Reconstruction of Epsilon-Machines in Predictive Frameworks and Decisional States

Documentation for the reference code.

Nicolas Brodu <nicolas.brodu@numerimoire.net>, 2009-2011.

Table of Contents

1 What this is about

You will find here the documentation for the code corresponding to the "Reconstruction of Epsilon-Machines in Predictive Frameworks and Decisional States" article, and what you need to start your own experiments using it. This is about :

- The reconstruction of ε-machines.
- How to get sub-machines relevant to the user in terms of a "Utility" function.

If you do not know about ε-machines you may consider them at first as Markovian automata where the series of data is emitted on state-to-state transitions (edges). This gives them very nice properties compared to classical Hidden Markov Models, where the data is output with a probability distribution attached to each state (and not to the transitions). In particular the ε-machine is the minimal deterministic automaton able to statistically predict the data optimally. You can also estimate the ε-machine structure from data: number of internal states, their transitions, and the probabilities of being in each state and of taking each transition. The above article contains a short explanation as well as references to the whole theory. Cf Crutchfield and Shalizi papers, references are given in the main article [11, 23].

For **quick recipes** and **ready to use programs** see the **next section "Quick recipies for the impatient"**

For people who prefer to **see what the code looks like** skip to **Section 5, "Putting it all together"**

2 Quick recipies for the impatient

If all you want is to try this code in place of CSSR [25] just run the “SymbolicSeries” example. It will tell you what arguments it needs and will reconstruct an ε-machine out of your data.

Is your data a discrete time series, or can it be made discrete in an natural way? Then discretise it and run the “SymbolicSeries” example.

Is your data a continuous time series? Try the “TimeSeries” example and adapt the default parameters in the source code (past and future size, subsampling and Taken's style time-lag vectors, utility function). Pay attention in particular to the kernel width parameter, and not to under- or over- smooth your data. The best kernel width is usually determined a posteriori, not a priori. Say for example that you use the statistical complexity to discriminate between one kind of series and the others. The best kernel width is the one that gives the best discrimination accuracy, implicitly realizing the best bias/variance compromise for your specific problem, and not the kernel width that maximizes an a priori criterion like AMISE. You should thus explore a range of kernel width and optimize an a posteriori criterion. If you are not satisfied with the results try to apply a high-pass filter (even a bad one like first differences) or any detrending method. In fact the best results are obtained when the range taken by the data values doesn’t vary too much from the beginning to the end of the series. If nothing works or if you reach cpu or memory limits, try to discretize your series in a clever way and switch to the SymbolicSeries example.

Is your data spatial, or cast in a light-cone setup, or simply not a time series? See the image processing and the cellular automaton example and adapt them to your scenario. All you need is to cast your problem in a predictive framework: ex: Predict the center pixel from its neighbors. Then the equivalent of the “past” of the time series becomes the data you need for the prediction (ex: the neighbors in this case) and the “future” of the time series becomes the data you predict (ex: the center pixel in this case). The API is explicitly worded as “DataType” and “PredictionType” for this reason, not to be restricted to time series. You may even provide transitions and the associated symbols if you are in a discrete setup (see the CellularAutomaton for how to do that).

Do you want a Generative markov model relevant to your Utility function ? Then modify one of the above examples, you will have to implement your Utility function in C++. See section 4.4 for how to do that.

3 Code Design

The code is header-only, there is nothing to link with. All you need is to include the files in your project.

The code is highly templatized. This allows to use the same generic routines while adapting precisely to the user needs. Various algorithms may be used interchangeably, for example clustering algorithms, without impacting the other parts of the computation.

Auto-detection of C++ types and methods. The generic code adapts to your class, just declare what you need and unused parts (ex: symbols processing) are removed altogether without performance penalty.

Using this code is generally a matter of :

- Deciding on the meta-parameters you will use, how your data is represented, which algorithms to use, which template arguments. Hopefully the defaults classes cover a vast majority of cases and all you have to do is a few C++ typedef.
- Feeding the algorithm with your real data, reading file and the like.
- Calling the appropriate methods like computeCausalStates() in order to process your data.
- Processing the results, like outputting state series or filtered images.

The following sections will guide you through the first and third points. See also the provided examples.

4 Core objects and concepts

This section matches the algorithm description part 5.1 of the paper [1]. Please refer to the paper for further explanations.

4.1 Data management

4.1.1 DataSet = Observations

These are pairs of (some-observed-system-state, some-observed-result). Or in other words, a "past light cone" of all causal influence, and another "future light cone" of all possible consequences.

Say that you can encode the system state in a C++ string, and the results in a floating-point value. Then you may define your observations simply as:

```
typedef map<string, float> DataSet;
```

4.1.2 DataType = observed system state, PredictionType = observed result

The above approach is limited in that it does not allow you to specify much about the data you observed. A better approach is to specify what your data set consists of explicitly:

```
struct DataSet {
    typedef string DataType;
    typedef float PredictionType;
    ...
};
```

Now you say that you encode the system state relevant for predictions in strings, and that your predictions take the form of floating-point values. So far we have not introduced anything more than a naming convention compared to the map, but the main interest of doing so will be apparent in the next sections. Simply declaring the two types with the correct names is enough, the generic templates will recognize them.

4.1.3 Feeding observations to the algorithm

In the first example the generic templates know how to run through the map in order to get each pair of observations. But when you define your own class, you need to also provide a way to retrieve the observation pairs. This loosely corresponds to the concept of the iterator, albeit with less constraints than in the C++ standard containers.

```
struct DataSet {
    ...
    typedef int iterator;
    iterator begin() {return 0;}
    iterator end() {return num_observations;}
    DataType& data(iterator it) {return some_data_array[it];}
    PredictionType& prediction(iterator it) {return some_prediction_array[it];}
};
```

Suppose that you have stored your observation pairs in two arrays. The above code tells the generic algorithm how to access your data. Notice that you just used a simple integer as an index, without loss of performance compared to directly accessing the arrays (all the code will be inlined by the compiler).

In the first example using a map the iterator is simply the standard container iterator. When the methods "DataType& data(iterator it)" and "PredictionType& prediction(iterator it)" exist in your class they are used (you may also return copies instead of references if you wish). Otherwise it->first and it->second are used instead. You could for example rewrite the first example as:

```
typedef vector< pair<string, float> > DataSet;
```

Which, depending on your application, is probably more efficient than the map. In a large-scale application you may also consider writing an iterator object that advances in a file which is read on demand. Just adapt the iterator concept to your needs.

4.1.4 Symbolic series

An ε-machine is a Markov automaton with labeled transitions. When you think about it, your data need not be *explicitly* represented in a string of symbols, although there will of course be an implicit equivalent symbol series for any consistent data value if you follow the ε-machine transitions.

Therefore all you need is to tell the algorithm which symbol is emitted when passing from one data value to the next, and we preserve the genericity: you may encode your data in integers for example, and compute on the fly which symbol is emitted. This is what's done in the SymbolicSeries example:

```
struct DataSet : public vector< pair<uint64_t,uint64_t> > {

    typedef uint64_t DataType;
    typedef uint64_t PredictionType;
    typedef char SymbolType;

    // Symbol emitted when passing from a to b
    bool getSymbol(DataType a, DataType b, SymbolType& symbol) {
        // See the SymbolicSeries program, here we extract the emitted symbol
        // The 64-bit value contains all symbols as a number in a nsymbols base
        symbol = symbols[b % nsymbols];
        // The symbol is valid only if both data values are consistent.
        // Here we consider the possibility of multiple observed sequences,
        // and therefore discontinuities are possible.
        return a % discard_first_symbol_factor == b / nsymbols;
    }
};
```

When the "SymbolType" exists within the scope of the DataSet structure (or the transition feeder, see the next section), then symbol processing is turned on in the generic algorithms. You need to define a "getSymbol" method that accepts two data values (by copy or reference, it does not matter), a symbol by reference (it needs to be modified on return), and that returns a boolean telling whether the transition is valid or not.

Providing symbols is optional for computing the decisional states, but mandatory for computing the full ε-machine with labeled transitions. As of now the only way to provide symbols is to write your own class. This can be as simple as inheriting from a standard container and adding the symbol processing, as is done in the above example.

4.1.5 Explicit transitions

Note that in the previous example transitions are considered between the successive entries returned by your iterator. This is how the constraints on the causal states (in order to keep the ε-machine deterministic) are automatically determined from these transitions : some data values will be put in the same causal state, and some causal states will be split, depending on the implicit relations imposed by the symbol transitions. See the paper [1] for details.

Suppose now that you have recorded multiple sequences of the same physical process as separate time series. You would not want to introduce spurious transitions between the end of one series and the

beginning of the next series. This is handled in the previous example by returning false in the getSymbol method. However this simple example does not generalize well (see the image processing example) and you would be better off specifying explicitly which are the transitions between the values in your data set. This can be done with a TransitionFeeder, a class that shall respect the following API:

```
struct TransitionFeeder {
    typedef … iterator;
    iterator begin() ;
    iterator end() ;

    // The data values can be returned by copy or reference
    DataType& getDataBeforeTransition(iterator idx);
    DataType& getDataAfterTransition(iterator idx);

    // Optional, only if you use symbols
    typedef … SymbolType;
    SymbolType getSymbol(iterator idx);
}
```

You can feed this class to the main Analyser object and it will be used instead of the data set iterator in order to provide the transitions between data values. See the CellularAutomaton example.

4.2 Probability distributions management (using the provided classes)

The distribution manager is responsible for gathering data values X and predictions Z, in order to build the conditional probability distributions p(Z|X).

The distribution type as well as how to manage them, compare them, aggregate them, etc. is entirely customizable by the distribution manager.

If you have discrete data you may wish to use the provided distribution manager that handles discrete probability distributions:

```
typedef DiscreteDistributionManager<DataSet> DistManager;
```

See the SymbolicSeries example.

If you have continuous data you may start with the kernel distribution manager, which will perform Kernel Density Estimation in order to estimate the p(Z|X) distributions:

```
typedef KernelDistributionManager<DataSet, SimpleGaussianKernel<float>,
SimpleGaussianKernel<float>, Sampler> KDM;
```

As you see you have to provide Kernels for the Data and Prediction types, as well as a Sampler object. In this example we use the predefined `SimpleGaussianKernel<float>` object, which assumes that your data or prediction types are some floating-point values and/or a vector of floating-point values (the kernel detects whether your data is a float or whether it is a kind of vector of floats, and if so what is its dimension). The Sampler is a container of PredictionType objects that defines at which points to evaluate the p(Z|X) estimations when comparing distributions.

Another distribution manager is provided, which is especially useful for time series:

```
typedef JointDataManager<TypeTraits>::DataSet DataSet;
typedef JointDataManager<TypeTraits>::DistributionManager DistributionManager;
```

The joint data manager is an object that takes as argument the TypeTraits defining all the types and objects you need (e.g. kernel for density estimation, algorithms for seeking nearest neighbors, how to store probability distributions in memory...). One particular object is of interest, the Joint data type (see TimeSeries.cpp) :

```
struct TypeTraits {
    typedef JointData<n_past_samples,n_future_samples> JointType;
    typedef JointType::DataType DataType;
    typedef JointType::PredictionType PredictionType;
    ...
};
```

This class is tailored for time series where the (past_observation, future_observation) pairs are more naturally considered as a series of size (number_of_past_observations + number_of_future_observations). Of course it may also be used for images where you would more naturally consider the center and neighbor pixels as part of a larger square area containing both the center and its neighbors.

The joint distributions p(X,Z) are then estimated first, and marginalized in order to get p(Z|X). Usage is simple: given a series container (ex: vector<float>) you have the following facilities:

```
for (int i = JointType::begin(series); i<JointType::end(series); ++i) {
    allJointData[i-JointType::begin(series)] = JointType::at(i,series);
}
```

This loop shows how to use the `begin`, `end`, and `at` members of the JointType in order to convert the series into Joint data objects. See the TimeSeries.cpp example for further details.

If the default DistributionManger classes do not suffice for your needs you can write your own. Start by looking at the DiscreteDistributionManager example, and build upon it.

4.3 Clustering

Clustering is used for three distinct tasks :

- Building Causal States by gathering data with similar probability distributions p(Z|X).
- Building Iso-Utility states by gathering data with similar maximal expected utility values.
- Building Iso-Prediction states by gathering data with similar best prediction sets.

Two clustering algorithms are provided, corresponding to the two descriptions in the article [1] :

- Aggregation of similar objects. This algorithm is O(N) with N the data size. Its limitations include a dependence on the data iteration order as well as cluster shapes limited to balls with respect to the similarity measure.
- Connected components. This algorithm is $O(N^2)$. Its limitations include being slow, and being sensitive to spurious data that would link together two clusters with a single link.

The default algorithm is the first one, using 3 passes and data order randomization so as to greatly reduce the dependence on the data order limitation. You may specify different arguments or a different algorithm when calling the state computation methods :

```
analyser.computeCausalStates(ConnectedClustering<>());
```

or for specifying only 1 pass instead of the default 3 :

```
analyser.computeCausalStates(AggregateClustering<>(1));
```

The syntax is the same for the other methods computing the iso-utility and iso-prediction states.

If you wish to write your own clustering algorithm start by looking at the AggregateClustering example and build upon it.

4.4 Utility related objects

4.4.1 Utility function

The utility function encodes the user knowledge of how bad it is to make mistakes on predictions. As much as causal states represent the underlying dynamics of the system, inferred from data, the utility function brings information from the user that is external to the data. The associated decisional states can be viewed as the pattern brought by the utility function on top the causal states. See the paper [1] for a longer discussion.

The utility function is from the C++ perspective a two-argument function or functor acting on PredictionType. For example, it can be the following functor :

```
struct Utility {
    // the arguments may be by value or (possibly const) reference
    double operator()(PredictionType prediction, PredictionType what_happens) {
        return utility_cost_table[prediction][what_happens];
    }
};
```

In this example the function is coded as matrix which explicitly lists the utility gained (or cost incurred) by having predicted something while something else actually happens.

Of course plain functions are also supported :

```
float SqUtility(float y, float z) { return -(y-z)*(y-z); }
```

Other examples could be a call to an external program, a physical device on a PCI bus, etc.

4.4.2 Optimiser

Recall the iso-utility and iso-prediction states construction [1]:

- The iso-utility states are the equivalence classes of data values that lead to the same maximal expected utility. The exact predictions for each data value in the state may differ.
- The iso-prediction states are the equivalence class of data for which the maximal expected utility is obtained at the same set of points. The exact utility for these points may differ for each member of the state.
- (the decisional states are the intersection of both, but this is out of scope here)

The expected utility function is automatically defined by using the Distribution Manager : expected_utility(y) = Expectation_over_z (utility (y, z)).

The next step is thus to apply a multi-modal optimiser on that expected utility function to infer the maximal expected utility and the set of best points. This is the task of the Optimiser class.

Some default classes are provided :

- ExhaustiveOptimiser: run through a preset range of z values. That range is given using a container object, like a std::vector<PredictionType>.
- GAOptimiser: apply a very basic genetic algorithm to get the results. Actually this is a proof-of-concept, but it was tested in real conditions and shall work. You may wish to review the results closely though: unlike the exhaustive optimiser the genetic algorithm provides some approximate solutions. Increase for example the population size or the number of generations if obtained accuracy is below what's required by your application.
- NullOptimiser: no optimisation. You will not be able to call utility-related function. This is the default when only the causal states are needed, as in the SymbolicSeries program.

You can write a new optimiser by respecting the following API :

```
struct Optimiser {

    // Some type like std::set<PredictionType>.
    // A helper template class BasicPredictionSet<PredictionType> is provided
    // for you. It behaves as a shared_ptr of std::set<PredictionType> and
    // additionnally handles aggregation of sets for the clustering algorithm.
    typedef … PredictionSet;

    // The functor given here is the expected utility function, a one-variable
    // function of PredictionType returning a floating-point value.
    // You should return the best utility found by your multi-modal optimiser
    // and fill the referenced prediction set with the corresponding points.
    template<class Functor> double optimise(Functor f, PredictionSet& res) {…}
};
```

The CellularAutomaton program illustrates with a simple case how you can write your own optimiser.

5 Summary : Putting it all together

This section recapitulates the previous points with a toy example. The scenario is of predicting whether it will be sunny or raining based on observations from the past 3 days. Based on this prediction we plan each day to uncover or keep shut a tent containing a collection of mirrors concentrating sun rays on a solar oven.

Let's first declare the data types and meta-parameters. We decide to encode the past three days state into a 3-bit field : 000 = rainy-rainy-rainy, 001 = rainy-rainy-sunny, … 111 = sunny-sunny-sunny. These values fit on an unsigned 8-bit integer. From this we wish to predict a boolean value, whether it will be sunny or not. We decide to store all observations in a vector. The data set can thus be defined:

```
typedef vector< pair<uint8_t, bool> > DataSet;
```

Given the nature of the data, we will simply invoke the default discrete distribution manager:

```
typedef DiscreteDistributionManager<DataSet> DistManager;
```

We have in this example a utility function. If we open the tent when it is sunny we can benefit from the solar oven, so we have utility(planned=sunny, real=sunny)=1. When we open the tent but it rains we do not benefit from the oven and we will have to clean up the mirrors in the evening, a costly operation, so utility(planned=sunny, real=rainy)=-1. When we keep the tent shut we gain nothing so we set a utility of 0. Let's define this function simply :

```
int utility(bool predicted, bool real) {
    if (predicted && real) return 1;
    if (predicted && !real) return -1;
    return 0;
}
```

In order to optimise the expected utility we can invoke the default exhaustive optimiser:

```
typedef ExhaustiveOptimiser< vector<bool> > Optimiser;
```

We have to provide to this object the range of all possible prediction values, in this case a vector containing two elements true and false. See Section 4.4.2 and the CellularAutomaton for a perhaps more elegant way of defining the optimiser. This is out of scope of the toy scenario.

Now let's define the main analyser object:

```
typedef DecisionalStatesAnalyser<DistributionManager, DataSet, Optimiser> Analyser;
```

We are done with the declarations of C++ types, and we can now build objects of these types:

```
DataSet dataset;
DistManager distManager(dataset);
vector<bool> all_predictions(2); all_predictions[0]=false; all_predictions[1]=true;
Optimiser optimiser(all_predictions);
Analyser analyser(distManager,dataset,optimiser);
```

Assuming you have filled the data set with your observations (ex: collected in a file), then we can compute the causal states :

```
analyser.computeCausalStates();
```

Then we can apply the utility function on top of these states:

```
analyser.applyUtility( &utility );
```

And now we can compute the iso-utility, iso-prediction, and finally the decisional states

```
analyser.computeIsoUtilityStates();
analyser.computeIsoPredictionStates();
analyser.computeDecisionalStates();
```

Note that we used the default clustering algorithm with default parameters, see Section 4.3.

We have computed the states, now is the time to do something with them. You could for example compute the statistical complexity of the system:

```
double C = analyser.statisticalComplexity();
```

Or perhaps, display the local decisional complexity values:

```
for (int i=0; i<dataset.size(); ++i) {
    uint8_t observed_state = dataset[i].first;
    cout << analyser.localDecisionalComplexity(observed_state) << endl;
}
```

See the examples provided with the source code for other uses, like filtering an image or writing the series of inferred states in a file.

One particularly interesting operation is to compute the ε-machine. But notice that in this toy scenario we did not so far introduce the notion of symbols. Since we have declared the data set using a standard vector container, we will provide the symbols here with a transition feeder (see Section 4.1.5).

```
struct TransitionFeeder {
    // we will simply access the data set elements by their indices
    typedef int iterator;
    iterator begin() { return 0; }
    // but the last transition is between the two last data set elements
    iterator end() { return dataset.size()-1; }

    // our constructor needs the data set reference
    DataSet& dataset;
    TransitionFeeder(DataSet& ds) : dataset(ds) {}

    // We assume the vector has no gap, all entries are transitions
    uint8_t getDataBeforeTransition(int idx) { return dataset[idx].first; }
    uint8_t getDataAfterTransition(int idx) { return dataset[idx+1].first; }

    // Now the interesting part. We will use text symbols as an illustration
    typedef char SymbolType;
    SymbolType getSymbol(int idx) {
        // Recall that we encoded the last 3 days as 3 bits: 000, 001, etc.
        // The emitted symbol corresponds to the last bit of the new day.
        // This is easily extracted with a bit mask (value AND 001)
        uint8_t symbol = getDataAfterTransition(idx) & 1;
```

```
        // we return the symbol as a character in this example
        if (symbol) return '1';
        return '0';
    }
};
```

Using this transition feeder is easy:

```
TransitionFeeder feeder(dataset);
```

And the above code for declaring the analyser now uses the transition feeder instead of the data set:

```
Analyser analyser(distManager,feeder,optimiser);
```

The causal states computation is done by calling `analyser.computeCausalStates();` exactly as before, but the symbols are now taken into account for making the ε-machine deterministic. The ε-machine itself can be computed simply:

```
analyser.buildCausalStateGraph();
```

And we can write it on the standard output as a graph in the "dot" format:

```
analyser.writeCausalStateGraph(cout);
```

This simple toy scenario was designed to set you up quickly for using the main objects in your own code. Please see also the examples provided with the source code.