = frovedis::make_dftable_loadtext("./t.csv",
                                         {"int", "double", "string"},
                                         {"c1", "c2", "c3"});
```

This function creates dftable by loading a file "./t.csv". The second argument is types of the columns, and the third argument is the names of the columns. Both are represented as `std::vector<std::string>`.

Here, we assume that the columns are separated by comma. If you want to use a different separater, you can use `make_dftabke_loadtext_sep` and specify the separater as the last argument of it.

In addition, if the first line of the file is the name of the columns, you can remove the column name argument; the function treats the first line of the file as the column names.

You can also save the table as text.

```
auto coltypes = t.savetext("./saved.csv");
```

The member function `savetext` saves the table as text. The value of each column is separated by comma by default. You can change the separater by giving the string as the second argument of savetext.

It returns the `std::vector<std::pair<std::string, std::string>>` as the return value; the pair represents the column name and its type. The return value is the same as that of `dtypes` member function.

Lastly, you can create dftable from a dvector of tuple. Please look at "src/tut5.1-3/tut.cc". You need to include `frovedis/dataframe/make_dftable_dvector.hpp`

```
vector<tuple<int,double,string>>
  v = {make_tuple(1,10.0,string("a")),make_tuple(2,20.0,string("b")),
       make_tuple(3,30.0,string("c")),make_tuple(4,40.0,string("d")),
       make_tuple(5,50.0,string("e")),make_tuple(6,60.0,string("f")),
       make_tuple(7,70.0,string("g")),make_tuple(8,80.0,string("h"))};
auto d = frovedis::make_dvector_scatter(v);
```

First, create a dvector of tuple.

```
auto t = frovedis::make_dftable_dvector(d,{"c1","c2","c3"});
```

Then, dftable is created by `make_dftable_dvector`. First argument is the dvector, and the second argument of the list of column names.

You can also create a dvector of tuple from dftable.

```
auto d2 = frovedis::dftable_to_dvector<int,double,string>(t);
```

To use `dftable_to_dvector`, you need to include `frovedis/dataframe/dftable_to_dvector.hpp`.

5.2 Select

Please look at "src/tut5.2/tut.cc".

```
auto t2 = t.select({"c2","c3"});
```

The select method selects only specified columuns. Here, actual copy does not happen. Columns are managed as `shared_ptr` in dftable and only `shared_ptr`s are copied. With the dataframe operations, columns are not destructively modified.

You can drop the column by drop.

```
t2.drop("c2");
```

And you can change the name of the column.

```
t2.rename("c2","cx");
```

5.3 Filter

You can filter the rows by conditions. Please look at "src/tut5.3/tut.cc".

```
auto teq = t.filter(frovedis::eq("c1","c4"));
```

You can specify the condition of filter as the argument of the filter member function of the dftable. In this case, it filters rows where the value of clumn c1 and c2 are the same. The type of the comparing columns should be the same. Otherwise an exception will be thrown.

There are other operators like neq (not equal), lt (less than), le (less than or equal), gt (greater than), ge (greater than or equal).

If you want to compare with immediate values, you can use operator with "_im".

auto teqim = t.filter(frovedis::eq_im("c2",30.0));

Again, the types of the comparing column and the immediate value should be the same. Please not the ".0" of the immediate value, which makes the type of the immediate value as double.

There are other other operators like neq_im, lt_im, le_im, gt_im, ge_im.

If the type of the column is string, you can use regular expression as the operator.

auto tisreg = t.filter(frovedis::is_regex("c3",".*c"));

The first argument of is_regex is the column name, and the second argument is the regular expression. The regular expression is processed by std::regex. In this case, the row with "c" matches with the regular expression. If you want to exlude the mached rows, you can use is_not_regex.

If you do outer join of two tables, it might create rows with null data. You can filter the row if it is null or not, by using is_null (or is_not_null);

You can also combine the conditions with and_op and or_op.

```
auto tandor = t.filter(frovedis::and_op
                       (frovedis::or_op
                        (frovedis::eq("c1","c4"),
                         frovedis::le_im("c2",30.0)),
                         frovedis::is_not_regex("c3",".*c")));
```

Actually, the type of the result of filter is filtered_dftable, not dftable. It includes the information selected rows as indices. When actual data is required (select, show, for example), the selected columns are actually created. This techniquie is called late materialization. Most of the case, you do not have to be aware of it. If you want to create dftable explicitly, you can use materialize() member function, which calls select of all the columns.

5.4 Sort

You can sort by a column by calling sort. Please look at "src/tut5.4/tut.cc".

```
auto t2 = t.sort("c2");
```

This sort is stable sort. If you want to sort by multiple columns, you can just call sort again for other columns.

```
auto t3 = t2.sort("c1");
```

The output should look like:

```
c1  c2  c3
1   40  g
1   50  a
1   50  h
2   20  d
2   30  e
2   40  f
3   20  b
3   30  c
```

Please note that if you want to sort c1 and c2, first sort by c2, and then sort by c1.

If you want to sort descendant order, you can use sort_desc.

Again, sort produces sorted_dftable, instead of sort. The other columns are materialized when it is required.

You can call sort even when the table is filtered_table, like this.

```
auto t4 = t.filter(frovedis::eq_im("c1",2)).sort("c2");
```

In this case, it is a bit faster than calling sort after creating dftable by materialize or select.

5.5 Join

You can join two tables. Please look at "src/tut5.5/tut.cc".

```
auto joined = t1.hash_join(t2, frovedis::eq("t1c3", "t2c1"));
```

In this case, t1 and t2 are joined where t1c3 of t1 and t2c1 of t2 are the same. The result should be like:

```
t1c1  t1c2   t1c3  t2c1  t2c2
3     30     1     1     a
8     80     1     1     a
2     20     2     2     d
2     20     2     2     b
4     40     2     2     d
4     40     2     2     b
6     60     2     2     d
6     60     2     2     b
1     10     3     3     c
5     50     3     3     c
```

Note that there are two rows that have 2 in t2c1. Since there are tree rows that have 2 in t1c3, there are 6 rows that have 2 in either t1c2 or t1c3 in the joined table.

You can use bcast_join instead of hash_join. The result is the same (except the order of the rows), but bcast_join is implemented by broadcasting the right table (or column to join, to be exact) , which is faster especially when right table is small, but may consume larger memory if right table is large. In the case of hash_join, both tables (or columns) are partitioned by hash values.

Like in the case of sort, you can use filtered_dftable as the inputs.

```
auto filtered_t2 = t2.filter(frovedis::is_not_regex("t2c2","d"));
auto filter_joined = t1.filter(frovedis::ge_im("t1c1", 2)).
  hash_join(filtered_t2,frovedis::eq("t1c3", "t2c1"));
```

Again, it is a bit faster than using materialized dftable.

In this case, the join is inner join; t1 contains a row where t1c2 is 4, which does not appear in the joined table because t2c1 does not contain 4.

We also supports outer join.

```
auto outer_joined = t1.outer_hash_join(t2, frovedis::eq("t1c3", "t2c1"));
```

The result contains a row like

```
t1c1  t1c2  t1c3  t2c1  t2c2
7     70    4     NULL  NULL
```

Here, t2c1 and t2c2 becomes NULL.

Lastly, if you want to join multiple table to one table, and if it is guaranteed that the right hand columns that are used for join has unique values, you can use star_join, which is more efficient than using hash_join or bcast_join multiple times.

```
auto star_joined = t1.star_join({&filtered_t2, &t3},
                                {frovedis::eq("t1c3", "t2c1"),
                                 frovedis::eq("t1c3", "t3c1")});
```

You need to give vector of pointer to the tables as the first argument. This is because you can use the pointer to filtered_dftable, which is derived from dftable. The second argument is the vector of joining conditions.

At this moment, star_join does not support outer join. In addition, these join operations only support equi join.

5.6 Group by

You can group the rows by the values of columns[s]. Please look at "src/tut5.6/tut.cc".

```
auto grouped = t.group_by("c1");
```

This groups the rows by the value of column c1. The returned type is grouped_dftable and it is a bit different from other tables. You need to explicitly call select to create dftable, and the select can only include column[s] that are used for grouping and/or aggretation of other column[s].

```
auto grouped_sum = grouped.select("c1", frovedis::sum("c3"));
```

In this case, first argument of select is the column that is used for grouping, and the second argument specifies the sum of the column c3. Now, grouped_sum is dftable and show() can be called. It should look like

```
c1  c3
1   120
2   90
3   50
```

In this case, "c3" is used the name of the column of sum. You can specify the name by using sum_as. In addition, You can specify multiple aggregations as the second argument of select.

```
auto aggregated = grouped.select("c1",
                                 {frovedis::sum_as("c3","sum"),
                                  frovedis::count_as("c3","count"),
                                  frovedis::avg_as("c3","avg"),
                                  frovedis::max_as("c3","max"),
                                  frovedis::min_as("c3","min")});
```

The result should look like

```
c1  sum  count  avg  max  min
1   120  3      40   50   30
2   90   3      30   40   20
3   50   2      25   30   20
```

Type of sum, max, min is the the same as the type of the column. The type of avg is is double. If the type of the column does not support these operations (in the case of string), exception is thrown. The type of count is size_t.

In addition, you can specify multiple columns for grouping, like:

```
auto multi_grouped = t.group_by({std::string("c1"), std::string("c2")});
auto multi_aggregated = multi_grouped.select({std::string("c1"),
                                              std::string("c2")},
                                             {frovedis::sum_as("c3","sum"),
                                              frovedis::count_as("c3","count"),
                                              frovedis::avg_as("c3","avg"),
                                              frovedis::max_as("c3","max"),
                                              frovedis::min_as("c3","min")});
```

The result should look like

```
c1  c2  sum  count  avg  max  min
1   0   30   1      30   30   30
2   0   20   1      20   20   20
3   0   50   2      25   30   20
1   1   90   2      45   50   40
2   1   70   2      35   40   30
```

5.7 Other operations

There are other miscellaneous member funcitons in dftable. Please look at "src/tut5.7/tut.cc".

The member functions num_rows() and num_cols() returns the number of rows and columns respectively. The member function columns() returns the list of column names as vector of string. The member function dtypes() returns the vector of column name and its type name pair.

The member function count() returns the count of the column (it is actually the same as num_rows()). The member function sum() returns the sum of the column. In this case, you need to provide the type of the column as the template argument of sum, because the type of the column cannot be known statically. The member functions max and min work similarly. The member function avg does not need the template argument, because its return type is always double.

You can add a column as a return value of a function. The calc member function takes new column name as the first argument, function object as the second argument, and column names as the other arguments. The values of the columns are given as the arguments of the function, and the return value of the function is used as the value of the new column.

```
struct multiply {
  double operator()(int a, double b){return a * b;}
  SERIALIZE_NONE
};

t.calc<double, int, double>("multiply", multiply(), "c1", "c2");
t.show();
```

In this example, the function multiplies c1 and c2 and creates new column multiply. Here, you need to give the type of return value and the types of columns as the template arguments. The number of columns that can be provided is limited to 6 in the current implementation.

The result should be like:

```
c1  c2  c3          multiply
1   30  2018-01-10  30
3   20  2018-03-13  60
```

```
3   30  2018-08-21  90
2   20  2018-02-01  40
2   30  2018-05-15  60
2   40  2018-07-09  80
1   40  2018-04-29  40
1   50  2018-06-17  50
```

As the utility function, we provides a function that converts date time string into time_t, and a function that extracts year, month, etc. from time_t.

```
t.calc<time_t, std::string>("time_t",
                            frovedis::string_to_time("%Y-%m-%d"), "c3");
t.calc<int, time_t>("year", frovedis::time_extract(frovedis::MONTH), "time_t");
t.show();
```

The function object string_to_time requires a format string that is used to convert the string into time_t. The conversion is implemented by strptime; please refer to the manual of strptime for format string.

The function time_extract is implemented as localtime to create struct tm; then tm_sec, tm_min, etc. are extracted. Please also refer to the manual of localtime for the result.

The result of the example should look like:

```
c1  c2  c3          multiply  time_t      month
1   30  2018-01-10  30        1515510000  0
3   20  2018-03-13  60        1520866800  2
3   30  2018-08-21  90        1534777200  7
2   20  2018-02-01  40        1517410800  1
2   30  2018-05-15  60        1526310000  4
2   40  2018-07-09  80        1531062000  6
1   40  2018-04-29  40        1524927600  3
1   50  2018-06-17  50        1529161200  5
```

Because of the specification of struct tm, January is assigned to 0, etc.