MixComp Specification: Framework and Developer

May 28, 2013

Contents

Abstract

This is a mini-report for global architecture of MixComp. We will make use of two terms "framework" and "developer". The framework provides a unified development environment whereas developer will refer to the coding part that will make use of the framework (without modifying the framework) to realize new mixture laws. In practice, this is just a separation of Interface (framework) and Implementation (developer).

1 Framework Introduction

The idea of unified framework is to integrate the existing and future clustering models. The things that can be expected from the framework are:

1. Unified environment for composite mixture model based on the independence assumption (amongst different mixture models).
2. Abstract(Interface) plug-in class that must be derived by developer to develop a new mixture law.
3. Facilitate creation of Rpackage, Web Interface, GUI and other High level functionalities.
4. Facilitate parallelization using distributed and shared memory models.
5. Take input from user and run the whole software.

The developer is not expected to change the functionalities provided by this framework but can only provide concrete behavior to these functionalities. Any new functionalities should be first introduced in the framework. Hence the framework is expected to evolve with time. The

developer is only expected to provide all the low level implementations needed by the framework without worrying about how these functionalities will be brought together to realize a composite mixture model. Hence in short, the developer can concentrate on development of existing and new mixture laws without worrying about how to run them in integrated(composite) environment. The developer is free to chose it's development environment (including numerical libraries for example STK++, Eigen, Lapack or anything more suitable to developer needs) and in no way will be restricted by framework. For example, a developer can re-factor their existing codes and fit them into the framework (Quentin existing codes will be a good test for it) or one can re-implement from scratch using this framework (development of simple models including Bernoulli and Gaussian(with diagonal co-variance matrix) with Serge on STK++ platform will be a good test for it). This is the most interesting feature of this architecture as it will allow to independently develop new models.

2 Plug-in Specification

The Plug-in specification includes description on how to define a new mixture law using the framework. To create a new mixture law, the developer must create a new class inherited from **IDeveloper** class. There are certain functions that are already defined in the **IDeveloper** class and hence can be called directly inside the inherited class. These functions are **protected** and hence cannot be called from outside (publicly). They are enumerated below:

- **int nbCluster()**
  This function will return the number of clusters.
- **int nbSample()**
  This function will return the number of samples.
- **int* classLabels()**
  This function will return the class labels.
- **double* proportions()**
  This function will return the class proportions.

Important Notes: None of the above functions are defined inside the constructor of your derived class and hence should never be called inside constructor. They are only functional after you register the object of your class as elaborated in section 3. These functions must be called whenever you need above variables. This will ensure that all the mixture laws in composite environment should have the same values for all the above variables. The **IDeveloper** class also contained following member variables, hence you must not redefine/redeclare them inside your inherited class as you will inherited these members automatically.

- **int nbVariable_ :** Number of variables. This value will be set inside the setData() function as explained in section 2.2.
- **char id_ :** Identification character. You must set this value inside the constructor. This value must match with the value of various corresponding columns name characters in your data file.

2.1 Interface specification

The **IDeveloper** class contains Interface methods that are mostly based on SEM-gibbs algorithm to allow for generalization. Most of these methods are abstract, hence the developer must

provide concrete implementation for them in the derived class. The functions ending with "() = 0" are abstract. The functions ending with "() {}" do nothing by default and the functions ending with "()" are already implemented in framework (the developer can of-course overwrite them in some cases for performance reasons.) The following interface functions are available:

- **virtual void initializeStep() = 0**
  This function must be use for initialization of mixture parameters. This method will be called only once in the very beginning.

- **virtual IDeveloper* clone() = 0**
  This is a standard clone function in usual sense. It must provide a new object of your class with values of various parameters equal to the values of calling object. In other words, this is equivalent to polymorphic copy constructor.

- **virtual void copy(const IMixture&) = 0**
  This function must be defined in derived class to provide with copy semantics.

- **virtual void imputationStep() {}**
  This function should be used for Imputation of data. The default implementation (in the base class) is to do nothing.

- **virtual void samplingStep() = 0**
  This function must be used for simulation of all the latent variables and/or missing data excluding class labels. The class labels will be simulated by the framework itself because to do so we have to take into account all the mixture laws.

- **virtual void paramUpdateStep() = 0**
  This function is equivalent to Mstep and must be defined to update parameters.

- **virtual void storeIntermediateResults(int iteration) {}**
  This function should be used to store any intermediate results during various iterations after the burn-in period. The **iteration** argument gives the iteration number beginning after the burn-in period.

- **virtual void finalizeStep() {}**
  This step can be used by developer to finalize any thing. It will be called only once after we finish running the SEM-gibbs algorithm.

- **virtual double posteriorProbability(int sample_num,int Cluster_num) = 0**
  This function must be defined to return the posterior probability (PDF) for corresponding sample and cluster.

- **virtual double logLikelihood() const = 0**
  This must be defined to return the current log-likelihood.

- **virtual int freeParameters() const = 0**
  This function must return the number of free parameters.

- **virtual void setData() = 0**
  This function must be defined to set the data into your data containers. To facilitate data handling, framework provide templated functions, that can be called directly to get the data. Data-Handling is explained in more details in section 2.2.

- **virtual void writeParameters(std::ostream&) const {}**
  This function can be used to write summary of parameters on to the output stream. This is useful for debugging purposes but need not be defined necessarily. The default implementation is to do nothing.

There is no notion of Estep, Mstep, Sstep and Cstep in the plug-in part. These names will be used inside the framework to make it generic enough to expand for future.

2.2 Data Handling

As briefed in previous part, data must be set inside the virtual function **setData()**. The framework provides templated classes for data handling. The psedo code for **setData()** is given below:

```
virtual void setData()
{
  Data<type> mydatahandler;
  STK::Array2D<type> data = mydatahandler.getData(id_,nbVariable_);
  // Fill up your data container using data
}
```

You should replace "type" in above pseudo code with you own data-type which can be "double", "int" or "bool". Also this function will set the variable **nbVariable_** as it is passed by reference.

3 Writing your main function

This section explains how to write your main function to use the library. Also for illustration, refer to the example file **framework.cpp** in the source codes.

1. **Framework information:** Create an object of struct **FrameworkInfo** and fill in various input values.
2. **Create framework:** Create an object of class **MixtCompFacade** by passing **FrameworkInfo** object into it's constructor.
3. **Instantiate mixtures:** Create objects of all the mixtures.
4. **Register mixtures:** Register all the objects of your mixtures with Framework using it's member function **registerMixture(IMixture&)**.
5. **Instantiate framework:** Instantiate framework by calling it's member function **instantiateFramework()**.
6. **Run the framework:** Finally run the framework by calling it's **run()** member function.

4 stk++ Statistical Models

TODO

A Coding conventions

For the coding we use the java coding convention. All data members are defined with an underscore at the end of their name, like `data_`. Data members referring to pointer start with `p_`. Data members that are vector should start with `v_` and that are matrices should begin with `m_`. All alternatives should be enclosed in `enum`. The notations proposed in this document can be modified if needed or enclosed in namespace in order to avoid name collision.