PILOTS (version 0.4): Tutorial

Turorial authored by David Glowny, Richard Klockowski

PILOTS software contributors: Sida Chen, Alessandro Galli, David Glowny, Shigeru Imai, Richard Klockowski, Wennan Zhu

Contents

PILOTS: Tutorial index **2**

PILOTS: Examples **3**
- Introduction . . . 3
- Twice (real-time mode) . . . 3
- Twice (simulated mode) . . . 5
- SpeedCheck . . . 6

PILOTS: Programming **8**
- Introduction . . . 8
- Grammar . . . 8
- Basic example . . . 10
- Example with error detection/correction . . . 10
- Example with prediction . . . 11
- Additional Notes . . . 12

PILOTS: Running **13**
- Introduction . . . 13
- Input data file format . . . 13
- Real-Time mode vs. Simulated mode . . . 14
- External software components . . . 14
- Compiling and running a program . . . 14

Learning Model Overview **16**
- Installation of Necessary Packages . . . 16
- Build a PILOTS jar file . . . 17
- Configure aliases . . . 17
- Training a Learning Model (Offline) . . . 17

Add Data for Producers 18
Running Learning Model Server 19
Running the plt program with Producer and Output Handler 19
(If using simulation) Output Producer Data to Output Handler 19
Learning Model Examples 19
Additional Notes 20

Simple Linear Regression Example **21**
Build a PILOTS jar file 21
Configure aliases 21
Add definition for linear regression model: 21
Run the learning model server: 22
Add Data for Producers 22
Run the Output Handler: 22
Compile and Run the PLT file: 22
Run the Producer: 23
Output simulated data to the Output Handler: 23
Final Output 23

Simple Bayes Classifier Example **25**
Build a PILOTS jar file 25
Configure aliases 25
Add definition for dynamic Bayes classifier model: 25
Run the learning model server: 26
Add Data for Producers 26
Run the Output Handler: 26
Compile and Run the PLT file: 26
Run the Producer: 27
Output simulated data to the Output Handler: 27
Final Output 27

PILOTS related publications **29**

Appendices **30**
Predictive Server 30
Installation (Linux/MacOS) 30
Quick Start 30
Definition Files 31
Server 32
Example 33

PILOTS: Tutorial index

The tutorial for PILOTS (version 0.4) is broken down as follows:

- Basic Examples – how to run the basic PILOTS examples (without learning model) provided in the distribution.
- Programming – how to write and compile your own PILOTS programs.
- Running – how to run a PILOTS program using external software components.
- Learning Model Overview – how to run a PILOTS program with machine learning models *(optional)*.
- Simple Linear Regression Example – a detailed walkthrough of constructing and implementing a linear regression model in PILOTS.
- Simple Bayes Classifier Example – a detailed walkthrough of constructing and implementing a dynamic online Bayes classifier model in PILOTS.

For more information, visit the PILOTS homepage at: https://wcl.cs.rpi.edu/pilots/

PILOTS: Examples

Introduction

This section of the tutorial deals with running the examples from the associated publications. The source code for these examples can be found in *$PILOTS_HOME/examples*; each example comes precompiled in the PILOTS distribution. Please note that each example comes in two flavors: corrected and uncorrected. The instructions below are for running uncorrected examples. In order to run the corrected version of the examples simply run the script that ends with "Correct."
The examples include:

1. Twice (real-time mode)

    - Two input streams (a,b) where, at any given moment, the value of the data on input stream b is twice the value of the data on input stream a. The output stream o, computed as b-2*a, should continuously produce zero under normal circumstances.

2. Twice (simulated mode)

    - Similar to the real-time mode, but the input data is generated via a file (in *$PILOTS_HOME/11-May-2013_Twice*) using a simulated time counter.

3. SpeedCheck

    - A simulated flight-monitoring application that uses real data recorded on April 3rd, 2012 during a flight from Albany, NY to Tipton, MD. The application monitors the air speed, wind speed, and ground speed of the moving object to detect when there is a failure in any one (or a combination) of the sensors.

Each of these examples make use of external software components provided in the PILOTS distribution. These will be described in more detail in the running tutorial.

Twice (real-time mode)

Twice is a very basic streaming application with two input streams a and b. At every given point in time, the value on stream b should be twice the value on

stream a. We use PILOTS to easily produce this application. Furthermore, we were able to observe the PILOTS application to produce error correcting code. This is the first example that is used to exemplify error signatures.

Before running the executable scripts, complete the following steps for compiling the plt file:

1. Run the following command to compile the PILOTS program to Java source code:

```
$ plc Twice.plt
```

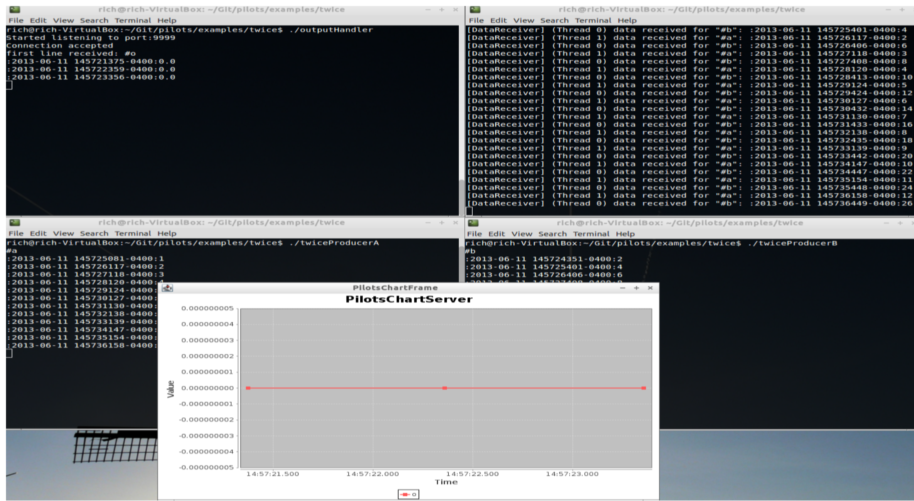
2. Run the following command to compile the generated Java source code:

```
$ javac Twice.java
```

Finally, in order to run this example, open four identical terminals looking at the *$PILOTS_HOME/examples/twice/* directory. Be sure that your classpath includes *$PILOTS_HOME/lib/** in each of these terminals. Running the PILOTS application is a three step process. First the data sink (in this case, an external software component to visualize output) must be initialized. Next the application and finally all data sources. In this example the data source is an external software component. We have provided scripts that make executing these examples more convenient. The three steps described above are summarized as follows:

1. On one terminal, run the *outputHandler* script. This will initialize the data sink and wait for the PILOTS application to produce an output stream.
2. On the next terminal, run the *twice* script. This will initialize the PILOTS application, connect with the output handler's socket, and start listening for input data.
3. On the last two terminals, run the *twiceProducerA* and *twiceProducerB* scripts (respectively). Try to make sure that these two processes are initialized as close as possible. Starting one a long while after the other will cause an out of sync error in the PILOTS application.

Here is a screenshot of the Twice application in action:



In order to simulate a failure in one of the streams, kill the data producing process. For example to simulate a failure in stream b you can activate the terminal that is running *twiceProducerB* and kill the proces (using CTRL+C). These errors produce characteristic patterns which we describe using error signatures. Error signatures will be discussed more in the programming tutorial

---

Twice (simulated mode)

This example is functionally equivalent to the real-time version of Twice. However in the simulated mode, all data is generated from a file. Thus all data is known ahead of time, and there is no waiting for streams to produce new data.

Before running the executable scripts, complete the following steps for compiling the plt file:

1. Run the following command to compile the PILOTS program to Java source code:

```
$ plcsim Twice.plt
```

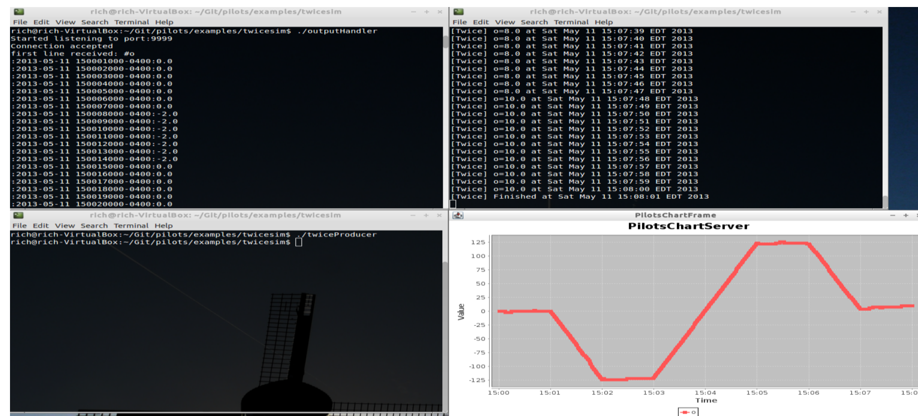
2. Run the following command to compile the generated Java source code:

```
$ javac Twice.java
```

Finally, running the simulated version of Twice is slightly different than running the real-time version. Only three terminals need to be opened, all directed to *$PILOTS_HOME/examples/twicesim*.

1. On the first terminal, run the *outputHandler* script.
2. On the second terminal, run the *twicesim* script.
3. On the third terminal, run the *twiceProducer* script. This will send all of the data from the files in *$PILOTS_HOME/data/11-May_2013* to the PILOTS application.
4. Finally, to initiate the simulation activate the window that is running the *twicesim* script and press ENTER.

Here is a screenshot of the simulated Twice application after execution:



Note that failures are simulated by the input data themselves. The files in *$PILOTS_HOME/data/11-May-2013_Twice* were generated programmaticaly to exhibit certain errors at certain times (see the publication regarding error detection and correction for details).

SpeedCheck

This example is a more practical motivating example. Real data was recorded during an actual flight (see Quest paper for details). This data includes air speed (measured from plane's sensor), ground speed (measured from GPS), and wind speed (measured from weather reports). Using the principles of relative forces we are able to derive equations for each of these speeds in terms of the other two. The point of this example is to show that under certain erroneous conditions, such as a pitot tube (airspeed sensor) freezing, it is possible to recompute erroneous data from redundant counterparts. This application also uses simulated mode, the data can be found in *$PILOTS_HOME/data/03-Apr-2012-KALB-KFME*. Three terminals need to be opened all pointing to *$PILOTS_HOME/examples/speedCheck*.

Before running the executable scripts, complete the following steps for compiling the plt file:

1. Run the following command to compile the PILOTS program to Java source code:

```
$ plcsim SpeedCheck.plt
```

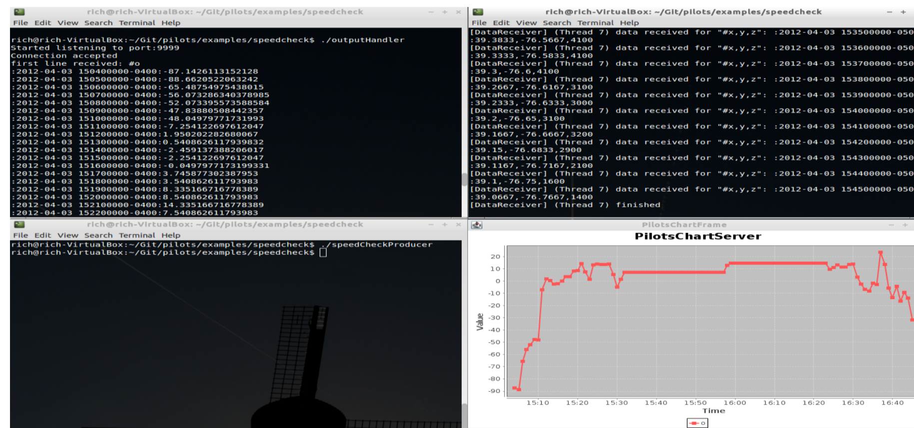
2. Run the following command to compile the generated Java source code:

```
$ javac SpeedCheck.java
```

Finally, use the following steps to execute the example:

1. On the first terminal, run the *outputHandler* script.
2. On the second terminal, run the *speedCheck* script
3. On the third terminal, run the *speedCheckProducer* script. Then, in the terminal window that is running the *speedCheck* script, press ENTER.

Here is an screenshot of the SpeedCheck application after execution:



In order to simulate different errors (such as airspeed sensor failure), replace the *speedCheckProducer* script with one of the other *speedCheckProducer_** scripts.

PILOTS: Programming

Introduction

PILOTS is a programming language designed to enable a high level specification of streaming applications which handle spatio-temporal data. Specifically, PILOTS applications are able to monitor heterogeneous input streams and combine their data to produce output streams. The computational options for combining input data is limited due to the experimental nature of this language. Instead the focus of PILOTS applications should be on the input data and how they are related to each other. Currently as of v0.2.x, the input data for PILOTS applications is limited to the double type and can be generated in two ways: from a file with a special format or from a periodic thread which produces linear data. Input data generation falls under the responsibility of external software components, which will be discussed in the running tutorial.

With a high level specification, PILOTS programmers can rapidly prototype experiments on spatio-temporal objects in real time. The language incorporates a data selection model, and application model, and an error correction model.

The data selection model views discrete data points as a continuous spatio-temporal domain.

The application model designates what is to be done with the input data: how to compute the output stream data.

The data correction model allows input redundancies to be made explicit so that a correction may be made in case of a particular failure.

A PILOTS program is compiled directly into Java (using Java CC) which utilizes TCP/IP sockets to handle input and output. A PILOTS application, which is the compiled version of the program, runs on Java so it is compatible cross-platform as long as Java is installed. This tutorial will explain how to write a PILOTS program. First the PILOTS grammar is introduced, which is similar in style to Pascal. Next a very simple example, called *Squared* is demonstrated as an equivalent to the famous "Hello World!" program.

Grammar

The grammar of the language is separated into clauses which describe the inputs, outputs, and error correcting mechanics of a program. A formal definition of the grammar can be seen in the figure below. Every PILOTS program *must* have an

inputs and outputs clause. Only programs which incorporate error correcting code need to include the errors, signatures, and correct clauses. The signatures clause enables both error correction and detection. A program without a correct clause is still capable of error detection, but not correction. The *Corrects* and *Correct* fields will be removed after version 0.3.1.

Program	:=	program Var; inputs *Inputs*; outputs *Outputs*; [errors *Errors*;] [signatures *Signatures*; [correct *Corrects*;]] end
Inputs	:=	[(*Input*;)* *Input*]
Input	:=	*Vars Dim* using *Methods*
Outputs	:=	[(*Output*;)* *Output*]
Output	:=	*Vars* : *Exps* at every *Time*
Errors	:=	[(*Error*;)* *Error*]
Error	:=	*Vars* : *Exps*
Signatures	:=	[(*Signature*;)* *Signature*]
Signature	:=	*Var*[*Const*] : *Var* = *Exps* [*Estimate*][*String*]
Estimate	:=	estimate *Var* = *Exp*
Corrects	:=	[(*Correct*;)* *Correct*]
Correct	:=	*Var*[*Const*] : *Var* = *Exps*
Vars	:=	*Var* | *Var*, *Vars*
Var	:=	{ a, b, c, ... }
Const	:=	(*Var*)
String	:=	{ “a”, “b”, “c” }
Dim	:=	‘(t)’ | ‘(t,x)’ | ‘(t,x,y)’ | ‘(t,x,y,z)’ | ‘(t,x,y,z,n)’
Methods	:=	*Method* | *Method*, *Methods*
Method	:=	(closest | euclidean | interpolate | predict) ‘(’ *Exps* ‘)’
Time	:=	*Number* (msec | sec | min | hour | day)
Exps	:=	*Exp* | *Exp*, *Exps*
Exp	:=	*Func*()*Exps* | *Exp Func Exp* | ‘(’ *Exp* ‘)’ | *Value*
Func	:=	{ +, -, *, /, sqrt, sin, cos, tan, abs, ... }
Value	:=	*Number* | *Var*
Number	:=	*Sign Digits* | *Sign Digits*‘.’*Digits*
Sign	:=	‘+’ | ‘-’ | ”
Digits	:=	*Digit* | *Digit Digits*
Digit	:=	{ 0, 1, 2, ..., 9 }

The input clause is where the spatio-temporal dimensions of each input variable is defined along with associated data selection operations. The possible data selection operations are *closest*, *euclidean*, and *interpolate*. The closest operation takes one dimension as a parameter and selects the closest available data in

that dimension. Euclidean is similar to closest, except that it takes multiple spatial dimensions and selects the data that is closest in euclidean distance to the queried location. Interpolate takes an extra parameter (denoted above as n) which denotes the number of nearby points to consider as part of a euclidean average. The specific dimensions that are used (out of {x,y,z,t}) within a PILOTS program are important. They determine whether a PILOTS application's data streams are spatio-temporal, or just temporal. All PILOTS programs must select on at least the 't' dimension, otherwise PILOTS may not be an appropriate language choice. Therefore all PILOTS applications (whether real-time or simulated) have a notion of "current time." If 't' is the only dimension that is specified, then application data is only temporal. However, if 'x','y', or 'z' are used then the application must be told what the "current location" is at the "current time." This is achieved by supplying a text file called "x-y-x.txt" which is processed before the input streams enter the data selection module. The exact format of the "x-y-x.txt" file will be described in the running tutorial.

Basic example

As an initial example we present Squared.plt, which simply squares input data. This example can be found precompiled in *$PILOTS_HOME/examples/squared*.

```
program Squared;
inputs
    x(t) using closest(t);
outputs
    o: x*x at every 1 sec;
end
```

In this application the selection module has received data for a variable named "x". The output is simply the input squared, which is produced every second. Presumably this application could be a link in a larger chain of similar applications where the output of one is the input of another.

Example with error detection/correction

```
program Twice;
inputs
      a(t) using closest(t);
      b(t) using closest(t);
outputs
      o: b-2*a at every 1 sec;
errors
      e: b-2*a;
signatures
      s0: e = 0                          "Normal mode";
      s1(K): e = 2 * t + K                "A failure"
         estimate a = b / 2;
      s2(K): e = -2 * t + K               "B failure"
         estimate b = a * 2;
      s3(K): e = K, abs(K) > 20     "Out-of-sync";
end
```

In this example input stream b is intended to always be twice that of stream a. We feature error detection and correction using error signatures depicted in the signatures clause. Each signature has a label followed by a pattern and an optional description string (which is used for debugging purposes). The pattern within the signatures is an expression that may include any input variable or error function. Additionaly, a constant K is used to denote the presence of an unknown. Internally this is handled by executing search over a (small) number of measured data points to find the best estimate for the unknown. Notice that the signature *s3* has a constraint after its pattern. Finally, the correct clause describes what actions to take when a signature has been detected. If a signature is not included in the correct clause then it is unrecoverable (or the "Normal mode" signature where no corrective action is necessary).

Example with prediction

```
program Twice;
inputs
      a, b(t) using closest(t);
      c(t) using predict(linear_model, a);
outputs
      o: b - c at every 1 sec;
end
```

In this example input stream b is intended to always be twice that of stream a. After the offline training process in machine learning component using existing data containing a and b, we use data stream a as parameter to the linear_model

to generate a new data stream c, which is an estimation of b. Finally, we output the difference of b and c to verify the correctness of the trained model linear_model.

Additional Notes

Example with error detection/correction before v0.3.1

```
program Twice;
inputs
    a(t) using closest(t);
    b(t) using closest(t);
outputs
    o: b-2*a at every 1 sec;
errors
    e: b-2*a;
signatures
    s0: e = 0                          "Normal mode";
    s1(K): e = 2 * t + K               "A failure";
    s2(K): e = -2 * t + K              "B failure";
    s3(K): e = K, abs(K) > 20          "Out-of-sync";
correct
    s1: a = b / 2;
    s2: b = a * 2;
end
```

Same function as "Example with error detection/correction" with different grammar

---

PILOTS: Running

Introduction

This part of the tutorial deals with running a generic PILOTS application. This can be particularly confusing because PILOTS is not very flexible, and depends on a very specific data format. If the PILOTS application is being run in simulated mode, and all data is originated from a file, then execution is relatively straightforward. Real-time PILOTS applications can be difficult to set up, unless an appropriate external software component is already in place. In this tutorial we will first introduce the PILOTS data format. Then the differences between real-time mode and simulated mode will be discussed. Finally, the external software components provided in the PILOTS distribution will be described.

Input data file format

PILOTS applications handle spatio-temporal input/output data of a specific format. All input/output in a PILOTS application looks like:

```
#var0, var1, ... varn \r\n
< space > : < time > :val0,...,valn \r\n
...
< space > : < time > :val0,...,valn \r\n
```

var0,...,varn identifies the name of the data being received from in the inputs clause of a PILOTS program.
val0,...,valn are the associated values of the variables for the given spatio-temporal coordinates.
<space> may have up to three dimensions (latitude, longitude, height) separated by commas, and <time> is specified in Java date/time format (using 24-hour time).
Additionally, each specification of <space> or <time> may be a single point or a range between two points separated by '~'. Examples of the spatio-temporal data format can be found in *$PILOTS_HOME/data/**

Real-Time mode vs. Simulated mode

At compilation PILOTS programs must be designated as real-time programs or simulated programs. The reason for this is that the PILOTS compiler generates different code for each situation. In a real-time program, time is associated with the processor's internal clock. This mode is appropriate for real-time systems where new input data is continuously arriving and future data is unknown. Simulated programs assume that all data for an experiment is available ahead of time, so processing can occur as fast as possible without busy waiting. We have found that both real-time and simulated mode are necessary and useful. Real-time mode is the obvious choice for most applications of stream processing. Simulated mode has the advantage of being able to quickly evaluate potential data error scenarios, which helps in devising error correcting code.

External software components

PILOTS applications assume all input and output are communicated over TCP/IP sockets. Thus PILOTS applications require you to run external programs that act as data input producers or output handlers. The PILOTS distribution provides simple input/output handler programs which can be found in *$PILOTS_HOME/pilots/util*.

- **ChartServer.java** is an output handler (data sink) that, given a port number, listens to a socket on that port and graphs the data over time (using jfreechart).
  Parameters: <port_number>
- **FileInputProducer.java** is an input handler (data source) that simply relays the contents of a file which contains spatio-temporal data over a designated socket on port 8888 of the local machine.
  arameters: <filename>
- **LinearInputProducer.java** is an input handler (data source) that generates temporal data incrementing linearly over time (with optional timing jitter). The data is relayed over a designated socket on port 8888 of the local machine.
  Parameters (all required, in order): <num_iterations> <variable_name> <initial_value> <increment_value> <random_timing_jitter>

Compiling and running a program

Compile your PILOTS program (*YourProgram.plt*) into Java source code by the following:

- $ java pilots.compiler.PilotsCompiler *YourProgram.plt*
  OR
  $ plc YourProgram (‘plc‘ is an alias for the above java command)
  OR
  $ plcsim YourProgram (if your program produces a simulation rather than using real-time streaming of data)

The command line options for the PILOTS compiler are:

- Dstdout : sends generated code to standard output instead
- Dsim : compilation switch for simulation mode
- Dpackage : changes the generated code’s target package

Compile the generated Java code. The result is the PILOTS application and can be run using Java.

- $ javac YourProgram.java

The resulting java class, *YourProgram.class*, is the executable pilots application. Note that you can not simply run *\“$ java YourProgram\”* because PILOTS applications use TCP/IP sockets for input and output. In order for the input and output to be received correctly, external software components must be configured to handle the data in the PILOTS spatio-temporal format. The external software components (found in *$PILOTS_HOME/pilots/util*) are elementary examples of input producers and output handlers.

To run an application, all data sinks must be initialized before sources. Thus the order of process execution must be:

1. output handlers
2. PILOTS application
3. data input producers

The IP addresses that will be used for socket communication between the PILOTS application and the external programs must be specified. The PILOTS application takes -input and -outputs arguments along with error detection parameters -tau and -omega as follows:

- $ java <your PILOTS application> -input=<port> -outputs=<ipaddr:port>+ -tau=<t> -omega=<w>

(Where + means one or more)

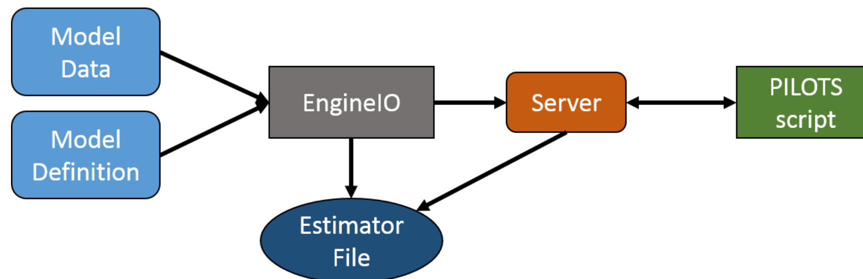
Simulation mode, as opposed to real-time mode, relies on a different algorithm that uses simulated time. When running a PILOTS application in simulation mode, the -Dtimespan=tb~te (where tb and te are both java date/time format strings) argument must be specified.

Learning Model Overview

This page details instructions on how to execute a given PILOTS example with machine learning and serves as a guide for constructing and implementing new learning models.

Running a PILOTS program entails updating the machine learning server, generating new estimator files with new learning model definitions (with corresponding data), and using a PILOTS script to interact with the server to predict results using predefined models.

Note: The following commands shown for the command line are assumed to be implemented in the bash shell



Overview of Interaction with Machine Learning Component

The steps for interacting with the machine learning component of PILOTS are as follows:

1. Installation of necessary packages
2. Build a PILOTS jar file (from home directory)
3. Configure aliases (from home directory)
4. Train any new necessary learning models (offline)
5. *(If using simulation)* Add Data for Producers
6. Run the learning model server
7. Compile and run the plt program with Producer and Output Handler
8. *(If using simulation)* Output producer data to Output Handler

Installation of Necessary Packages

Make sure to install the following before working with the learning model component of PILOTS:

1. python-pip
2. virtualenv

Then simply run at the root directory of the project:

```
virtualenv .
source bin/activate
```

Finally run the following in the *pilots/util/learningmodel/* directory:

```
pip install -r requirements.txt
```

Build a PILOTS jar file

In `$PILOTS_HOME`, run the build script

```
$ ./build.sh // (for Windows, use build.bat instead)
```

After doing so, pilots.jar will be created under the `$PILOTS_HOME/lib` directory.

Configure aliases

In order to make sure that aliases for the `plc` and `plcsim` compiler commands are correctly figured as well as the definition of `$PILOTS_HOME`, make sure that you are in the *root directory* of the project and then use the following command:

```
source setenv
```

Training a Learning Model (Offline)

Before implementing a new learning model in PILOTS, the model has to be trained offline. In order to train a new model for the machine learning component in the PILOTS project, follow the specifications for the JSON definition file that are detailed in the README in the learningmodel directory. *The details of this README document are shown in the* ***Predictive Server*** *section in the Appendix.*

Also, follow the specifications for the corresponding JSON schema file that are detailed in the README in the learningmodel directory, and add the corresponding data file in the same directory as the schema file. Within this data, make sure that the input data is separated by adjacent columns and that the first row acts as the header.

In `server.json`, add the loading path of the ".estimator" file with the name to be exposed to the PILOTS program.
For example, look at the linear regression model component of the server.json file:

```
{
    "id":"linear_regression",
    "model":"linearRegression.estimator",
    "translation": {"aoa": "a"},
```

```
    "update": false
},
```

Finally, make sure that the id used is mapped to a given **sklearn** learning model in `pilots/util/learningmodel/engine/scbase/algorithms.py` (for example, look at line 6):

```
1  from sklearn.linear_model import LinearRegression
2  from sklearn.svm import SVR
3  from algo.bayes import Bayes
4  from algo.custom import CruiseAlgorithm
5  # all models are registered in this file as attribute
6  linear_regression = LinearRegression
7  linear_regression_twice = LinearRegression
8  svr = SVR
9  bayesonline = Bayes
10 bayes_prediction_test = Bayes
11 cruise_algorithm = CruiseAlgorithm
```

Once this is done, execute the **engineIO** from the *pilots/util/learningmodel/engine/* directory with the new JSON definition file as the command line argument (this will train the model that you have defined based on your definition file, schema, and data):

```
python engineIO.py <training definition file>
```

Once the EngineIO is executed, the input data for the learning model with be read, and the learning model with be trained offline. This training will result in a generated estimator file in the learningmodel directory and a message will display saying, "engineIO: Finished Training" if there are no errors in the offline training process.

After running the **engineIO**, the `server.json` file should be updated with the new model and a new `<learning_model>.estimator` should be created which corresponds to the given JSON definition file.

Add Data for Producers

Make sure that the data that you intend to use for your producers in a simulation are stored in the data directory.

Follow the format similar to a.txt from the PredictionTest linear regression example.

Note: Notice that the first line of the file must be a header beginning with a pound sign (#). If you are expecting an output for the input, based on the learning model, then make sure to also include the label of the output similar to that defined in the model training data.

Running Learning Model Server

A socket connection must be made with the learning model server client in order to interact with the PILOTS machine learning component. Therefore, before running the **plt** program, run the learning model server by executing the `server.sh` script in *pilots/util/learningmodel/*

```
./server.sh
```

When the server is run, any new ".estimator" files will be loaded into the predictive server (loading the models defined in server.json)

(note: you may need to wait a few seconds for the server to start up)

Running the plt program with Producer and Output Handler

Once the previous procedures are completed, follow these 3 main steps:

1. Run the desired output handler script for the corresponding PILOTS program in a new terminal.
2. In a separate terminal, follow the instructions in the Compiling and running a program section of the *PILOTS: Running Tutorial* page. This involves the PILOTS compiler, javac, and execution.
    - If you are running a simulation in which all the pre-existing producer data is sent at once, compile the *plt* file using the command ***plcsim***
    - If you are running a real-time program in which the producer data is continuously streamed, compile the *plt* file using the command ***plc***
3. In a separate terminal, run the necessary **producer** script that will provide the input data.

(If using simulation) Output Producer Data to Output Handler

If a simulation is being run, go back to the terminal running the class generated from the plt file, and press **enter**. (After this, some warning messages may appear in this terminal, but they do not hinder the simulation.)

Examples

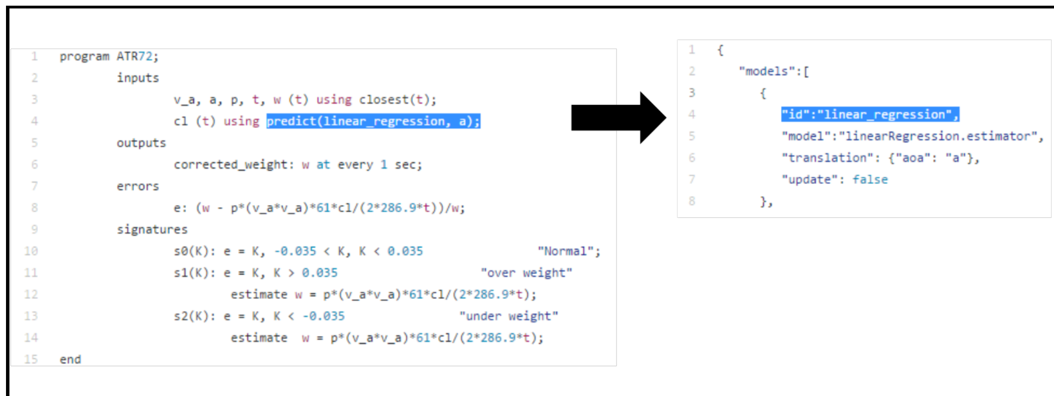
For some simple examples of the execution of a PILOTS program with the machine learning component (offline), refer to the:

- PredictionTest Linear Regression example
- PredictionTest Dynamic Bayes Classifier example

Additional Notes

- The learning model (first command line argument) used in the **predict** function in a **plt** file must refer to the specific corresponding id in pilots/util/learningmodel/engine/server.json

(In the following example, the id **linear_regression** was used in both the ATR72.plt and the server.json file)



- There is no compilation step for the output handler and for the producer because it is assumed that the corresponding java files are already compiled in a separate directory (i.e. `pilots/util/ChartServer.java` and `pilots/util/FileInputProducer.java`)

Simple Linear Regression Example

In this example, a new linear regression learning model will be trained to recognize the relationship $b = 2a$. Then, this learning model will be implemented and the learned relationship will be compared to another output, generated from `c.txt`, which is similar to `b.txt` from the Twice (simulated mode) example found from the PILOTS tutorial examples page. As such, this example will produce an output similar to the Twice (simulated mode) example.

Complete the following steps:

Build a PILOTS jar file

In `$PILOTS_HOME`, run the build script:

```
$ ./build.sh // (for Windows, use build.bat instead)
```

After doing so, pilots.jar will be created under the `$PILOTS_HOME/lib` directory.

Configure aliases

In order to make sure that aliases for the `plc` and `plcsim` compiler commands are correctly figured as well as the definition of `$PILOTS_HOME`, make sure that you are in the *root directory* of the project and then use the following command:

```
source setenv
```

Add definition for linear regression model:

- Make sure the following **definition file** is added to *pilots/util/learningmodel/engine/definitions*
  - `regression_twice.json`
- Make sure the following **schema file** is added to *pilots/util/learningmodel/engine/data*
  - `twice_schema.json`
- Make sure the following **data file** is added to *pilots/util/learningmodel/engine/data*
  - `twice_training.csv`
    - *(Notice that this data file maintains an approximate linear pattern of* $\boldsymbol{b = 2a}$ *with some variation.)*

- Make sure `server.json` in *pilots/util/learningmodel/engine* is edited to include the new learning model by adding these lines:

```
{
    "id":"linear_regression_twice",
    "model":"lin_reg_twice.estimator",
    "translation": {},
    "update": false
},
```

Make sure the learning model id is correctly mapped to **sklearn** learning model in `pilots/util/learningmodel/engine/scbase/algorithms.py`

In **Terminal 1**, run the engineIO with the new learning model definition (while in *pilots/util/learningmodel/engine*):

```
python engineIO.py definitions/regression_twice.json
```

Run the learning model server:

In **Terminal 1**, run `server.sh` in *pilots/util/learningmodel/engine*

(note: you may need to wait a few seconds for the server to start up)

Add Data for Producers

Make sure that the following **data files** are added to *data/PredictionTest_twice*

- `a.txt`
- `c.txt`

Run the Output Handler:

In **Terminal 2**, make sure to call `source setenv` from the root directory, and then execute **outputHandler** in *examples/PredictionTest_twice*

Compile and Run the PLT file:

The plt file for this example is shown below:

```
program PredictionTest_twice;
inputs
    a, c (t) using closest (t);
    b (t) using predict(linear_regression_twice, a);
outputs
    o: c - b at every 1 sec;
end
```

In **Terminal 3**,make sure to call `source setenv` from the root directory, and then compile `PredictionTest_twice.plt` in *examples/PredictionTest_twice* using the command:

`plcsim PredictionTest_twice.plt`

In **Terminal 3**, compile the generated `PredictionTest_twice.java` in *examples/PredictionTest_twice*

`javac PredictionTest_twice.java`

In **Terminal 3**, execute the new PredictionTest_twice class file with the appropriate arguments. To simplify this process, execute the script **PredictionTest_twice** in *examples/PredictionTest_twice*

(Notice that the classpath is defined within this execution script.)

Run the Producer:

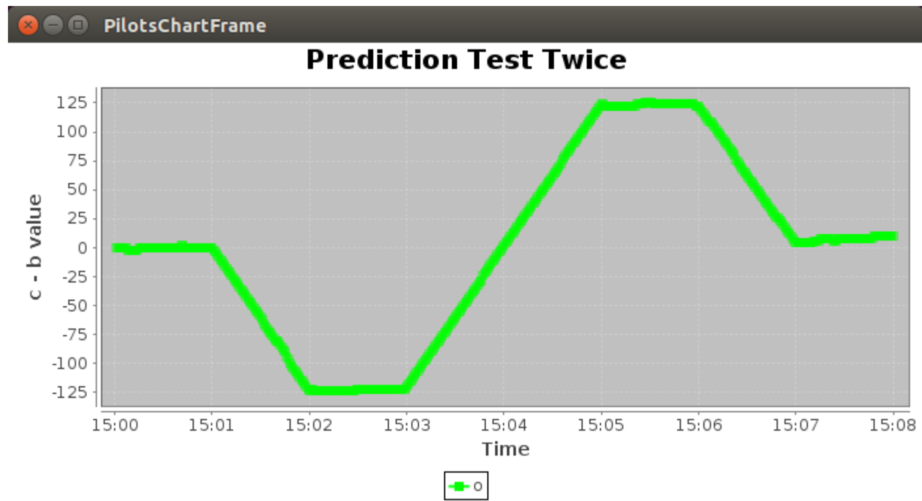
In **Terminal 4**, make sure to call `source setenv` from the root directory, and then (in order to produce the simulation) execute **PredictionTest_twiceProducer** in *examples/PredictionTest_twice*

Output simulated data to the Output Handler:

In **Terminal 3**, press **enter**. (After this, some warning messages may appear in this terminal, but they do not hinder the simulation.)

Final Output

If these steps are executed correctly, you should observe the following output in the **Output Handler** (terminal 2):



These results show the difference between the calculated output vs the simulated error data.

Notice that this result is very similar to the Twice (simulated mode) example. In the old example, **b** was the output of **a**, and **b** was *explicitly* defined as **twice** that of **a**. In the learning model example, **c** was the output of **a**, and **c** was *learned* to be about **twice** that of **a**.

Simple Bayes Classifier Example

In this example, a Bayes classifier learning model will be trained to 2 modes of relationships between b and a.

Mode	Relationship
0	b/a = 2
1	b/a = 3

After training the model, this dynamic learning model will be tested with some input data and will be able to recognize when the input data matches these 2 trained modes as well as 2 new modes (which will be automatically categorized as mode 2 and mode 3.

Complete the following steps:

Build a PILOTS jar file

In `$PILOTS_HOME`, run the build script:

```
$ ./build.sh // (for Windows, use build.bat instead)
```

After doing so, pilots.jar will be created under the `$PILOTS_HOME/lib` directory.

Configure aliases

In order to make sure that aliases for the `plc` and `plcsim` compiler commands are correctly figured as well as the definition of `$PILOTS_HOME`, make sure that you are in the *root directory* of the project and then use the following command:

```
source setenv
```

Add definition for dynamic Bayes classifier model:

- Make sure the following **definition file** is added to *pilots/util/learningmodel/engine/definitions*
  - `bayes_prediction_test.json`
- Make sure the following **schema file** is added to *pilots/util/learningmodel/engine/data*
  - `prediction_test_mode_schema.json`

- Make sure the following **data file** is added to *pilots/util/learningmodel/engine/data*
  - `prediction_test_mode.csv`
    - *(Notice that this data file classifies mode 0 when b/a=2 and mode 1 when b/a=3.)*
- Make sure `server.json` in *pilots/util/learningmodel/engine* is edited to include the new learning model by adding these lines:

```
{
    "id":"bayes_prediction_test",
    "model":"bayes_prediction_test.estimator",
    "translation": {},
    "update": true
},
```

- Make sure the learning model id is correctly mapped to **sklearn** learning model in `pilots/util/learningmodel/engine/scbase/algorithms.py`
- In **Terminal 1**, run the engineIO with the new learning model definition (while in *pilots/util/learningmodel/engine*):

```
python engineIO.py definitions/bayes_prediction_test.json
```

Run the learning model server:

In **Terminal 1**, run `server.sh` in *pilots/util/learningmodel/engine*

(note: you may need to wait a few seconds for the server to start up)

Add Data for Producers

Make sure that the following **data file** is added to *data/PredictionTest_mode*

- `data.txt`

Run the Output Handler:

In **Terminal 2**, make sure to call `source setenv` from the root directory, and then execute **outputHandler** in *examples/PredictionTest*

Compile and Run the PLT file:

The plt file for this example is shown below:

```
program PredictionTest_mode;
inputs
    a, b (t) using closest (t);
    mode (t) using predict(bayes_prediction_test, a, b);
outputs
    Mode: mode at every 1 sec;
end
```

In **Terminal 3**, make sure to call `source setenv` from the root directory, and then compile `PredictionTest_mode.plt` in *examples/PredictionTest_mode* using the command:

`plcsim PredictionTest_mode.plt`

In **Terminal 3**, compile the generated `PredictionTest_mode.java` in *examples/PredictionTest_mode*

`javac PredictionTest_mode.java`

In **Terminal 3**, execute the new PredictionTest_mode class file with the appropriate arguments. To simplify this process, execute the script **PredictionTest_mode** in *examples/PredictionTest_mode*

(Notice that the classpath is defined within this execution script.)

Run the Producer:

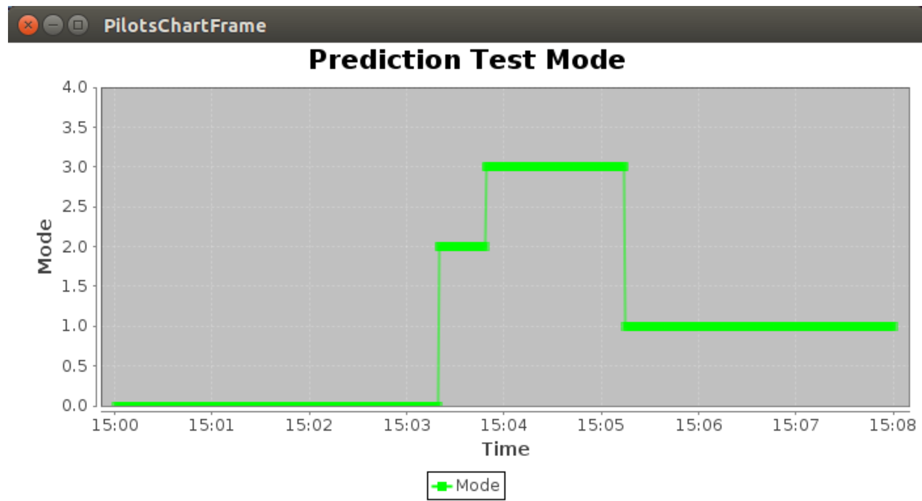
In **Terminal 4**, make sure to call `source setenv` from the root directory, and then (in order to produce the simulation) execute **PredictionTest_modeProducer** in *examples/PredictionTest_mode*

Output simulated data to the Output Handler:

In **Terminal 3**, press **enter**. (After this, some warning messages may appear in this terminal, but they do not hinder the simulation.)

Final Output

If these steps are executed correctly, you should observe the following output in the **Output Handler** (terminal 2):



Notice in this output that modes 0 and 1 were correctly identified in the appropriate moments in the input data (from the producer). Additionally, 2 new modes were dynamically discovered when (b/a) fell into significantly new patterns (which were b/a=4 and b/a=0.5).

PILOTS related publications

---

[1] Sida Chen, Shigeru Imai, Wennan Zhu, and Carlos A. Varela. Towards learning spatio-temporal data stream relationships for failure detection in avionics. In *Dynamic Data-Driven Application Systems (DDDAS 2016)*, Hartford, CT, Aug 2016. To appear.

[2] Shigeru Imai, Erik Blasch, Alessandro Galli, Wennan Zhu, Frederick Lee, and Carlos A. Varela. Airplane flight safety using error-tolerant data stream processing. *IEEE Aerospace Electronic Systems Magazine (IEEE-AESM)*, 2016. To appear.

[3] Shigeru Imai, Alessandro Galli, and Carlos A. Varela. Dynamic data-driven avionics systems: Inferring failure modes from data streams. In *Dynamic Data-Driven Application Systems (DDDAS 2015)*, Reykjavik, Iceland, June 2015.

[4] Shigeru Imai, Richard Klockowski, and Carlos A. Varela. Self-healing spatio-temporal data streams using error signatures. In *2nd International Conference on Big Data Science and Engineering (BDSE 2013)*, Sydney, Australia, December 2013.

[5] Shigeru Imai and Carlos A. Varela. A programming model for spatio-temporal data streaming applications. In *Dynamic Data-Driven Application Systems (DDDAS 2012)*, pages 1139–1148, Omaha, Nebraska, June 2012.

[6] Shigeru Imai and Carlos A. Varela. Programming spatio-temporal data streaming applications with high-level specifications. In *3rd ACM SIGSPATIAL International Workshop on Querying and Mining Uncertain Spatio-Temporal Data (QUeST) 2012*, Redondo Beach, California, USA, November 2012.

[7] Richard S. Klockowski, Shigeru Imai, Colin Rice, and Carlos A. Varela. Autonomous data error detection and recovery in streaming applications. In *Proceedings of the International Conference on Computational Science (ICCS 2013). Dynamic Data-Driven Application Systems (DDDAS 2013) Workshop*, pages 2036–2045, May 2013.

[8] Stacy Patterson and Carlos A. Varela. Steering complex systems using a dynamic, data-driven modeling approach. Streaming Technology Requirements, Application and Middleware (STREAM2016), March 2016.

[9] Carlos A Varela. Dynamic data driven avionics systems. Streaming Technology Requirements, Application and Middleware (STREAM2015), October 2015.

Appendices

Predictive Server

Goal of this project: Define machine learning algorithms, training/testing processes and build a server to handle communication about data streaming and machine learning.

Service:

1. Prediction service: Input: some homogeneous datastreams; Output: some predicted values using requested model
2. Training service: Input: training definition file; output: trained estimator
3. Specified for PILOTS project: produce Java Client for streaming data communication between PILOTS main program (Data Selection) and Predictive Server.

Installation (Linux/MacOS)

prerequisite

Make sure to install the following before working with the learning model component of PILOTS:

1. python-pip
2. virtualenv

Then simply run at the root directory of the project:

```
virtualenv .
source bin/activate
```

Finally run the following in the *pilots/util/learningmodel/* directory:

```
pip install -r requirements.txt
```

Quick Start

train a model

```
python engineIO.py <training definition file>
```

run the server

At `engine` directory, run

```
./server.sh
```

Definition Files

Schema File (JSON)

Schema files contain schematic meaning of each column (attribute). The following describes mandatory or optional fields in JSON file. Every field's value is *list of string* and every string contained in list corresponding to the column in the file with the same index in the list.

1. `name` (**mandatory**) variable names, which will be used in learning model.
2. `unit` (**optional**) the units, required if `unit_transformation` appears in the `preprocessing` field.

Training Definition File (JSON)

Training definition files contain learning model training parameters including the following:

1. `data` (**mandatory**,*dictionary*) The definition for what data and schema file will be loaded as the training set, and what constants are involved in evaluation (if necessary).
    - `file` (**mandatory**, *list of string*) A list of string containing training set (notice: they should be in the same schema and type)
    - `type` (**mandatory**, *string*) The type of files described in `file`.
    - `header_type` (**mandatory**, *string*) The header type of files described in `file`.
    - `schema` (**mandatory**, *string*) The path to schema of the files, see Section Schema File (JSON).
    - `constants` (**mandatory**, *dictionary*) The constants used in modeling. Each element in the dictionary contains a string as key and a number as value.
2. `preprocessing`(**mandatory**,*dictionary*) The definition for preprocessing phase of model training. [In development]
    - `unit_transformation` (**optional**,*dictionary*) a built in method for unit transformation as an example of preprocessing.
3. `model`(**mandatory**,*dictionary*) The definition for learning algorithm and feature/labels transformation. [In development]
    - `features` (**mandatory**, *list of string*) The features' values of the model. The variables including constants defined in `constants` should be embraced with '{' and '}'.

- `labels` (**mandatory**, *list of string*) The target value (label) of the model. The variables including constants defined in `constants` should be embraced with '{' and '}'.
- `algorithm` (**mandatory**, *list of string*) The definition of learning algorithm and its parameters.
  - `id` (**mandatory**, *string*) The ID of algorithm defined in algorithm registration file.
  - `param` (**mandatory**, *dictionary*) The parameters used in learning algorithm.
  - `save_file` (**mandatory**, *string*) The file path for trained model to be saved.
  - `serialize_function` (**optional**, *string*) The function called to save the trained model, the default function is pickle.dump
  - `deserialize_function` (**optional**, *string*) The function called to load the trained model, the default function is pickle.load

Server Definition (JSON)

1. `models`(**mandatory**, *list of dictionary*) The definition of models that could be accessed through socket. Each element in the list is a model with the following attributes:
   - `id`(**mandatory**, *string*) Distinct ID for client to discriminate between different models.
   - `model`(**mandatory**, *string*) The path to an estimator file generated by offline training component (EngineIO).
   - `translation` (**mandatory**, *dictionary*) The translation between client defined variable names and model variable names. [In Development]
   - `update` (**mandatory**, *boolean*) A switch of whether server stores the change of estimator to file when server shuts down.

Server

Request

The server is designed to accept URL request with parameters

`http://hostname:port/?model=x&name=y&value=z`

where x is a string id of model id defined in server definition; y is a list of strings (variable names) seperated by comma (,). z is a list of numbers (variable value) seperated by comma (,). Each z's schematic meaning corresponds to the string in y with the same index.

Response

The response is composed in a json response with the root key `value`, and the value nxd matrix, where each row represents a single data point.

Example

URL Request

```
http://127.0.0.1:5000/?model=linear&name=aoa&value=1.0
```

JSON Response

```
{"value": [[6.7476931583308]]}
```

Schema File

```
{
    "names": ["v","p","t","w","a","cruise"],
    "units": ["knot","in_Hg","celsius","force_pound","degree",""]
}
```

Training Definition File

```
{
    "data":{
        "file": ["data/training.csv"],
        "type": "csv",
        "header_type": "csvheader",
        "schema": "data/schema.json",
        "constants": {"s": 61.0, "R": 286.9}
    },
    "preprocessing":{
        "unit_transformation": {"v":"m/s", "p":"pascal","t":"kelvin","w":"newton","a":"radia
    },
    "model":{
        "features": ["{a}"],
        "labels": ["2*{w}/({v}**2*({p}/{R}/{t})*{s})"],
        "algorithm":{
        "id": "linear_regression",
        "param": {},
        "save_file": "regression.estimator"
        }
    }
}
```

Server File


```
{
   "models":[
     {
         "id":"linear",
         "model":"regression.estimator",
         "translation": {"aoa": "a"},
         "update": false
      },
      {
         "id":"bayes",
         "model":"bayes_online.estimator",
         "translation": {"v_a": "v"},
         "update": true
      }
   ]
}
```