Augustus: Modeling Primer

Release 0.5.0.0

Open Data Group

December 19, 2011

Contents

Before proceeding, the reader should have Augustus installed on an accessible computer. No knowledge of Python or of PMML (Predictive Model Markup Language) is assumed at this point. For the advanced user, `README` files under the `augustus-examples` directory of the Augustus release may be more appropriate.

1 Introduction

This section briefly introduces Augustus. The reader should afterward be able to identify when Augustus would be useful and should be able to create a simple project from scratch.

Description

Augustus is an open source software toolkit for building and scoring statistical models. It is written in Python and its most distinctive features are:

- Ability to be used on sets of *big data*; these are data sets that exceed either memory capacity or disk capacity, so that existing solutions like **R** or **SAS** cannot be used. Augustus is also perfectly capable of handling problems that can fit on one computer.
- *PMML* compliance and the ability to both:
  - produce models with PMML-compliant formats (saved with extension `.pmml`).
  - consume models from files with the PMML format.

Augustus has been tested and deployed on serveral operating systems. It is intended for developers who work in the financial or insurance industry, information technology, or in the science and research communities.

Usage

Augustus produces and consumes Baseline, Cluster, Tree, and Ruleset models. Currently, it uses an event-based approach to building Tree, Cluster and Ruleset models that is non-standard. Standard model producers are schedule for relase in January of 2011.

Augustus produces and consumes segmented models and can continue training models on the same input data it is scoring, so that recent events can influence how future events are scored. This is a new feature introduced with this (Augustus 0.5) release. A typical model development and use cycle with Augustus is as follows:

1. Identify suitable data with which to construct a new model.
2. Provide a model schema which proscribes the requirements for the model.
3. Run the Augustus producer to obtain a new model using training data.
4. Run the Augustus consumer on new data to effect scoring.

This tutorial walks through two examples that each highlight different Augustus features. Each of the examples is included in the Augustus release, under the `augustus-examples` directory.

2 Walkthrough: gaslog producer

The `augustus-examples/aim/gaslog` directory demonstrates how to use Augustus as both a PMML model producer and consumer. This small problem demonstrates using a baseline model to calculate the z-score of new observations relative to a known distribution, and then demonstrates the use of variants on the baseline model outputs and configurations.

Steps

1. Identify data.
2. Organize the project.
3. Preprocess the data, if necessary.
4. Create a PMML file that describes the model.
5. Run Augustus.
6. Post-process the results.

Identify suitable data

The data are from an Augustus contribor, Jim, who kept a log of every time he filled his car with gas since midway through 2003. The data file, `augustus-examples/gaslog/gaslog.xml`, is XML-formatted; Augustus can other file types such as CSV.

His gas log contains the date, gallons, car mileage, miles traveled since the previous fill-up, the price he paid in dollars, and whether the car was an old one or a new one. There are missing data fields sometimes; Augustus handles this by reporting a score of *Missing* if a variable is missing or *Invalid* if a variable is derived from a missing value or the derivation involves a division by zero or a result that cannot be cast to the derived data type.

In this example, the goal is to show how two common change-detection models are expressed in PMML, and how to run Augustus. Additional advanced examples demonstrate other tools Augustus provides to create and modify models.

Create a project directory (organize your project)

No particular project directory structure is required by Augustus; all input and output file names and paths are specified in the XML configuration files. The directory for this project is in `augustus-examples/gaslog/`, which has the following subdirectories:

```
gaslog/
  data/
  advanced/
  introductory/
```

Data files are stored in `gaslog/data`, and are shared between two projects. This primer describes the contents of the `introductory` directory; the `advanced` directory contains a slightly more advanced use case of Augustus, with instructions in a `README` file. Makefiles for each step are provided for the advanced example.

A typical project file structure would separate the data from the consumer model schema (for example, `example_model.pmml`) and configuration file (for example, `consumer_config.xcfg`). This makes it easier to manage multiple data files, configuration files, and model schema during development. Under `introductory` are the folders:

```
introductory/
  config/
  models/
  results/
```

the configuration files in `gaslog/config` are in XML format, and communicate the location of the data, the PMML model, and other details about how to run to Augustus. The `models` directory contains the statistical models, described using the *PMML* format. The `results` directroy will contain output scores. Additional directories may be valuable for some projects, in which there could be preprocessing or post processing scripts. (This release of Augustus provides new tools for common post processing steps.)

Pre-processing (Prepare the data)

In this example, the data were manually entered into an XML format in `gaslog/data/gaslog.xml`, and need no further processing.

In general, the input data have to be readable by Augustus. Augustus's PMML consumer can analyze data in an *XML* format (possibly output from a database), from a CSV file, or in its own *UniTable* format (with file extension `.nab`).

The UniTable is a data structure analogous to a data frame in the *R* statistical programming system: briefly, a data structure containing an array with columns that can have different data types, and whose columns are also accessible by a string label.

UniTable also has methods to read directly from a CSV or otherwise delimited file, and can deduce the delimiter. Below are the first and last few lines of `gaslog/data/gaslog.xml`. The entries in each row are accessible by their tag names, for example `date` or `mileage`. The tag names *table* and *row* can be any XML-compliant string the user chooses:

```
<table>
  <row>
    <date>2003/06/02</date> <gallons>14.7</gallons> <car>old</car>
  </row>

  ...

  <row>
    <date>2011/09/22</date>
    <gallons>10.043</gallons>
    <mileage>60882</mileage>
    <miles>334.0</miles>
    <price>38.86</price>
    <car>new</car>
  </row>
</table>
```

Provide a model schema

The model can be built once every field in the data set has been named (e.g. `date` and `mileage`) and the data types of each field are known (`date` is an ISO 8601 formatted date string; the same date format used in HTML; and `mileage` is an integer).

The model is written in *PMML*, a XML-based language that describes predictive models. First, it defines the format of the data, and then (optionally) it defines the model that will be applied to the data.

Augustus can calculate the properties of a distribution when given a set of training data, and can automate the model segmentation, it just has to be told where the training data are, where the model skeleton is, and where to put the trained model. An example PMML skeleton is in `<Install-dir>/augustus-examples/gaslog/introductory/models/model_template.pmml`.

As a brief interlude: XML elements in a PMML file have to be present in a specific order. Their placement, and whether or not they are required, is determined by the standards body (the Data Mining Group), and communicated in PMML's XSD schema, available at: http://dmg.org/v4-1/GeneralStructure.html. A user who wants access to a richer set of controls than are presented here will have to learn to read through pieces of the DMG's schema in order to find out where to put parts of the model. ODG will also soon offer a graphical tool to help with model creation–check back on our web site for this.

The best way to start describing a statistical model in PMML format is to copy and paste from existing files and change, delete, or add things to meet the current model's needs. The file `example_model.pmml` is commented to identify some of the commonly available options for a Baseline Model. It is also included in the *Example section*. The sequence of elements in a PMML file is fixed; items must appear in the order specified. The smallest possible complete PMML file for this example looks like:

```
<PMML version="4.1">
    <Header />
    <DataDictionary>
        <DataField name="miles" dataType="double" optype="continuous" />
    </DataDictionary>
</PMML>
```

In the above, all optional elements, including the specification of a statistical model, are omitted. At least one data field is required to be defined in the data dictionary. The field name should be the same name as the column name in a UniTable, or the element name in XML-formatted information like our example of `gallons` and `miles`. The attributes for the data field depend on the nature of the data; `optype` can be `categorical`, `ordinal` or `continuous`.

After the data dictionary comes an optional transformation dictionary. The transformation dictionary defines functions that can be applied to the data, and any derived fields that could be used in the subsequent model. Finally comes the optional model specification. With these elements, the document would look something like the below:

```
<PMML version="4.1">
    <Header copyright="Open Data Group, 2011" />
    <DataDictionary>
        <DataField name="date" optype="continuous" dataType="date" />
        <DataField name="gallons" optype="continuous" dataType="double" />
          .
          . (Insert additional data fields here...)
          .
    </DataDictionary>
    <TransformationDictionary>
          .
          . (With optional DefineFunction elements followed
          .  by optional DerivedField elements.)
          .
    </TransformationDictionary>
    <MiningModel functionName="regression">
          .
          . (Or another type of model; Augustus has algorithms
          .  to support MiningModel, BaselineModel, ClusteringModel, etc...
          .  The contents of the MiningModel are described
          .  separately below.)
          .
```

```
    </MiningModel>
</PMML>
```

Since options are listed in the comments in `example_model.pmml`, they are not described in this tutorial. The mining model has an option for segmentation, that makes it possible to score only a subset of the data. In this example, the segment template will never be calculated because its predicate is `False`; it will evaluate to False when compared against any data. The other segments are not yet generated, but there will be 24 segments created; one for each of Jim's two new cars multiplied by the twelve months of the year.

The smallest `MiningModel` element in a PMML file would look something like this:

```
<MiningModel functionName="regression">
    <MiningSchema>
        <MiningField name="date" />
    </MiningSchema>
</MiningModel>
```

The mining schema would list every field to be used in the statistical model; the `name` attribute must be the name of one of the `DataField` elements in the data dictionary, or the name of a `DerivedField` element in the transformation dictionary. Not all defined data fields or derived fields need to be used. Again, options are not described in this tutorial but are in the comments of the *Example* section. With segments, the mining model would look more like:

```
<MiningModel functionName="regression">
    <MiningSchema>
        <MiningField name="date" />
        .
        . (Additional items omitted for brevity)
        .
    </MiningSchema>
    <Segmentation multipleModelMethod="selectAll">
        <Segment>
            <SimplePredicate field="car" operator="equal" value="old" />
            <BaselineModel functionName="baseline">
            .
            . (A simple or compound predicate determines which
            .  data belong to which segment.  Then the model
            .  is inserted.  The details of the model are only
            .  in the actual example file, for brevity.)
            .
            </BaselineModel>
        </Segment>
        .
        . (As many segments as needed...)
        .
    </Segmentation>
</MiningModel>
```

To make complex models, it will eventually be necessary to read the PMML schema and find the elements and attributes used to describe parts of that model. All existing schema are available from the DMG web site: http://www.dmg.org/ [1].

[1] Open Data Group is a member of the Data Mining Group; the group that manages the PMML standard.

Run Augustus as a model producer

Running Augustus as a *model producer* means either training an existing PMML model file or creating segments to populate a PMML model given a skeleton file.

The first reason to use Augustus' producer tools is to identify baseline statistics for a model from existing data during a period of normal operation. The second reason is to automatically create PMML segments in a model. Augustus typically handles thousands or tens of thousands of segments; the actual upper bound depends on computer memory. Even though a person could manually edit a model with 24 segments, even small numbers of segments make editing at best tedious and at worst error-prone.

One reason to create segmented models is to capture consistent behavior that is different on different days, for example the amount of traffic on the roads on weekends versus weekdays, or during rush hour versus during the early morning. Then, all of the traffic on these different days and times can be grouped together, and the baseline statistics related to each conceptual group can be calculated separately to provide a richer insight into the overall system behavior and more accurate inferences about when observed behavior does not match the expected behavior.

Create a PMML skeleton file

The skeleton file is a PMML document containing:

- A complete header section.
- A complete data dictionary.
- A complete transformation dictionary (if derived fields are used).
- **The shell of a model, which**
  - *Must* contain the MiningSchema section that lists fields used in the segments or for calculation. (Or it can contain another model type, but then it would not produce any segments, just train the existing model with new data.)
  - *Can* contain an Output section if there are OutputFields that are common to all segments that the user wants to include with the calculated scores.
  - *Can* contain a LocalTransformations section that describes the common to all segments that the user wants to include with the calculated scores.
  - *Must* contain the Segmentation section that will eventually contain segments, and may contain some segments already.
  - *Must* contain a special Segment, with `id="ODG-SegmentTemplate"` and predicate `<False />` as the selection predicate. This template will be used to auto-generate all of the segments requested in the Augustus configuration file. It should contain a complete model exactly the way you want it to be, except with zeros or nonsense data for the description of the distribution; this will be replaced when Augustus trains the model.

The model template that will be described in this primer is

`<Install-dir>/augustus-examples/gaslog/introductory/models/model_template.pmml`. The LocalTransformations element is described here to introduce how to define DerivedFields and how to apply functions. There are three transformations: the first converts the `date` from an ISO date string, which is the date format used for date strings in PMML, to the year, and names the new variable `year`. The second converts `date` to the month, and the third estimates the miles per gallon by dividing the miles between fill-ups by the gallons in the current fill-up. The PMML excerpt that defines the new field `mpg` is:

```
<LocalTransformations>
    <DerivedField name="mpg"
```

```
                optype="continuous"
                dataType="double">
            <Apply function="/">
                <FieldRef field="miles" />
                <FieldRef field="gallons" />
            </Apply>
        </DerivedField>
        .
        . (With additional DerivedField elements below...)
        .
</LocalTransformations>
```

in which each derived field is named and identified. The '<Apply function="a_function"> ... </Apply>' tags describe a function call in PMML. Arguments to the function are passed in the order they are written, so the above PMML could be translated to something like:

```
mpg = miles / gallons
```

Some predefined functions are +, −, *, log10, ln, sqrt, abs, exp, round, uppercase. The model template also contains the function formatDateTime, which is very useful for formatting temporal data. It uses conversion strings that are exactly the same as that used by C's strftime on (on UNIX-like systems, type man strftime at a command prompt for a description). These and a few dozen others are described on the DMG web site: http://www.dmg.org/v4-1/BuiltinFunctions.html

Derived fields can be used in exactly the same way as defined fields; they can be referred to in output, for scoring, and can be used in any derived field defined below its definition.

For the type of model described here, the most important part of the PMML file is the definition of the Segment Template. The template in model_template.pmml is excerpted here:

```
<Segment id="ODG-SegmentTemplate" >
    <False/>

    <BaselineModel functionName="regression">
        <MiningSchema>
            <MiningField name="mpg" />
            <MiningField name="score" usageType="predicted" />
        </MiningSchema>

        <Output>
            <OutputField name="score" feature="predictedValue" />
        </Output>

        <TestDistributions field="mpg" testStatistic="zValue">
            <Baseline>
                <GaussianDistribution mean="0." variance="1." />
            </Baseline>
        </TestDistributions>
    </BaselineModel>
</Segment>
```

It will be copied exactly and inserted into the output PMML file for every new segment observed, but with the GaussianDistribution's properties changed to have a mean and a variance that matches the mean and variance in the training data. Also, the segment id will be an automatically generated number, and most important, predicate <False /> will be replaced by a predicate that selects data for a specific segment. Example predicates are:

```
<SimplePredicate field="year" operator="equal" value="2005" />
```

or:

```
<SimplePredicate field="year" operator="notequal" value="2003" />
```

or:

```
<CompoundPredicate booleanOperator="and">
   <SimplePredicate field="year" operator="equal" value="2005" />
   <SimplePredicate field"car" operator="equal" value="old" />
</CompoundPredicate>
```

During training, each time that a data point is processed which does not fall into any of the existing segments, Augustus will check to see whether any of the user's requested Segment predicate specifications match, and if so will create an appropriate segment.

Augustus's configuration file for Producing

The Augustus Producer configuration file specifies, in the following order:

1. Whether to log errors and information, the format of the logging string, and where to put logging information messages.
2. The location of the PMML model skeleton.
3. The location of the input data.
4. All information related to creating the model:
   (a) The location of the output model file
   (b) The minimum acceptable number of data points to define a data set before model will return a valid score. [Default is one.]
   (c) How segments are defined.

The best way to understand the options available for using Augustus as a PMML file producer is to modify the configuration file and run it multiple times. A commented version of `producer_config.xcfg` is available in the *Example section*.

Define the segments

Part of building the model is choosing how to segment the data for scoring. The MiningModel in `model_template.pmml` lists data with six dimensions:

```
<MiningSchema>
  <MiningField name="date" />
  <MiningField name="gallons" />
  <MiningField name="mileage" />
  <MiningField name="miles" />
  <MiningField name="price" />
  <MiningField name="car" />
</MiningSchema>
```

The miles per gallon calculated from the gallons and miles fields is the variable of interest; the purpose of all of the other dimensions is either for supplementary information to be used in the output, or to assign the miles per gallon to a specific segment.

Note: If a field is not identified in the MiningField of the MiningModel's MiningSchema or derived in LocalTransformations after that MiningSchema, then it cannot be used in Segmentation predicates, Output, or in calculating the score.

The producer configuration file `producer_config.xml` divides the data by car and month, meaning data related to the same car, that are collected in the same month of the year, belong to the same Segment of the data set, and will contribute to the same Baseline probability distribution. Both car and month are implemented as *enumerated dimensions*; every individual value in the category will belong to a different Segment in the PMML model.

Build the PMML

In this section, multiple input configurations will be run to clarify how different segmentation options work, and what kind of error messages occur. To run Augustus with the example configuration file, change directories into `<Install-dir>/augustus-examples/gaslog/introductory/config`. Then type:

```
$ Augustus producer_config.xml
```

Output should begin like this:

```
root     : INFO     Loading PMML model.
root     : INFO     Setting up data input.
root     : INFO     Setting up model updating/producing.
root     : INFO     Setting up Augustus's main engine.
root     : INFO     Calculating.
```

and should continue to completion, stating when it sees new segments:

```
META INFO   New segment created: (car EQ 'old') and (month EQ 'Jun'), ID=Untitled-1
root     : WARNING  Data not found for field: mileage
root     : WARNING  Data not found for field: miles
```

The information about new segments is communicated through the Metadata logger, and information about what Augustus is doing and general problems are communicated through the root logger (set up using Logging). The warning messages indicate that there are missing columns in some of the entries in the data set. The verbosity of the messages can be changed by setting the Logging and Metadata logging levels to higher values (like 'WARNING' or 'ERROR') or by removing these elements from the configuration file.

The output continues:

```
META INFO     New segment created: (car EQ 'new') and (month EQ 'Mar'), ID=Untitled-23
META INFO     New segment created: (car EQ 'new') and (month EQ 'Apr'), ID=Untitled-24
```

and then, a summary of the run is printed:

```
META INFO     ### Current MetaData content ###
META INFO     Run time : 0.322700977325
META INFO     Score calculation, total : 0.0364532470703
META INFO     Time Reading Data : 0.0998890399933
META INFO     Time searching for blacklisted items : 0.00146222114563
META INFO     Time to advance through data : 0.0380392074585
META INFO     Time to find and create new PMML segments : 0.0507431030273
META INFO     Time to load PMML model : 0.0140819549561
```

```
META INFO     Time to look up existing segments : 0.0162174701691
root     : INFO     Augustus is finished.
```

A full model should now be in

`<Install-dir>/augustus-examples/gaslog/introductory/models/produced_model.pmml`.

The following subsections demonstrate some additional configuration options. Since we will be modifying the producer configuration file, it would be a good idea to copy `producer_config.xcfg` to another file name, say `test_config.xcfg` so that the original file does not get lost.

Segmentation options

Augustus can handle thousands of segments, but even 24 make the PMML model file difficult to present in documentation. To reduce the size for a better discussion and to learn the format of the configuration file, open `test_config.xcfg` and modify the `EnumeratedDimension` element with `field="month"` to specifically list only January and February. The entire file should now look like the below:

```
<AugustusConfiguration>
    <Logging
        formatString="%(name)-9s: %(levelname)-8s %(message)s" level="INFO">
        <ToStandardError />
    </Logging>

    <Metadata
        formatString="META %(levelname)-8s %(message)s" level="INFO">
        <ToStandardError />
    </Metadata>

    <ModelInput>
        <FromFile name="../models/model_template.pmml" />
    </ModelInput>

    <DataInput>
        <ReadOnce />
        <FromFile name="../../data/gaslog.xml" />
    </DataInput>

    <ModelSetup outputFilename="../models/produced_model.pmml"
        mode="replaceExisting"
        updateEvery="event">

        <SegmentationSchema>
            <GenericSegment>
                <EnumeratedDimension field="car" />
                <EnumeratedDimension field="month">
                    <Selection value="Jan" />
                    <Selection value="Feb" />
                </EnumeratedDimension>
            </GenericSegment>
        </SegmentationSchema>
    </ModelSetup>

</AugustusConfiguration>
```

except that there are commented sections in the actual file that are omitted above. If you run the producer with the new configuration file:

```
$ Augustus test_config.xcfg
```

there will now be INFO messages peppering the rest of the logger output–to tell the user that a data row (an Event) did not match any of the segments, and would therefore be discarded (lines are wrapped for display purposes):

```
root     : INFO     Loading PMML model.
root     : INFO     Setting up data input.
root     : INFO     Setting up model updating/producing.
root     : INFO     Setting up Augustus's main engine.
root     : INFO     Calculating.
root     : WARNING  Data not found for field: mileage
root     : WARNING  Data not found for field: miles
root     : WARNING  Data not found for field: price
root     : INFO     Event 0 did not match any segment descriptions; discarding.
                      Data=car:old, date:2003-06-02, gallons:14.7...
.
. (intermediate output between events 0 and 185 is omitted...)
.
root     : INFO     Event 185 did not match any segment descriptions; discarding.
                      Data=car:new, date:2011-08-21, gallons:9.794...
root     : INFO     Event 186 did not match any segment descriptions; discarding.
                      Data=car:new, date:2011-09-22, gallons:10.043...
META INFO     ### Current MetaData content ###
META INFO     Run time : 0.267997980118
META INFO     Score calculation, total : 0.0104904174805
META INFO     Time Reading Data : 0.0434489250183
META INFO     Time searching for blacklisted items : 0.00136375427246
META INFO     Time to advance through data : 0.037558555603
META INFO     Time to find and create new PMML segments : 0.0111610889435
META INFO     Time to load PMML model : 0.0145878791809
META INFO     Time to look up existing segments : 0.0132262706757
root     : INFO     Augustus is finished.
```

if the INFO messages are too frequent, change the level in the <Logging /> element to WARNING or ERROR. With it set at ERROR there is no main logging output, only Metadata output, and the test run will look like:

```
META INFO     New segment created: (car EQ 'old') and (month EQ 'Jan'), ID=Untitled-1
META INFO     New segment created: (car EQ 'old') and (month EQ 'Feb'), ID=Untitled-2
META INFO     New segment created: (car EQ 'new') and (month EQ 'Jan'), ID=Untitled-3
META INFO     New segment created: (car EQ 'new') and (month EQ 'Feb'), ID=Untitled-4
META INFO     ### Current MetaData content ###
META INFO     Run time : 0.204999923706
META INFO     Score calculation, total : 0.0100080966949
META INFO     Time Reading Data : 0.0791020393372
META INFO     Time searching for blacklisted items : 0.00152134895325
META INFO     Time to advance through data : 0.0427370071411
META INFO     Time to find and create new PMML segments : 0.0120093822479
META INFO     Time to load PMML model : 0.0135910511017
META INFO     Time to look up existing segments : 0.0136258602142
```

The only segments created were for January and February, for the two different types of cars. We can confirm that only four segments exist by looking in the full PMML file: models/produced_model.pmml

Suppose we want to partition the months into groups rather than into categories. There is also an option for a *Partitioned Dimension* that can be commented out and used in place of the enumerated months. In the example, the months of the year (ranging from 1 to 12) are divided into five segments. This could, for example, capture that Jim logged more highway driving (and presumably better gas mileage) during months with traditional holidays, like June through

August and November and December. We will comment out the previous separately from months that usually contain work driving. We can modify `test_config.xcfg` further to define a Partitioned Dimension. Below is an excerpt of the Segmentation Schema only:

```
<SegmentationSchema>
    <GenericSegment>
        <EnumeratedDimension field="car" />
        <PartitionedDimension field="intmonth">
            <Partition low="0" high="5" />
            <Partition low="6" high="8" closure="closedClosed" />
            <Partition low="8" high="12" divisions="2" closure="openClosed" />
        </PartitionedDimension>
    </GenericSegment>
</SegmentationSchema>
```

The example shows that there can be more than one partition in a `PartitionedDimension` element, making it possible to segment the data using nonuniform ranges.

By default, the range for each bin is open on the low end and closed on the high end. The default number of divisions is 1. Allowed values for the `closure` attribute are `"closedOpen"` and `"openClosed"` when there are more than one division. If there is only one division, `closure` can also be `"closedClosed"` and `"openOpen"`. The ranges defined above are: (0, 5], [6, 8], (8, 10], and (10, 12]. The output will look like (lines are wrapped for display purposes):

```
Augustus test_config.xcfg

META INFO     New segment created: (car EQ 'old') and ((intmonth GE 6) and
                   (intmonth LE 8)), ID=Untitled-1
META INFO     New segment created: (car EQ 'old') and ((intmonth GT 8) and
                   (intmonth LE 10)), ID=Untitled-2
META INFO     New segment created: (car EQ 'old') and ((intmonth GT 10) and
                   (intmonth LE 12)), ID=Untitled-3
META INFO     New segment created: (car EQ 'old') and ((intmonth GT 0) and
                   (intmonth LE 5)), ID=Untitled-4
META INFO     New segment created: (car EQ 'new') and ((intmonth GT 0) and
                   (intmonth LE 5)), ID=Untitled-5
META INFO     New segment created: (car EQ 'new') and ((intmonth GE 6) and
                   (intmonth LE 8)), ID=Untitled-6
META INFO     New segment created: (car EQ 'new') and ((intmonth GT 8) and
                   (intmonth LE 10)), ID=Untitled-7
META INFO     New segment created: (car EQ 'new') and ((intmonth GT 10) and
                   (intmonth LE 12)), ID=Untitled-8
META INFO     ### Current MetaData content ###
META INFO     Run time : 0.247933149338
META INFO     Score calculation, total : 0.0319168567657
META INFO     Time Reading Data : 0.0750558376312
META INFO     Time searching for blacklisted items : 0.00143218040466
META INFO     Time to advance through data : 0.0404446125031
META INFO     Time to find and create new PMML segments : 0.0173301696777
META INFO     Time to load PMML model : 0.0142331123352
META INFO     Time to look up existing segments : 0.0179903507233
```

Blacklisting segments

If we check in the new model `gaslog/introductory/models/produced_model.pmml`, some of the baseline Gaussian Distributions for the miles per gallon look suspect. From within

gaslog/introductory/config/, type:

```
$ grep "mean" ../models/produced_model.pmml
```

the output should look like:

```
<GaussianDistribution variance="1.0" mean="0.0" />
<GaussianDistribution variance="1106.06928831" mean="41.9758043943">
<GaussianDistribution variance="58.870609039" mean="28.7080709112">
<GaussianDistribution variance="94.7170020643" mean="29.4167089563">
<GaussianDistribution variance="80.1025060911" mean="27.7009284687">
<GaussianDistribution variance="7015428.31403" mean="-554.894869158">
<GaussianDistribution variance="1302.66492634" mean="46.2428828612">
<GaussianDistribution variance="29692.5897065" mean="36.0739058162">
<GaussianDistribution variance="1020.96454195" mean="49.6721277817">
```

The first mean and variance, with `variance="1.0"` and `mean=0.0"` are from the segment template, and the rest are from the different month and car combinations. Some of them have a much bigger variance, and one has a negative mean. This is partly from typos during Jim's data entry and partly from missing receipts, and is a good example of the way real data look. Suppose, just for the sake of example, that a good option would be to ignore data points with negative total miles or in which the total miles between fill-ups are clearly too large to have been from one single gas tank.

One alternative is to use the `<BlacklistedSegments />` configuration option. Every record is compared with the entries in the `BlacklistedSegments` and if it matches, the record will be ignored. This means, if you are producing a model, the record will not contribute to the model's training. When the `<BlacklistedSegments />` option exists and Augustus is consuming a model, the score output will still be logged, but it will be empty, like this,

```
<Event id="10"></Event>
```

even if another segment in the model would have matched. To black list an item, first, make sure that the field with blacklisted contents exists in the LocalTransformations or MiningSchema section of the PMML Mining Model, or else Augustus will not be able to locate the value being used. In our case, the *mpg* field name is defined in the LocalTransformations, so lookup will not be a problem. Next, add a `BlacklistedSegments` section to the configuration file. It goes right before the `GenericSegment` section:

```
<ModelSetup outputFilename="../models/produced_model.pmml"
            mode="replaceExisting" updateEvery="event">
    <SegmentationSchema>

        <BlacklistedSegments>
            <PartitionedDimension field="mpg">
                <Partition high="5" closure="openClosed" />
                <Partition low="100" closure="closedOpen" />
            </PartitionedDimension>
        </BlacklistedSegments>

        <GenericSegment>
          .
          . (entries omitted for brevity)
          .
        </GenericSegment>
    </SegmentationSchema>
</ModelSetup>
```

The above will blacklist the following ranges (-infinity, 5] and [100, infinity); to mask the extreme entries. Run Augustus again with the modified inputs:

```
$ Augustus test_config.xcfg
```

and the new output will now have more reasonable means:

```
$ grep "mean" ../models/produced_model.pmml

 <GaussianDistribution variance="1.0" mean="0.0" />
 <GaussianDistribution variance="716.528207605" mean="48.1312898116">
 <GaussianDistribution variance="58.870609039" mean="28.7080709112">
 <GaussianDistribution variance="94.7170020643" mean="29.4167089563">
 <GaussianDistribution variance="80.1025060911" mean="27.7009284687">
 <GaussianDistribution variance="346.793653314" mean="45.184295448">
 <GaussianDistribution variance="13.2770564116" mean="32.7013033238">
 <GaussianDistribution variance="166.84629477" mean="38.4476267558">
 <GaussianDistribution variance="177.798362894" mean="41.0303215955">
```

Run the Augustus consumer

Statistically, it is not appropriate to run this model against the same data we used to train it, but we can still do it for the sake of the walk through. Augustus can be run from command-line mode if the user does not need any logging, segmentation, or model updating. To do this, type:

```
$ Augustus --model ../models/produced_model.pmml \
   --data ../../data/gaslog.xml  > ../results/output.xml
```

This output will not have an opening and closing tag, so if you use it as an input to other programs, remember to wrap it. There should be no output to the screen, but if you change directories to `../results/` you should see the `output.xml` file.

Post-process the results

In this part of the example, there is no post-processing; presumably the output will be sent to another program that will then send alert notifications or update a display... so congratulations, you have successfully run of Augustus as a Baseline Producer to create a PMML-formated segmented model from a set of training data.

3 Walkthrough: gaslog consumer

To present the consumer, a sample model is provide if you do not want to work through the steps required to produce a model above.

In this section, the The data set is the same as with producing, so the initial steps of data identification, project organization, preprocessing, and model creation have been done. The purpose of this section is to focus on how to consume PMML models using Augustus.

Provide a model schema

The model can be built once every field in the data set has been named (e.g. `date` and `mileage`) and the data types of each field are known (`date` is an ISO 8601 formatted date string; the same date format used in HTML; and `mileage` is an integer).

The goal in this example is to score the values from the `gaslog.xml` data set against the distributions in the example model file `gaslog/introductory/models/example_model.pmml`. The two test distributions chosen are **Chi squared independence** and **z-value** tests.

The z-value test is described first: for every row of data Augustus will score, it compares the new observed data point to a known distribution and calculates the z-value; the number of standard deviations the observed value is away from the mean. Augustus can score against a Gaussian or a Poisson distribution.

The Chi squared test does remember events. That is, for every row of data Augustus will score, it compares the distribution currently in memory with the model distribution and calculates the current score. The score indicates the probability that the observed distribution is different from the model distribution, so a value of one means the observed data are orthogonal to the model data, and a value of zero means the observed data exactly match the model data. More details about the meaning of the score are available at the DMG web site: http://www.dmg.org/.

Run Augustus as a model consumer

The Augustus consumer, like the producer, requires configuration with a XML-formatted file. The consumer configuration file tells the consumer where the model is, where the data are, and where to write the output.

Just like with the producer configuration, the best way to get started is just to open the configuration file and copy and modify if for your needs. An example file is included in the *Example section*. If you would like to get a sense for the output of the Augustus PMML consumer, you can run the consumer (scoring engine) using the example consumer config file. Just copy it into your `<Install-dir>/augustus-examples/gaslog/consumer` directory and then type the following at the command prompt:

```
$ Augustus consumer_config.xcfg
```

The output should look like this:

```
root     : INFO     Loading PMML model.
root     : INFO     Setting up data input.
root     : INFO     Setting up output.
root     : INFO     Setting up Augustus's main engine.
root     : INFO     Calculating.
root     : WARNING  Data not found for field: price
   .
   . (additional warning messages about missing data omitted)
   .
root     : WARNING  Data not found for field: gallons
```

```
root     : WARNING  Data not found for field: date
root     : INFO     Augustus is finished.
```

and a new file, `example_scores.xml` should now be in your

`<Install-dir>/augustus-examples/gaslog/introductory/results/` directory with the output from the PMML consumer.

If you open the configuration file, you will notice that the path to the model file is specified as:

```
<ModelInput>
  <FromFile name="../models/example_model.pmml" />
</ModelInput>
```

If instead you changed directories to `<Install-dir>/augustus-examples/gaslog` and tried to type the following:

```
$ Augustus consumer/consumer_config.xcfg
```

You would see an error message:

```
root     : INFO     Loading PMML model.
Traceback (most recent call last):
  File "/home/odg/augustus-dum/augustus-scoringengine/augustus/bin/Augustus",
    line 49, in <module>
      main(options.config)
  File "/home/odg/augustus-dum/augustus-scoringengine/augustus/engine/mainloop.py",
    line 265, in main
      pmmlModel, pmmlFileName = getModel(child)
  File "/home/odg/augustus-dum/augustus-scoringengine/augustus/engine/mainloop.py",
    line 111, in getModel
      raise RuntimeError, "no files matched the given filename/glob: %s" % filename
 RuntimeError: no files matched the given filename/glob: ../models/example_model.pmml
```

The error message is long. Look at the last line to see the actual message. The rest is just traceback information for a programmer.

```
RuntimeError: no files matched the given filename/glob: ../models/example_model.pmml
```

The message means Augustus could not find the model file because we ran Augustus in the directory `<Install-dir>/augustus-examples/gaslog` and if, from this directory we followed the path `../models/example_model.pmml` we would go up one directory and look for the `models` folder, which does not exist. The solution is to either change the path to the model to become, and then run Augustus from this directory:

```
<ModelInput>
  <FromFile name="models/example_model.pmml" />
</ModelInput>
```

or to change directories back to `<Install-dir>/augustus-examples/gaslog/consumer` and run Augustus from that directory. The point is just to make sure that all of the paths point to the right place...and to introduce you to how Augustus communicates when there are errors.

Note: The PMML consumer will append to, not overwrite, an output file by default. Use the attribute *overwrite="true"* to overwrite any existing output.

There should now be a results file,

<Install-dir>/augustus-examples/gaslog/results/example_scores.xml. Currently Augustus only creates XML-formatted output, but in the future it will support additional formats such as JSON.

Because it can be confusing exactly what the score means for a Chi Squared Distribution, we intentionally trained the model with data from the first half of the data set–through the end of 2006. The predicted value for a Chi Squared Distribution is one if the distribution in the scoring data is orthogonal to the distribution in the model, and zero if the distribution in the scoring data matches the distribution in the model. Events for the dates 2004-01-03 through 2004-11-28 all have a score of one. The score decreases as the observed data start to match the training data, and becomes zero on 2006-12-02, when the PMML model's training data set matches the current amount of information available to Augustus. After that, the score increases as the collected information in Augustus diverges from the PMML model's training set.

Post-processing the results

This section gives an example post-processing step that turns the XML output into a more human-readable CSV format.

With Augustus comes a handful of tools to manipulate PMML model files (this becomes useful when a model has a few thousand segments, to spare the user the tedium of changing things by hand) and to manipulate the output. They are all in the file <Install-dir>/augustus-scoringengine/augustus/bin. The one described here is **ScoresAwk**; named after AWK because its function, structure, and command names are similar. It is not necessary to understand any AWK to continue with the example. Some of the other tools are **PmmlSplit** and **PmmlSed**.

Change directories into the folder: <Install-dir>/augustus-examples/gaslog/postprocess. It contains another configuration file, that will be fed to the **ScoresAwk** script. It contains instructions about how to convert the example_scores.xml file to a CSV format, and output it to a new file example_scores.csv. The file is shown in its entirety below so that it can be described with appropriate context:

```
<ScoresAwk>
  <FileInput fileName="../results/example_scores.xml" excludeTag="output" />

  <PythonFunction condition="BEGIN" action="makeHeader">
    <![CDATA[

# The code goes here, between the braces.
# It should be exactly the same as in a Python script.
# Whatever is returned will be written to the output file.

def makeHeader():
    return "event, date, price, price_score, price_alert, gallons, gallons_score\n"

    ]]>
  </PythonFunction>

  <PythonFunction condition="notEmpty" action="getRow">
    <![CDATA[

def notEmpty(event):
    return len(event.children) > 0

def getContent(seg, tagName):
    return seg.child(tagName).content()

def getRow(event):
    event_no = event["number"]
    the_date = " "
    price = " "
```

```
    price_score = " "
    price_alert = " "
    gallons = " "
    gallons_score = " "

    for segment in event:
        if segment["id"] == "pricePerGal-zValue":
            the_date = getContent(segment, "date")
            price = getContent(segment, "pricePerGal")
            price_score = getContent(segment, "score")
            price_alert = getContent(segment, "alert")

        elif segment["id"] == "gallons":
            gallons = getContent(segment, "gallons")
            gallons_score = getContent(segment, "score")

    return ", ".join([
        event_no,
        the_date,
        price,
        price_score,
        price_alert,
        gallons,
        gallons_score]) + "\n"

    ]]>
  </PythonFunction>

  <FileOutput fileName="../results/example_scores.csv" />
</ScoresAwk>
```

It is in XML format, so tags surround information that is communicated to the program. The outer tag `<ScoresAwk>` just names the program. The entry:

```
<FileInput fileName="../results/example_scores.xml" excludeTag="output" />
```

identifies the location of the input file. The attribute `excludeTag` tells the program to ignore the opening and closing tag in the `example_scores.xml` file. We named it 'output' in our configuration file:

```
<Output>
    <ToFile name="../results/example_scores.xml" overwrite="true" />
    <ReportTag name="output" />
</Output>
```

If instead the `ReportTag` tag was deleted, there would be no 'output' tag wrapping the output in `example_scores.xml` and the `FileInput` entry in the ScoresAwk configuration file would look like:

```
<FileInput fileName="../results/example_scores.xml" />
```

without the `excludeTag` attribute. At the bottom of the file is another entry:

```
<FileOutput fileName="../results/example_scores.csv" />
```

that states where to put the file output. In between are the instructions that tell ScoresAwk how to convert the XML file to CSV. The text inside of the tags is actually Python code that will be given directly to Python to be run:

```
<PythonFunction condition="BEGIN" action="makeHeader">
  <![CDATA[

  # The code goes here, between the braces.
  # It should be exactly the same as in a Python script.
  # Whatever is returned will be written to the output file.

  def makeHeader():
      return "event, date, price, price_score, price_alert, gallons, gallons_score\n"

  ]]>
</PythonFunction>
```

The statement above directs ScoresAwk to run the Python function named `makeHeader` (the action) when the condition `BEGIN` is met (at the beginning, before executing any other code). The `condition` attribute can be `BEGIN`, or `END`, or the name of a function that the user defines in between the `<![CDATA[ ... ]]>` brackets. If there is no `PythonFunction` element that has condition `BEGIN`, then nothing would be done in the beginning (so in this case, there would be no header row in the file). Likewise, if there is no need to do anything after processing the file, there is no need to have any `PythonFunction` element with an `END` condition.

In Python, spaces are meaningful: everything has to be indented the same number of spaces, so if you copy the code keep the indentation the same. Python has a great online manual at http://docs.python.org/tutorial/ for those who want to do more things than copying and pasting. The way to define a function in Python is to write:

```
def functionName():
   # function content goes here (indented by 4 spaces;
   # be careful of tabs --- they will mess you up!)
   print "This is a print statement"
   return "String formatting is like in C, so \n is newline and \t is tab."
```

The long part of the script is for going through the XML output file row by row. The `condition` attribute here tests whether the Event tag for the row contains any information. If it does, then the `action` will be applied:

```
  <PythonFunction condition="notEmpty" action="getRow">
    <![CDATA[

def notEmpty(event):
    return len(event.children) > 0

def getContent(segment, tagName):
    return segment.child(tagName).content()

def getRow(event):
    event_no = event["number"]
    the_date = " "
    price = " "
    price_score = " "
    price_alert = " "
    gallons = " "
    gallons_score = " "

    for segment in event:
        if segment["id"] == "pricePerGal-zValue":
            the_date = getContent(segment, "date")
            price = getContent(segment, "pricePerGal")
            price_score = getContent(segment, "score")
```

```
            price_alert = getContent(segment, "alert")

        elif segment["id"] == "gallons":
            gallons = getContent(segment, "gallons")
            gallons_score = getContent(segment, "score")

    return ", ".join([
        event_no,
        the_date,
        price,
        price_score,
        price_alert,
        gallons,
        gallons_score]) + "\n"

    ]]>
  </PythonFunction>
```

The function `getContent` above is neither the `condition` nor the `action`; but it is called inside of the `getRow` function...the point is to demonstrate that the content between the `<![CDATA[ ... ]]>` tags can really contain any Python code. To run the postprocessing script, type the following at the command prompt:

```
ScoresAwk scores_to_csv.xcfg
```

There should not be any output. When the program is done, the directory `<Install-dir>/augustus-examples/gaslog/results/` should contain the new `example_scores.csv` file.

Congratulations; you have successfully run an Augustus as a PMML consumer!

To more deeply understand what is happening, there needs to be a little explanation about how the XML is being processed. One of the core utilities in the Augustus release is a tool for parsing XML. For the curious, it is located in `<Install-dir>/augustus-scoringengine/augustus/core/xmlbase.py`. ScoresAwk uses this tool. Inside the post processing directory is `explore_xmlbase.py` which demonstrate a few properties of XML elements that are helpful when making one's own ScoresAwk configuration file. It is under `<Install-dir>/augustus-examples/gaslog/introductory/postprocess`. To run it, you need to have already created the file

`<Install-dir>/augustus-examples/gaslog/introductory/results/example_scores.xml` by running Augustus as a consumer. At the command prompt type:

```
$ python explore_xmlbase.py
```

to be guided through some of the basics of Augustus' xmlbase library.

4 Quick introduction to AIM

It is possible to run Augustus in a mode that simultaneously produces and trains new segments and scores existing data. We call this Automatically Incrementing Models (or AIM) mode. This is useful when data are constantly streaming, and where new segments may occur at any time; for example when trying to perform statistical analysis of activity over various IP addresses. With AIM,

- Segments are now automatically created as new data are encountered.
- Segmentation, transformations, and AIM event weighting are independent from the algorithm.
- Data input and model output are in separate threads from the main processor.

and the user can,

- Specify weights to the input data, so that recent data have more impact on the model than old data (e.g. exponential blending and moving windows).
- Weight (or window) consumer data differently from producer data, so that consumers can produce non-trivial results.

Augustus will run in AIM mode if its configuration file contains both output and scoring specifications. The following elements need to be added to the current `test_config.xcfg` file to run in AIM mode:

```
<EventSettings output="true" score="true" />

<Output>
   <ToFile name="../results/scores.xml" overwrite="true" />
   <ReportTag name="Report>
</Output>
```

With these added, the following command will create both a scores file `../results/scores.xml` and the model file `../models/produced_model.pmml`.:

```
$ Augustus test_config.xcfg
```

The output should look like this (lines are wrapped for display purposes):

```
META INFO     New segment created: (car EQ 'old') and ((intmonth GE 6)
                  and (intmonth LE 8)), ID=Untitled-1
META INFO     New segment created: (car EQ 'old') and ((intmonth GT 8)
                  and (intmonth LE 10)), ID=Untitled-2
META INFO     New segment created: (car EQ 'old') and ((intmonth GT 10)
                  and (intmonth LE 12)), ID=Untitled-3
META INFO     New segment created: (car EQ 'old') and ((intmonth GT 0)
                  and (intmonth LE 5)), ID=Untitled-4
META INFO     New segment created: (car EQ 'new') and ((intmonth GE 6)
                  and (intmonth LE 8)), ID=Untitled-5
META INFO     New segment created: (car EQ 'new') and ((intmonth GT 8)
                  and (intmonth LE 10)), ID=Untitled-6
META INFO     New segment created: (car EQ 'new') and ((intmonth GT 10)
                  and (intmonth LE 12)), ID=Untitled-7
META INFO     New segment created: (car EQ 'new') and ((intmonth GT 0)
                  and (intmonth LE 5)), ID=Untitled-8
META INFO     ### Current MetaData content ###
META INFO     Run time : 0.257986783981
```

5 Example Files

Example config.xcfg file: `consumer_config.xcfg`

```
<AugustusConfiguration>
    <!-- Note: XML elements have to appear in the same order as below. -->

    <Logging
        formatString="%(name)-9s: %(levelname)-8s %(message)s" level="INFO">
        <ToStandardError/>

        <!-- <Logging /> ******************************************
            level = one of DEBUG, INFO, WARNING, ERROR
                    (in descending order of verbosity)

            formatString = a string passed directly to Python logging.
                    The formats obey the rules in: http://docs.python.org/
                    library/logging.html#logrecord-attributes

            The output options are:
            <ToStandardError />
            <ToLogFile name="fname.txt" />
            <ToStandardOut />

            The Metadata options are the same as the logging options,
            except that metadata levels are either DEBUG or INFO

            Both Logging and Metadata are optional.  There will be no
            logging output if they are both omitted.
        ****************************************************************
        -->
    </Logging>

    <Metadata
        formatString="META %(levelname)-8s $(message)s" level="DEBUG">
        <ToLogFile name="META.txt"/>
    </Metadata>

    <EventSettings score="true" output="true" />
        <!-- <EventSettings /> ****************************************

            score = "true" or "false" : calculate the score
            output = "true" or "false" : create output

        The alternate tag is <AggregationSettings /> with the exact same
        attributes 'score' and 'output'.  Aggregate objects collect data
        over all of the input stream, such as the total number of
        observations, or the average of the observed value.

        When <AggregationSettings /> is used, the score is calculated at
        the end of the test instead of after every event.
        When both <EventSettings /> and AggregationSettings /> are used,
        the score is calculated both at the end of the test and after
        every event.
        ****************************************************************
        -->
```

```
<ModelInput>
    <!-- <ModelInput /> ***********************************************
     The ModelInput element is always required. It must contain
     a FromFile element that locates the input model file.

    ++++++ <FromFile /> ++++++++++++++++++++++++++++++++++++++++++++++
        name = path to the model file; can include a '*'
                character for globbing
        selectmode = one of "lastAlphabetic"(default) or "mostRecent"
    ************************************************************
    -->
    <FromFile name="../models/example_model.pmml" />

</ModelInput>

<DataInput>
    <!-- <DataInput /> ************************************************
    The DataInput element is always required.  It must contain an
    element that locates the input data file(s).

    ++++++ <ReadOnce /> ++++++++++++++++++++++++++++++++++++++++++++++
    The ReadOnce element is required when reading from a file.
    In future releases, there will be options to read forever from
    a file pipe, or from an HTTP port; for those cases the ReadOnce
    option could be omitted.

    ++++++ <FromFile /> ++++++++++++++++++++++++++++++++++++++++++++++

        The input options are:
        <FromFile name="../../data/other.csv" type="CSV" />
        type = XML (default), CSV, or UniTable

        <FromCSVFile name="../../data/other.csv"
                     header="col1,col2,col3" sep="," />

        <FromFixedRecordFile name="../../data/other.fixed">
            <RecordField name="col1" length="3"/>
            <RecordField name="col2" length="1"/>
            <RecordField name="col3" length="2"/>
        </FromFixedRecordFile>

        <FromStandardIn /> (CSV format only)

        <Interactive />

    ************************************************************
    -->
    <ReadOnce />
    <FromFile name="../../data/gaslog.xml" />

</DataInput>

<Output>
    <!-- <Output /> ***************************************************
    The Output element is required when creating output (scores). It must
    contain a <ToFile /> element that describes where to put the results.
    or else direct results to <ToStandardError /> or <ToStandardOut />; both
```

```
        tags do not have any other attributes.

        ++++++ <ToFile /> ++++++++++++++++++++++++++++++++++++++++++++++++++
            name = location of the output file
            overwrite = 'true' or 'false' (default)

        ++++++ <ReportTag /> ++++++++++++++++++++++++++++++++++++++++++++++
            The ReportTag is optional.  If omitted, there will be no tags wrapping
            the output, meaning the file will not itself be a valid XML file.
            The reason this is allowed is so that multiple test runs can append
            more output to the same file.

        ++++++ <EventTag /> +++++++++++++++++++++++++++++++++++++++++++++++
            The EventTag is optional. If omitted, the name of the tags wrapping
            each event will be 'Event'.  Otherwise it will be whatever is
            in the 'name' attribute.
        ****************************************************************
        -->
        <ToFile name="../results/example_scores.xml" overwrite="true" />
        <ReportTag name="output" />
        <EventTag name="Event" />

    </Output>
</AugustusConfiguration>
```

Example PMML file for a segmented Baseline model

```
<PMML version="4.1">
<!-- <PMML />: [1] ###################################
   The meaning of the top line of this comment, "<PMML />: [1]"
   indicates that the <PMML /> element must occur exactly once...
   every PMML document must have <PMML /> as its root element.

   ## Attributes ##
      - `version`: Required.
      Identifies the version number of the PMML standard used.
      Currently one of:
      "4.1", "4.0", "3.2", "3.1", "3.0", "2.1", "2.0", "1.1"

      Augustus is PMML 4.1 compliant. Use Augustus 0.4.4.1 for
      models written in PMML 4.0 and Augustus 0.3.1 for models
      written in PMML 3.

   ## Introduction ##
   This is an example PMML file that describes a model
      of type: `MiningModel`
      for: the gaslog example in
      {Install-Dir}/augustus-examples/gaslogprimer/

   It identifies mandatory elements of a PMML document,
   and all of the commonly used options for the Mining Model.
   It is meant to be copied and modified.  More details
   are available in the Augustus documentation.

   The Mining Model is one of the types of statistical models that
   Augustus can produce and consume.  PMML version 4.1 can describe
   fourteen types.  For the other models, refer to the PMML schema
   description: www.dmg.org/v4-1/pmml-4-1.xsd

  Augustus can produce and consume segmented models.
 ######################################################################
 -->
  <Header copyright="Open Data Group, 2011" description="example" />
  <!-- <Header />: [1] ******************************************
  The header element is required, but is only informational; nothing
  that goes in the strings affects the model or the data.

     ** Attributes **
        - `copyright`: Optional.
        - `description`: Optional.
  ************************************************************
  -->
  <!-- <MiningBuildTask />: [0-1] *********************************
    Not currently used by Augustus.
  ************************************************************
  -->
  <DataDictionary numberOfFields="6">
  <!-- <DataDictionary />: [1] ************************************
     The elements inside <DataDictionary /> identify each column
     in the input data set.

     ** Attributes **
        - `numberOfFields`: Optional.
```

```
**********************************************************************
-->
  <DataField dataType="date" optype="continuous" name="date" />
  <DataField dataType="double" optype="continuous" name="gallons" />
  <DataField dataType="double" optype="continuous" name="mileage" />
  <DataField dataType="double" optype="continuous" name="miles" />
  <DataField dataType="double" optype="continuous" name="price" />
  <DataField dataType="string" optype="categorical" name="car" />
  <!-- <DataField />: [1-inf] ===================================
     Define the data fields here.  Each <DataField /> corresponds
     to a column in the data set.

     == Attributes ==
      - `name` is Required.
        corresponds to the column name in the data set

      - `optype` is Required.
        one of OPTYPE, (one of the following strings):
        "categorical"  "ordinal"  "continuous"

      - `dataType` is Required.
        one of DATATYPE, (one of the following strings):
        "boolean"  "string"
        "integer"  "float"  "double
        "date"     "time"   "dateTime"
        "dateDaysSince[0]"    "dateDaysSince[1960]"
        "dateDaysSince[1970]" "dateDaysSince[1980]"
        "timeSeconds"
        "dateTimeSecondsSince[0]"     "dateTimeSecondsSince[1960]"
        "dateTimeSecondsSince[1970]"  "dateTimeSecondsSince[1980]"

      - `displayName` Optional.
        a string

      - `taxonomy` Optional.
        a string

      - `isCyclic` Optional; defaults to "0"
        one of "0" or "1"

     == Optional Elements ==
     The options <Interval />,  <Value />, and <Taxonomy />
     are not relevant to Augustus.
  ============================================================
  -->
</DataDictionary>
<!-- <TransformationDictionary />: [0-1] ************************
  Functions and derived fields are defined in this optional section.
  They are then available to be used in the statistical model
  defined below.

   ** Optional Elements **
   <DefineFunction /> and <DerivedField />

   as many of these elements as needed are allowed; all of the
   <DefineFunction /> elements must come before the <DerivedField />
   elements.  For brevity, all of the optional sub-elements and
   attributes for each of these items will not be described here.
```

```
    A good place to look for simple examples, and for definitions
    of functions assumed to be handled by any program that used
    PMML is:
       http://www.dmg.org/v4-0-1/BuiltinFunctions.html

    (search for DefineFunction and DerivedField to see examples
     on the lower section of the page.)

    Example function definition:
       <DefineFunction name="calculate_cylinder_volume"
                       optype="continuous"
                       dataType="double">
           <ParameterField name="radius" optype="continuous" />
           <ParameterField name="height" optype="continuous" />
           <Apply function = "*" >
               <Constant dataType="double"> 3.14159 </Constant>
               <Apply function = "*" >
                   <FieldRef field="height" />
                   <Apply function="pow">
                       <FieldRef field="radius" />
                       <Constant dataType="double"> 2 </Constant>
                   </Apply>
               </Apply>
           </Apply>
       </DefineFunction>

    The <DefineFunction /> attributes specify the function's
    name(`name`) and its return type (`dataType`).
    The <ParameterField /> elements specify the function's arguments;
    they will be referred to by their `name` in the function
    definition.  There are eight types of expression elements
    available to define the function body:
    <Constant /> <FieldRef /> <NormContinuous /> <NormDiscrete />
    <Discretize /> <MapValues /> <Apply /> <Aggregate />


  Example definition of a derived field:
  (Also shows how to call a function)
       <DerivedField name="volume_of_cylinder1"
                     optype="continuous"
                     dataType="double">
           <Apply function="calculate_cylinder_volume">
               <FieldRef field="r1" />
               <FieldRef field="h1" />
           </Apply>
       </DerivedField>
  Function arguments are identified using the <FieldRef />
  element. Arguments are expected to be in the same order
  as in the function definition, and `field` should be equal to
  the `name` in either a <DataField /> in the <DataDictionary />
  or as another <DerivedField />.

    ** Attributes **
      none.
*****************************************************************
-->
<MiningModel functionName="regression">
```

```
<!-- <MiningModel />: [0-inf] of one choice **********************
  For brevity, only the MiningModel option is explained in this
  file.

  The MiningModel is the only top-level model that can use
  segmentation.  A segment can contain other types of models,
  as described in <BaselineModel /> below.

   ** Attributes **
     - 'modelName': Optional.
       a string.

     - 'functionName': Required.
       one of MINING-FUNCTION, (one of the following strings):
       "associationRules"  "sequences"  "classification"
       "regression"  "clustering"  "timeSeries"

      - 'algorithmName': Optional.
        a string.
*****************************************************************
-->
  <MiningSchema>
  <!-- <MiningSchema />: [1] ====================================
    The Mining schema identifies every <DataField /> from the
    <DataDictionary /> and every <DerivedField /> from the
    <TransformationDictionary /> that will be used in the
    statistical model, and (optionally) how it will be used.

     == Attributes ==
       none.
   ===============================================================
   -->

    <MiningField name="date" />
    <MiningField name="gallons" />
    <MiningField name="miles" />
    <MiningField name="price" />
    <MiningField name="car" />
    <!-- <MiningField />: [1-inf] ++++++++++++++++++++++++++++++++
      Each entry identifies a single <DataField />
      or <DerivedField /> that will be used in the model.

       ++ Attributes ++
         'name': Required.
         a string that matches the name of a previously
         defined <DataField /> or <DerivedField />.

         'usageType': Optional; defaults to "active"
         one of FIELD-USAGE-TYPE, (one of the following strings):
         "active"  "predicted"  "supplementary"  "group"
         "order"   "frequencyWeight"    "analysisWeight"

         'optype': Optional.
         one of OPTYPE, (one of the following strings):
         "categorical"  "ordinal"  "continuous"

         'importance': Optional.
         type is PROB-NUMBER
```

```
        (basically a double/the widest available floating point number)

        `outliers`: Optional; defaults to "asIs"
        one of OUTLIER-TREATMENT-METHOD, (one of the following):
        "asIs"  "asMissingValues"  "asExtremeValues"

        `lowValue`: Optional.
        a double.

        `highValue`: Optional.
        a double.

        `missingValueReplacement`: Optional.
        a string.

        `missingValueTreatment`: Optional.
        one of MISSING-VALUE-TREATMENT-METHOD, (one of the following):
        "asIs"  "asMean"  "asMode"  "asMedian"  "asValue"

        `invalidValueTreatment`: Optional; default: "returnInvalid"
        one of INVALID-VALUE-TREATMENT-METHOD, (one of the following):
        "returnInvalid"  "asIs"  "asMissing"
    ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    -->
  </MiningSchema>
  <Output>
<!-- <Output />: [0-1] ====================================
   The Output identifies every <MiningField /> from the
   <MiningSchema /> that will be written to the output
   file.  Each is listed as a separate <OutputField />
   child element.

    == Attributes ==
      none.
 ================================================================
 -->

   <OutputField name="date" />
   <!-- <OutputField />: [1-inf] ++++++++++++++++++++++++++++++++
     Each entry identifies a single <MiningField />
     whose values will be printed

      ++ Attributes ++
        `name`: Required.

        `displayName`: Optional.
        if present, the displayName will be the tag name for the
        output (which is in XML format) instead of the actual
        field name.

        `feature`: Optional.
        This is only used with the MiningField that names the
        predicted value; an example is shown below.
    ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    -->
  </Output>
<!-- ============================================================
   The following are not currently used by Augustus, but
```

```
  they would appear in order if used.
     <ModelStats />: [0-1]  ... Descriptive information.
     <ModelExplanation />: [0-1] ... Descriptive information.
     <Targets />: [0-1]
 ================================================================
 -->

<LocalTransformations>
<!-- <LocalTransformations />: [0-1] ============================
  Contains a list of <DerivedField /> elements to supplement
  the ones in the <TransformationDictionary />, if desired.

   == Attributes ==
     none.
 ================================================================
-->
    <DerivedField name="year" optype="continuous" dataType="integer">
        <Apply function="formatDateTime">
            <FieldRef field="date" />
            <Constant dataType="string">%Y</Constant>
        </Apply>
    </DerivedField>
    <DerivedField name="month" optype="continuous" dataType="string">
        <Apply function="formatDateTime">
            <FieldRef field="date" />
            <Constant dataType="string">%b</Constant>
        </Apply>
    </DerivedField>
    <DerivedField dataType="double" optype="continuous" name="pricePerGal">
        <Apply function="/">
            <FieldRef field="price" />
            <FieldRef field="gallons" />
        </Apply>
    </DerivedField>
</LocalTransformations>

<Segmentation multipleModelMethod="selectAll">
<!-- <Segmentation />: [0-1] ====================================
  Allows the model to be split. For example, to score different
  categories separately.

   == Attributes ==
     `multipleModelMethod`: Required.
     one of MULTIPLE-MODEL-METHOD, (one of the following strings):
     "selectFirst"    "selectAll" (the remaining methods are
     currently not available in Augustus:
     "majorityVote"  "weightedMajorityVote"  "average"
     "weightedAverage"    "median"    "max"  "sum" )
 ================================================================
-->
  <Segment id="pricePerGal-Zvalue">
   <!-- <Segment />: [1-inf] ++++++++++++++++++++++++++++++++++
      Defines one segment of a segmented model.

      ++ Attributes ++
        `id`: Optional.
       a string
```

```
    'weight': Optional.
    a double.
 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
-->
<True />
<!-- <True />: [1] ++++++++++++++++++++++++++++++
  One PREDICATE element is required in each <Segment />.
  The options are:
     <SimplePredicate /> <CompoundPredicate />
     <SimpleSetPredicate />
     <True /> <False />


  ++ <SimplePredicate /> ++
   + Attributes +
    'field': Required.
    a string corresponding to the 'name' attribute of a
    <DataField /> or of a <DerivedField />.

    'operator': Required.
    one of the following strings:
    "equal"  "notEqual"  "lessThan"  "lessOrEqual"
    "greaterThan"  "greaterOrEqual"  "isMissing"
    "isNotMissing"

    'value': Optional.
    a string.


  ++ <CompoundPredicate /> ++
   + Elements +
    Contains [2-inf] <SimplePredicate /> elements.

   + Attributes +
    'booleanOperator': Required.
    one of the following strings:
    "or"  "and"  "xor"  "surrogate"


  ++ <SimpleSetPredicate /> ++
   + Elements +
    Contains [1] <Array /> element. (not described for brevity.)

   + Attributes +
    'field': Required.
    a string corresponding to the 'name' attribute of a
    <DataField /> or of a <DerivedField />.

    'booleanOperator': Required.
    one of the strings:  "isIn"  "isNotIn"

  ++ <True /> or <False /> ++
    no elements or attributes.
 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
-->
 <BaselineModel functionName="regression">
 <!-- <BaselineModel />: [1] ..............................
 The PMML standard does not allow all types of models inside
```

```
of a segment>.
  The ones that are allowed and supported by Augustus are:
  <BaselineModel /> <TreeModel /> <RuleSetModel />
  <ClusteringModel />

  For brevity, only the attributes for
  <BaselineModel /> are described.

  .. Attributes ..
    `modelName`: Optional.
    a string

    `functionName`: Required.
    one of MINING-FUNCTION, (one of the following strings):
    "associationRules"  "sequences"  "classification"
    "regression"  "clustering"  "timeSeries"

    `algorithmName`: Optional.
    a string
 ..........................................................
 -->
  <MiningSchema>
  <!-- <MiningSchema />: [1] ,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
     Same definition as above. No attributes.
   ,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
   -->
    <MiningField name="pricePerGal" />
    <MiningField usageType="predicted" name="score" />
  </MiningSchema>
  <Output>
      <OutputField name="pricePerGal" />
      <OutputField name="score" feature="predictedValue" />
  </Output>
  <!-- ,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
     As before, these elements can appear, in order, but are
     not used.
       <ModelStats />: [0-1]  ... Descriptive information.
       <ModelExplanation />: [0-1] ... Descriptive information.
       <Targets />: [0-1]
     Augustus would use the following, if present:
       <LocalTransformations />: [0-1]
   ,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
   -->
  <TestDistributions field="pricePerGal" testStatistic="zValue" >
  <!-- <TestDistributions >: [1] ,,,,,,,,,,,,,,,,,,,,,,,,,,,,
     Defines the distributions for the data used in this segment.

     ,, Attributes ,,
       `field`: Optional.
       would identify a field name defined in the data dictionary
       or the translation dictionary

       `testStatistic`: Required.
       one of BASELINE-TEST-STATISTIC, (one of the following):
       "zValue"  "chiSquaredIndependence" "chiSquaredDistribution"
       "CUSUM" "scalarProduct"

       `resetValue`: Optional; defaults to 0.0.
```

```
          a double

          'windowSize': Optional; defaults to 0
          an integer

          'weightField': Optional
          a Field Name. An example for this is in the second segment,
          below. it is the name of the field that will be accumulated
          in the "count" attribute of the <FieldValueCount /> element.

          'threshold': Required.
          a double.
      ''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
      -->
       <Baseline>
       <!-- <Baseline />: [1] """""""""""""""""""""""""""""""""""
          Contains the baseline distribution.
        """""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
        -->
         <GaussianDistribution mean="2.5" variance="0.3" />
         <!-- <GaussianDistribution />: [1] or other ~~~~~~~~~~
            One of the following options for a continuous distribution are
            allowed:
            <AnyDistribution mean={double} variance={double} />
            <GaussianDistribution mean={double} variance={double} />
            <PoissonDistribution mean={double} />
            <UniformDistribution lower={double} upper={double} />

            Plus a set of options for discrete distributions;
            the Chi squared distribution is described in the second
            segment, below.
         ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
          -->
       </Baseline>
       <!-- <Alternate />: [0-inf] """""""""""""""""""
          An alternate distribution is allowed when
          using the CUSUM test statistic. It would have the
          same contents as are allowed in <Baseline /> above.
        """""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
        -->
     </TestDistributions>
   </BaselineModel>
</Segment>
<Segment>
 <!-- <Segment />: [1-inf] ++++++++++++++++++++++++++++++++++
     This segment gives an example of the chiSquareDistribution.
     Outside of the description of the distribution, and the type of
     predicate that determines which data will be scored in this segment,
     the contents of the two segments are similar.
   ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
   -->
   <CompoundPredicate booleanOperator="and">
       <SimplePredicate operator="equal" field="car" value="old" />
       <SimplePredicate operator="greaterOrEqual" field="year" value="2004" />
       <SimplePredicate operator="lessOrEqual" field="year" value="2008" />
   </CompoundPredicate>

   <BaselineModel functionName="regression">
```

```
          <MiningSchema>
            <MiningField name="date" />
            <MiningField name="gallons" />
            <MiningField name="score" usageType="predicted" />
            </MiningSchema>

            <Output>
              <OutputField name="gallons" />
              <OutputField name="score" feature="predictedValue" />
            </Output>

            <TestDistributions field="month" testStatistic="chiSquareDistribution"
                               weightField="gallons">
              <Baseline>
                <CountTable sample="1001.477">
              <!-- <CountTable />: [1] or other ~~~~~~~~~~
                 One of the following options for a discrete distribution are
                 allowed:
                 <CountTable sample={double} />
                 <NormalizedCountTable sample={double} />

                 The contents of either of these are basically the bins in
                 a histogram that will define the distribution. These contents
                 are either
                     <FieldValue field={field name} value={the value} />
                 or
                 <FieldValue field={field name} value={the value}
                             count={weight field sum}/>
              ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
               -->
                  <FieldValueCount count="134.16" field="month" value="Jan" />
                  <FieldValueCount count="86.1" field="month" value="Feb" />
                  <FieldValueCount count="50.264" field="month" value="Mar" />
                  <FieldValueCount count="51.863" field="month" value="Apr" />
                  <FieldValueCount count="90.668" field="month" value="May" />
                  <FieldValueCount count="93.123" field="month" value="Jun" />
                  <FieldValueCount count="81.501" field="month" value="Jul" />
                  <FieldValueCount count="153.12" field="month" value="Aug" />
                  <FieldValueCount count="38.96" field="month" value="Sep" />
                  <FieldValueCount count="63.644" field="month" value="Oct" />
                  <FieldValueCount count="91.858" field="month" value="Nov" />
                  <FieldValueCount count="66.216" field="month" value="Dec" />
                </CountTable>
              </Baseline>
            </TestDistributions>

        </BaselineModel>
      </Segment>
    </Segmentation>
  </MiningModel>
</PMML>
```

Example config.xcfg file: `baselineproducer_config.xcfg`

```
<AugustusConfiguration>
```

```
<Logging
    formatString="%(name)-9s: %(levelname)-8s %(message)s" level="INFO">
    <ToStandardError/>

    <!-- <Logging /> ******************************************
        level = one of DEBUG, INFO, WARNING, ERROR
                (in descending order of verbosity)

        formatString = a string passed directly to Python logging.
                The formats obey the rules in: http://docs.python.org/
                library/logging.html#logrecord-attributes

        The output options are:
        <ToStandardError />
        <ToLogFile name="fname.txt" />
        <ToStandardOut />

        The Metadata options are the same as the logging options,
        except that metadata levels are either DEBUG or INFO

        Both Logging and Metadata are optional.  There will be no
        logging output if they are both omitted.
    ****************************************************************
    -->
</Logging>

<Metadata
    formatString="META %(levelname)-8s %(message)s" level="DEBUG">
    <ToLogFile name="META.txt" />
</Metadata>

<ModelInput>
    <!-- <ModelInput />***********************************************
     The ModelInput element is always required. It must contain a
     a FromFile element that locates the input model file.

    ++++++ <FromFile /> ++++++++++++++++++++++++++++++++++++++++++++
        name = path to the model file; can include a '*'
                character for globbing
        selectmode = one of "lastAlphabetic"(default) or "mostRecent"
    ****************************************************************
    -->
    <FromFile name="models/model_template.pmml" />
</ModelInput>

<DataInput>
    <!-- <DataInput /> ***********************************************
    The DataInput element is always required.  It must contain an
    element that locates the input data file(s).

    ++++++ <ReadOnce /> ++++++++++++++++++++++++++++++++++++++++++++
    The ReadOnce element is required when reading from a file.
    In future releases, there will be options to read forever from
    a file pipe, or from an HTTP port; for those cases the ReadOnce
    option could be omitted.

    ++++++ <FromFile /> ++++++++++++++++++++++++++++++++++++++++++++
    The input options are:
```

```
        <FromFile name="../../data/other.csv" type="CSV" />
        type = XML (Default), CSV, or UniTable

        <FromCSVFile name="../../data/other.csv" header="col1,col2,col3" sep="," />

        <FromFixedRecordFile name="../../data/other.fixed">
            <RecordField name="col1" length="3"/>
            <RecordField name="col2" length="1"/>
            <RecordField name="col3" length="2"/>
        </FromFixedRecordFile>

        <FromStandardIn /> (CSV format only)

        <Interactive />
    ****************************************************************
    -->
    <ReadOnce />
    <FromFile name="../data/gaslog.xml" />
</DataInput>

<ModelSetup outputFilename="models/produced_model.pmml" mode="replaceExisting"
            updateEvery="event">
    <!-- <ModelSetup /> ********************************************
        outputFilename = path to the output model.

        mode = 'replaceExisting' (default) or 'updateExisting' or 'lockExisting';
            The mode identifies what to do with the probability distributions
            currently in the model.
                replaceExisting: delete the existing properties of the distribution
                updateExisting: in the first run, will delete the existing properties
                    of the distribution, and in subsequent runs will continue to
                    add to them using new data
                lockExisting: freeze the existing properties of the distribution
                    and only use the training data for the automatically-generated
                    segments.

        updateEvery = 'event' (default) or 'aggregate'
            When to update the properties of the distributions; either every
            event, or upon an aggregation.

    ++++++ <ProducerBlending /> ++++++++++++++++++++++++++++++++++++
        The ProducerBlending element sets up how the AIM mode,
        Automatic Incremental Model updating, will occur.  It has
        the following attributes:

        method = 'exponential' 'unweighted' 'window' 'computerTimeWindowSeconds'
            the Producer Blending method determines how the current observation
            will be combined with prior data to update the properties of the model.
            'exponential' means exponential weighting,
            'unweighted' means pure averaging,
            'window' means average over the given window (number of events)
            'computerTimeWindowSeconds' means average over the events that
                have been seen in the past window (number of seconds) based
                on the computer's clock time.

        alpha = a double.
            The attenuation rate of information used in the exponential
            weighting method.  This can mean different things for different
```

```
        applications, and we recommend you either examine
        augustus-scoringengine/augustus/algorithms/eventweighting.py
        to completely understand what is happening, or contact us.
        It is an error to specify alpha when the method is not 'exponential'.

    windowSize = an integer.
        The number of events or number of seconds to be used if the
        'window' or 'computerTimeWindowSeconds' options are selected.
        It is an error to specify windowSize when the method is not
        'window' or 'computerTimeWindowSeconds'


    ProducerBlending also can contain one sub element:

        <MaturityThreshold threshod="1"/>, which indicates when a model is
        sufficiently trained to return a valid score.  Until the
        MaturityThreshold is reached, the score will be 'Invalid'.
        MaturityThreshold has two attributes:

        threshold = an integer.
            The number of observations required before the model's scores
            are considered valid.

        lockingThreshold = an integer.
            For performance, when in AIM mode, updating will be turned
            off after this number of observations have been seen.  The
            caveat here is that the model will no longer be updating,
            meaning any drift will not be captured.


++++++ <SegmentationSchema /> ++++++++++++++++++++++++++++++++++
    The SegmentationSchema element sets up how automatic segment
    generation will occur; the details are in its sub elements:

      <BlacklistedSegments /> Matching records will be ignored.
      <SpecificSegments /> Matching non-blacklisted records will be added.
      <GenericSegment /> Matching non-blacklisted non-added records are added.

    The BlacklistedSegments, SpecificSegments, and GenericSegment
    entries all have the same two types of sub-elements:
        <EnumeratedDimension />
        <PartitionedDimension />

    ... <EnumeratedDimension /> ................................
        field = The name of a MiningField element that appears in
                the MiningSchema of the MiningModel in the input
                PMML file.

        At least one <Selection /> element is required in every
        EnumeratedDimension unless it is in the GenericSegment.
        In the GenericSegment, when an EnumeratedDimension does
        not have any named Selection elements, every new value
        seen will generate a new segment.

        Each Selection in an EnumeratedDimension identifies a
        match for this field. It has the attributes:
        ____________________________________________________________

            value = The value that the field will take on
```

```
                operator = 'equal'(default) or 'notEqual'
                    if it is notEqual, then only one Selection entry
                    can exist for this field.

            ... <PartitionedDimension /> ..............................
                field = The name of a MiningField element that appears in
                        the MiningSchema of the MiningModel in the input
                        PMML file.

                Each <Partition /> in a PartitionedDimension identifies
                a match range for this field. It has the attributes:
                ___________________________________________________________

                    low = The low value in the range. Can be absent
                        if the closure on the left hand side is open.

                    high = The high value in the range. Can be absent
                        if the closure on the right hand side is Open.

                    divisions = The number of divisions that will be made
                        between the low and the high ends of the range

                    closure = 'openClosed'(default) or 'closedClosed' or
                        'openOpen' or 'closedOpen'
                        This determines the closure of every partition
                        that will be made.  This means the range can only
                        be 'closedClosed' or 'openOpen' if the 'divisions'
                        attribute is absent; meaning the low and high
                        values define one single partition; because otherwise
                        the boundaries would either overlap or be excluded.
        ****************************************************************
        -->
        <SegmentationSchema>
            <GenericSegment>
                <EnumeratedDimension field="car" />
                <EnumeratedDimension field="month" />
            </GenericSegment>
        </SegmentationSchema>
    </ModelSetup>

</AugustusConfiguration>
```

6 Glossary

Augustus A *PMML*-compliant software toolkit written in Python. Augustus is an open source system for building and scoring statistical models designed to work with data sets that are too large to fit into memory

Baseline Model Augustus uses the Baseline model element in PMML to describe a change detection method. The user selects a distribution that describes the data under normal conditions, then chooses a test type, a test statistic, and a threshold value. The Augustus PMML consumer would score the data using the given test, and if instructed, raise a flag when the test statistic exceeds the selected threshold.

Baseline Producer Augustus's Baseline Producers creates a PMML baseline model from three inputs: a configuration file, *PMML Skeleton* File, and a set of training data. It uses the training data to determine the baseline mean (and if used, the variance) for each section of data, called a *segment*.

big data Data that are too large to be effectively processed using many statistical tools because the data cannot fit in memory or the database does not fit on a single computer. For example: the network traffic on a popular web host, a time series of temperature, position, and population data for all algae species measured on every square foot of the surface of the ocean, or all of the trades for that occurred at a particular exchange for a portfolio of symbols during a given time period.

CSV An acronym for the Comma Separated Variable files. Augustus can read `.csv` files with any delimiter; it uses its *UniTable* to deduce the column separator from the first line of the file. The first line must be a header row.

CUSUM The CUSUM statistical test is a change detection test that measures the difference between a reference distribution and the data set of a new distribution. More details are available on wikipedia at http://en.wikipedia.org/wiki/Cusum.

DATATYPE This is a list of base data types defined in PMML: `string`, `integer`, `float`, `double`, `boolean`, `date`, `time`, `dateTime`, `dateDaysSince[0]`, `dateDaysSince[1960]`, `dateDaysSince[1970]`, `dateDaysSince[1980]`, `timeSeconds`, `dateTimeSecondsSince[0]`, `dateTimeSecondsSince[1960]`, `dateTimeSecondsSince[1970]`, `dateTimeSecondsSince[1980]`

DMG An acronym for the Data Mining Group, an independent, vendor led consortium that develops data mining standards, such as PMML. The group's web page is http://www.dmg.org.

GLR Generalized Likelihood Ratio. This is one option for the test statistic in the *Baseline Model*. The definition Augustus uses is the one given by Basseville and Nikiforov [Basseville-1993]; it is a test statistic that compares the set of events in a moving window to a known distribution. Currently the user specifies the number of events in the moving window, and the properties of a baseline Gaussian distribution.

NAB Num Array Binary is a format used by Augustus and Augustus's UniTable to speed up file reading. The file contains a text header with the name and format of each column of data in the UniTable data structure; the remainder of the file is the data structure itself, saved in binary format. It speeds up re-reading of files by around two orders of magnitude.

To convert a CSV file to NAB format when in the same directory as the file, start a Python session, and then type the following:

```
>>> from augustus.kernel.unitable import UniTable
>>> tbl = UniTable()
>>> tbl.fromfile('data_file.csv')
... method output omitted
>>> tbl.to_csv_file('tab_delimited_data_file.csv')
>>> tbl.to_nab_file('data_file_converted.nab')
>>> exit()
```

OPTYPE This is a list of allowable optypes defined in PMML: `categorical`, `ordinal`, `continuous`

PMML The Predictive Model Markup Language, developed by the Data Mining Group (*DMG*). It is an *XML*-based language that provides a standard way to communicate the contents of a data file, plus the statistical model that is used to analyze the data in that file. The benefit of a standard is that developers can create, test, and refine a statistical model using one PMML-compliant application, and then deploy it using another PMML-compliant application without having to write any custom code to translate between the two applications.

The PMML language is described by a standard, which is available in the form of an *XML Schema* for those who are interested in developing PMML-compliant applications. (Version 4.0 is can be found at: http://www.dmg.org/v4-0-1/pmml-4-0.xsd and is included in the Augustus distribution.) Those who just want to use Augustus will find enough information in the comments of the example PMML file **example_model.pmml**.

PMML Consumer A program or application that takes two inputs: a data set, and a statistical model for the data, described in a PMML file. The PMML Consumer then outputs scores for the data set based on the instructions in the PMML file. A PMML file has the extension `.pmml`. Augustus can be run as a PMML Consumer.

PMML Producer Augustus's PMML Producers require a configuration file, a PMML Skeleton File, and a set of training data. Producers can use the training data to automate addition of Segments to a Mining Model, and also to identify the properties of the distribution identified for each segment. (For example, the mean for a Poisson distribution, or the mean and variance for a Gaussian distribution.) Augustus can be run as a PMML Producer.

PMML Skeleton A PMML skeleton is an incomplete PMML file that is used for input into Augustus's PMML producer. The skeleton file's Data Dictionary and Transformation Dictionary should be complete. In the section where the model definition should be, the skeleton file should name the type of model to be used and list the fields to be used in that model. The configuration file for Augustus's PMML producer will identify all of the segments to place in the model, and the type of test distribution and test statistic to use for scoring. The output of an Augustus PMML producer is a complete PMML file. An example PMML skeleton and producer configuration file are in `<Install-dir>/augustus-examples/gaslog/primer`.

Python Augustus is written in the Python programming language. The project web page is http://www.python.org.

R R is a programming language and environment for statistical computing and graphics. The R project's web site is:http://www.r-project.org.

segment A segment is a subset of the total range of available outcomes for the data. It can be thought of as a cube or slice in the outcome space: for example, one sensor name out of hundreds, or one day in the week, or a set of hours in the day.

Or a combination of these: for example morning rush hour (from 5am to 9am, say) on weekdays at a specific traffic sensor. Then, other segments could be evening rush hour, and all other times, or some other subdivision of the possible days, times, and sensors.

UniTable An Augustus Python module, a data structure, and a file format created by Open Data Group and included in the Augustus package. The Python module defines the data structure and its properties. The data structure is analogous to a data frame in the **R** statistical programming system: briefly, a data structure containing an array. The array's columns can have different data types, and are accessible by an index number or a string label. It can also convert CSV files to XML or UniTable files. When a file format is called `UniTable`, it refers to the *NAB* file format.

See also the UniTable section of the Augustus code site: http://code.google.com/p/augustus/wiki/UniTable

XML An acronym for the eXtended Markup Language; a format for communicating information that can be structured as a tree. Examples of XML-based languages are PMML, which Augustus uses to describe its statistical models; and the HyperText Markup Language (HTML), which is used to describe the layout of a web page. The Augustus configuration files are also written using an XML format: one to produce a PMML model, and another to consume a PMML model.

XML Schema A standardized way to communicate the format of an *XML*-based language. PMML is described using the schema located at http://www.w3.org/2001/XMLSchema; developers who wish to contribute to modifications of future PMML standards would need to understand the schema; those who just want to use Augustus need not go to this level of detail.

Z-Value The Z-Value test is a change detection test that measures the how likely an observation is to occur in relation to a reference distribution. More details are available on wikipedia at http://en.wikipedia.org/wiki/Standard_score.

References

[Augustus-2011] The Augustus project site hosted on Google Code.

[Basseville-1993] Michèle Basseville and Igor Nikiforov. *Detection of Abrupt Changes: Theory and Application*. (1993). Available from: http://www.irisa.fr/sisthem/kniga/.

[Chaves-2006] John Chaves, Chris Curry, Robert L. Grossman, David Locke and Steve Vejcik. Augustus: The Design and Architecture of a PMML-Based Scoring Engine, *Proceedings of ACM KDD Workshop on Data Mining Standards, Services, and Platforms (DM-SSP 2006)*, ACM, 2006.

[Grossman-2005a] Robert L. Grossman, PMML Models for Detecting Changes, *Proceedings of ACM KDD Workshop on Data Mining Standards, Services, and Platforms (DM-SSP 2005)*, ACM, 2005.

[Grossman-2005b] Robert L. Grossman, Michal Sabala, Anushka Aanand, Steve Eick, Leland Wilkinson, Pei Zhang, John Chaves, Steve Vejcik, John Dillenburg, Peter Nelson, Doug Rorem, Javid Alimohideen, Jason Leigh, Mike Papka and Rick Stevens. Real time change detection and alerts from Highway Traffic Data, *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. (2005). Available from: http://portal.acm.org/citation.cfm?id=1105834.

[PMML-2011] The Predictive Model Markup Lanaguage (PMML), www.dmg.org.

[R-2011] The R Project for Statistical Computing, www.r-project.org.

[Taylor-2007] James Taylor and Neil Raden. Smart (Enough) Systems.

Index

A

B

C

D

G

N

O

P

R

S

U

X

Z