Automated test generation for `ctsa`

@hgfernan

January 30, 2025

Contents

Abstract

A routine for the automated test generation for the statistical library `ctsa` is outlined. It envolves the generation of simple tasks of model fitting and prediction using `ctsa`, compared with equivalent code in the Python libraries `pmdarima` and `statsmodels`, and in the R library `forecast`.

1 Motivation

Why using a test database and automatic generation of tests, instead of the handcrafted tests ?

2 Introduction

Overview of the project

3 Tables

Since the automated test generation is based on a database, here follows a description of the database to be used.

It has a mixed relational and document architecture: the main tables follow a conventional relational structure, but the parameters and test results are stored as JSON values. That's for pragmatic reasons: the parameters and test results are varied and have different structures. That could be easily mapped to a relational database structure, but it would be too laborious and cumbersome.

For instance: an $ARIMA$ model will have three basic parameters, while a $SARIMA$ model will have 7 basic parameters.

As such, test parameters and results will be stored as JSON objects encoded as strings, and dealt with by classes specialized in their content. There will be a class for each one of the models, that will be able to unpack and allow the use of the information in those JSON values.

In a few words, the algorithm to generate tests follow this way:

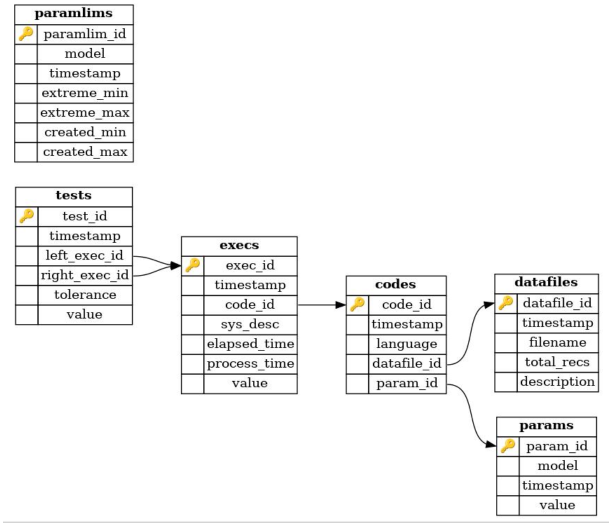

Figure 1: Database used for automated test generation

1. The table `paramlims` (on the upper left corner of Figure 1) is used to select a sequence of parameters for each model, according to the limits present in the fields `extreme_min` and `extreme_max`. They are stored in the table `params` to be detailed below.

   Not all parameters are generated, and the range of the parameters created up to the moment are saved in the fields `created_min` and `created_max`.

   The field `timestamp` contains the last alteration of any or all values in the range `created_min` and `created_max` for its corresponding `model`;

2. The list of data files available (stored in the folder `data/`) are listed and each name is contained in the field `filename` of the table `datafiles`. In this table the date of each file inclusion is recorded in the field `timestamp`. The field `total_recs` of the same table contain the number of records of each file. The field `description` can be used to introduce details of the file: its origin, the transformations used to generate it, etc.

3. There are simple templates for each model, and they're used to generate a single file for each of the elements of the cartesian product between the allowed range of `params` for each model, and the data files in `datafiles`.

   This process is replicated for each of the languages in use (C, Python, and R) and their corresponding libraries `ctsa`, `pmdarima` and `statsmodels`, and `forecast`. Those results are stored in the table `codes`, and only the programming language is stored there, as just a single library will be used for each case.

   After the testing *per se*, a fragment of the code stores in a CSV file the parameters, the calculated results and the elapsed and processing times, as well other information needed, as described in item 4;

4. The codes generated are then run, as the resources in processing hardware and time allow for it.

   The execution results are stored in the table `tables`, and they are retrieved from a CSV file that each execution generate, as described in the item 3.

   This table contains the field `timestamp` that will inform when the execution was started. The field `code_id` refers to the source code that was used. The field `sys_desc` should contain a description of the hardware that was used to run the code, as precise as possible. Even though precision in the results is the main purpose of this suite of tests, a marginal point is the processing speed of each implementation. Of course, that comparison would be meaninful only if the execution scenarios of each benchmark participant were kept constant.

   The table also contains the fields `elapsed_time`, that contains the clock time spent for the execution, and `proc_time` that contains the time spent using the processor or processors available in the hardware.

Finally, the field `value` is a JSON value encoded as a string that contain two data structures, named `params`, where a copy of the model parameters is stored, and `results`, where a copy of the numerical parameters is stored.

There's some space for the results of a model – for instance, the number of time steps forecasted –, but the intelligence to handle that variation will be kept in the class that handles each results;

5. Finally the table `tests` holds what could be considered a test, actually a comparison between two executions, or fields in the table `execs`.

   The field `timestamp`, as usual, contains the information of the time when the comparison was started. The field `left_exec_id` identifies the execution that will be compared against other execution, represented by the field `right_exec_id`. Typically, it will be an execution of a program based upon `ctsa` and another based upon one of its equivalent libraries using R or Python.

   And it is recommended that the comparisons are made only between two executions in the same hardware, described by `sys_desc`, as mentioned in the item 4.

   The comparison are executed against the value given in the field `tolerance`. The relative deviation d_{rel} is calculated for each parameter as

$$d_{rel} = \frac{|p_{right} - p_{left}|}{p_{right}}$$

   And a test passes when $d_{rel} \leq tolerance$; otherwise it fails.

   The field `value` is a JSON value encoded as a structure, that contains two data structures, one named `params`, where a copy of the model parameters is stored, and another named `results`, that contains for each result a triplet with the numeric value of the result of the left test, the value of the result of the right test, and string "`pass`" if the test passed, or "`fail`" if the test failed.

4 Software design

High level specification of the softwares and their use.

5 Examples by hand

Worked examples of the database and software use.

6 Implementation

A shiplog reporting details of the system development, and possible deviations from the specification in the section Software design.