Augustus 0.6: Design and Implementation of a Hot-Swappable PMML Scoring Engine

Jim Pivarski, Collin Bennett, and Robert Grossman

Open Data Group
400 Lathrop Ave, Suite 90
River Forest, IL 60305

ABSTRACT

This paper describes a new version of Augustus, a PMML model producer and scoring engine. Among its key features is the ability to change the implementation of and/or definition of PMML interactively. This provides a more fluid view of the PMML language and the option to experiment with new PMML-like forms, side-by-side and without needing to reload test data. The paper describes other features that are important for using PMML in a production environment, such as management of execution state, a fast implementation, and a MapReduce-ready design. With the perspective of having written a fresh implementation, we also comment on the status of the PMML language.

1. INTRODUCTION

For several years, the Augustus project[1] has provided an open-source Python[2] implementation of the Predictive Model Markup Language (PMML)[3]. One of the advantages of Python is its ability to dynamically redefine functions and classes on the fly. The upcoming version 0.6 of Augustus extends this ability to PMML.

Models are represented in Augustus as a data-binding of the XML elements of a PMML file. For each element

```
<SomeModel> ... </SomeModel>
```

in the PMML file, there is a corresponding instance of a "SomeModel" class in Augustus. This class contains code to execute ("score") the model for a given dataset as described by the PMML specification, and Augustus has tools to build ("produce") PMML models from its classes. The focus of this paper is the new ability to redefine the classes themselves or bind new classes at runtime, either to provide an alternate implementation with different performance characteristics or to change or add new model types. In the latter case, these new models are not strict PMML, but they may be candidates for future inclusion in the PMML specification. Thus, Augustus may be used as a laboratory for developing new PMML schema.

PMML '13, Aug 10, 2013, Chicago, IL, USA

2. REQUIREMENTS

The Augustus 0.6 upgrade was motivated by four main requirements:

- Model implementations must be hot-swappable. Augustus must support multiple, simultaneous "interpretations" of the PMML language as mutable objects. An "interpretation" is an XSD schema for local validation, an optional XSLT stylesheet for global validation, and a mapping from XML elements to Python classes, which is used to load a PMML file into memory for execution. Since these rules are encapsulated in distinct objects, the user can experiment with multiple model implementations for side-by-side performance tests and multiple PMML-like languages for side-by-side evaluation.

- The state of the PMML model's execution must be saved in an intermediate object, to use the state in future Augustus invocations, transmit the state to another site or across a distributed service, resume an archived calculation, or rewind to a trusted state. In other words, the PMML engine must be virtualized like a virtual machine.

- The execution must be fast. Since Python is a dynamic, interpreted language, numerical analysis in pure Python is orders of magnitude slower than the equivalent in C or Java. However, the Numpy[4] library provides compiled numerical algorithms to the Python interpreter.

  Augustus should take advantage of Numpy's speed by performing calculations in a column-wise order (one operation at a time for all records in the dataset), rather than row-wise (one record at a time for all operations in the algorithm), so that all loops over large datasets are performed in compiled code. Similarly, handling of large XML files is performed in compiled code by the lxml[5] library.

- Algorithms should be MapReduce-ready. MapReduce[6] is a popular paradigm for distributing processes over datasets that are too large for any one computer. Algorithms designed with the assumption that the whole dataset is contained on one machine usually need to be refactored to be deployed in a MapReduce framework like Hadoop[7].

  To avoid a future rewrite, Augustus 0.6 algorithms were designed in modules that can be easily split among

mappers and reducers. Augustus also includes a framework for testing MapReduce-ready algorithms on a single machine using threads; this same framework also automates the deployment of those algorithms in Hadoop.

3. DETAILS OF THE IMPLEMENTATION

3.1 Hot-Swappable Model Elements

Each of Augustus's interpretations of PMML is encoded in an instance of a ModelLoader class. The ModelLoader's primary responsibility is to load a PMML file into executable Python objects. This is the step in which XML structures are validated against the PMML schema and tags are associated with executable code— when the interpretation matters most. ModelLoaders are used for all construction of in-memory PMML objects, so that all models are filtered through the same lens.

The Augustus package comes with two ModelLoader instances, a strict ModelLoader that adheres to the official specification, and a custom one that introduces new functions, model elements, and capabilities, at the expense of producing non-portable output.

3.1.1 Example 1: New PMML-like Elements

One of the new elements, <Formula>, allows for a more succinct way to encode PMML expressions. Mathematical formulae placed between <Formula> tags are parsed and interpreted as the equivalent deeply nested PMML expression. For instance,

```
<Formula>(x + y)**2 + f(y)</Formula>
```

is equivalent to

```
<Apply function="+">
   <Apply function="pow">
      <Apply function="+">
         <FieldRef field="x"/>
         <FieldRef field="y"/>
      </Apply>
      <Constant dataType="integer">2</Constant>
   </Apply>
   <Apply function="f">
      <FieldRef field="y"/>
   </Apply>
</Apply>
```

Since the <Formula> element is not defined in the current version of the PMML specification (4.1), the strictly compliant ModelLoader does not recognize it. However, the custom ModelLoader does recognize it because we have added an association between the "Formula" tag name and a class that knows how to parse the formula string.

New elements like this can be defined in a Python script or interactively in the interpreter by registering a new class with the ModelLoader. The ability to register and re-register implementations without restarting the Python process shortens the development cycle for new PMML elements, particularly if they are tested against a large or slow-to-build dataset.

3.1.2 Example 2: Reimplementing Existing Elements

Redefining model implementations on the fly is also beneficial for performance tuning. This use-case does not change the meaning of the PMML language, so it can be used in situations where models must be strictly PMML compliant. Two or more ModelLoaders can be loaded with different implementations of the same element, and these can be run on the same dataset in the same process, minimizing the difference between the tests[8]. For instance,

```
import copy
from augustus.strict import *

# Split the modelLoader into two genetic lines.
modelLoader1 = copy.deepcopy(modelLoader)
modelLoader2 = copy.deepcopy(modelLoader)

modelLoader1.register("TreeModel",
    MyTreeModelTest1)
modelLoader2.register("TreeModel",
    MyTreeModelTest2)

# Load the same file two ways: <TreeModel>
# elements will be instantiated as MyTreeModelTest1s
# in model1 and MyTreeModelTest2s in model2.
model1 = modelLoader1.loadXml("file.pmml")
model2 = modelLoader2.loadXml("file.pmml")

# Run the algorithm using data from a DataTable,
# measuring the performance of each implementation.
pt1 = PerformanceTable()
pt2 = PerformanceTable()
model1.calculate(dataTable, performanceTable=pt1)
model2.calculate(dataTable, performanceTable=pt2)

# Performance results are reported at the level of
# PMML elements.
pt1.look(); pt2.look()
```

3.1.3 Example 3: Modifying the Behavior of PMML

As a third example, consider redefining the XSD schema of an existing PMML element. Since XSD defines the structure of a valid model, this would only alter which model files are considered valid and which are not. The <TransformationDictionary> element is intended for defining new functions and derived fields, but it would be advantageous for it to define models as well. In its class definition, we can replace

```
<xs:element ref="DerivedField"
   minOccurs="0" maxOccurs="unbounded"/>
```

with

```
<xs:choice minOccurs="0" maxOccurs="unbounded">
   <xs:element ref="DerivedField"/>
   <xs:group ref="MODEL-ELEMENT"/>
</xs:choice>
```

and now any model element (such as <TreeModel>, <ClusteringModel>, <AssociationModel>, etc.) can be used where a <DerivedField> would ordinarily be expected. Models input fields and output scores, so as long as the model provides a name for the scoring result, it may be associated with a new field. Removing this artificial distinction between derived fields and models vastly simplifies model-chaining:

```
<PMML version="4.1">
   <Header/>
   <DataDictionary>
      ...
   </DataDictionary>
   <TransformationDictionary>
      <TreeModel modelName="A">
         ...
      </TreeModel>
      <ClusteringModel modelName="B">
         ...
      </ClusteringModel>
      <AssociationModel modelName="C">
         ...
      </AssociationModel>
   </TransformationDictionary>
</PMML>
```

The leaf scores of the TreeModel become field "A," the closest clusters of the ClusteringModel become field "B," and the associated items of the AssociationModel become field "C." PMML 4.1 defines a mechanism for model chaining, but it involves setting up <Segments> and matching <OutputFields> of one model with the <MiningSchema> of the next, which is more cumbersome.

To produce the same effect without cluttering the namespace of derived fields, we could perform the XSD surgery on <LocalTransformation>, rather than <TransformationDictionary>. Then the following would be considered valid:

```
<PMML version="4.1">
   <Header/>
   <DataDictionary>
      ...
   </DataDictionary>
   <AssociationModel>
      <LocalTransformation>
         <ClusteringModel modelName="B">
            <LocalTransformation>
               <TreeModel modelName="A">
                  ...
               </TreeModel>
            </LocalTransformation>
            ...
         </ClusteringModel>
      </LocalTransformation>
      ...
   </AssociationModel>
</PMML>
```

All but the last step of the model chain is hidden inside of nested lexical scopes, so they would not appear as fields in the outermost scope. Furthermore, the last step is in the normal position for model elements, producing scores in the ordinary way.

3.2 Explicit Model Execution State

Most PMML calculations are stateless— that is, they depend only on the values in the current record, not on any previous records. However, the chiSquareIndependence, chiSquareDistribution, CUSUM, and scalarProduct test statistics of the <BaselineModel> and all functions of the <Aggregate> transformation are cumulative and not stateless. New PMML elements may be stateful as well— for instance, plotting tools embedded in a PMML-like language are also cumulative. If a PMML scoring engine is interrupted and re-started at another time or place, it may or may not be appropriate to re-start the calculation where it left off.

For maximum flexibility, Augustus encapulates all state-related information about the scoring process in a single key-value store. This store is serializable, so it can be saved, copied, transmitted to a remote site, or passed around a distributed service. For example, to continue a CUSUM (cumulative sum) analysis of a time series, one would:

```
import json
outputDataTable =
    cusumModel.calculate(inputDataTable)
json.dump(outputDataTable.state,
    open("intermediateFile.json"))
```

in the first session and

```
import json
inputDataTable.state =
    json.load(open("intermediateFile.json"))
outputDataTable =
    cusumModel.calculate(inputDataTable)
```

to continue it in the next.

Although the management of executable state is an implementation detail, outside the scope of the PMML specification, it would be useful if stateful fields had a "stateId" attribute to name their keys in the key-value store. Without explicit "stateIds," a scoring engine must rely on the location of an element within the file, such as a tuple of indexes that selects the element, starting from the root node. Position-based access is less reliable than an explicit name because annotating a PMML model (via <Extension> elements or standard non-executable elements) and producing-while-scoring both change the numbering of indexes. We will propose the new "stateId" attributes for an upcoming release of the PMML specification.

3.3 Column-wise Scoring

The declarative nature of PMML allows a problem to be solved in a different order than it is stated. Most models in the PMML specification describe operations on a record of data, which is a set of named fields in which each field has one numerical or categorical value. Usually, the PMML engine is applied to a sequence or records, with each record having the same set of field names but potentially different values. The simplest implementation would evaluate one record at a time, walking through the data structure of nested operations before applying them.

If we arrange the sequence of operations as rows of a table, this method performs each row of calculations before moving on to the next row. An alternative order would walk through the table the other way: perform all rows in a column before moving on to the next column. While both methods have the same time complexity and naively should take as much time to compute, the column-wise order is much faster for Numpy.

Numpy performs its operations in compiled, pre-typechecked machine code, so it does not suffer the performance issues of pure Python. However, it only provides a predetermined menu of operations, such as addition, multiplication, common mathematical functions, sorting, masking, partitioning,

and boolean logic. In the column-wise order, the Python code that interprets PMML and decides which operations to apply is executed once for the whole dataset, while the code that rapidly loops though the large dataset is executed in Numpy.

In a scaling test that compared Augustus 0.5 (which has row-wise order) and Augustus 0.6 (which has column-wise order), Augustus 0.5 took 97 μs per record in the asymptotic limit while Augustus 0.6 took 0.016 μs per record, a speed-up of roughly 6000. The column-wise implementation required about a million records for the numerical part of the calculation to become significant; for smaller datasets, the time is dominated by initialization (0.9 ms).

For similar reasons, we moved all XML processing to lxml, which uses pre-compiled routines for serialization, deserialization, recursive searching, and XSD- and XSLT-based validation. Comparing Augustus 0.5 (pure-Python XML handling) with Augustus 0.6 (lxml), reading-with-validation improved by a factor of 22, reading-without-validation by a factor of 59, and writing improved by a factor of 20. Augustus 0.6 also serializes models to and from JSON, with serialization speeds comparable to XML only if pre-compiled, third-party JSON serializers are used.

3.4 MapReduce-Ready Algorithms

As a Python module, Augustus can be used anywhere Python is used, including Hadoop Streaming[9]. However, the MapReduce paradigm puts some constraints on algorithms: a process running on a mapper has no access to data on other mappers, and a process running on a reducer has no guaranteed access to data associated with other keys. Algorithms that do not need a broad view of the dataset ("embarrassingly parallel") are easy to integrate into MapReduce, and all stateless PMML models are in this category. Stateful models can be integrated into MapReduce by passing a state object between processes.

This issue is more significant for algorithms that produce models. A linear regression, for instance, derives fit parameters from all points in the dataset. If the points are too numerous to fit in the memory of a single computer, then the algorithm must be split into a function that can summarize a part of the dataset (the mapper) and a part that can combine the summarized parts into a result (the reducer). In the case of a linear fit, the calculation can be decomposed into partial sums: each mapper computes a partial sum from its subset of the data and the reducer combines partial sums and derives fit parameters.

To facilitate the development of algorithms that can be run in MapReduce, Augustus has a framework that simulates a MapReduce environment. This framework hides parts of the dataset that would not be available in a MapReduce job and can optionally run them in separate threads, taking advantage of multiple CPUs if available. This framework can also submit the same algorithms, without modification, to Hadoop by marshaling their execution state as Python bytecode, serializing the data as SequenceFiles, and launching it as a Hadoop Streaming job.

Implementing an algorithm in this framework allows it to be distributed when necessary but not suffer a performance penalty when executed on a single machine. Currently, all Augustus model-producer algorithms are implemented in the MapReduce-ready framework. Segmentation is the most natural fit to this paradigm, since the mappers simply associate segment predicates with keys and the reducers build the independent models for each key in isolation. However, non-trivial examples like distributed k-means clustering have also been implemented.

4. LESSONS LEARNED ABOUT PMML

Reimplementing a PMML engine from the ground up gave us new perspective on PMML as a language. In most cases, the ease of implementation depended on the unity of its concepts. Aspects that are well-unified across all model types, such as lexical field scopes, transformations, functions, and model verification, were easy to implement. Other aspects are not as well unified, which we present below.

4.1 PMML's Type System

Perhaps the most significant issue is PMML's system of data types. PMML is strongly typed, specified by dataType (e.g. "integer" or "string") and optype (e.g. "continuous" or "categorical"). However, some constructs yield types that are not expressible in this way. For instance, the <Aggregate> transformation can produce structured data with

```
<Aggregate field="x" function="multiset"/>
```

and

```
<Aggregate field="x" function="count"
   groupField="y"/>
```

The first of these could be represented by a sequence type and the latter by a mapping, but these are not valid types in PMML. Furthermore, expressions must be wrapped in a <DerivedField> to be used, and <DerivedFields> require the user to give the field a dataType:

```
<DerivedField name="z" dataType="???" optype="???">
   <Aggregate field="x" function="count"
      groupField="y"/>
</DerivedField>
```

It is not clear how multiset or groupField aggregations could ever be used in a valid PMML document. If we allow model elements to produce new derived fields as described above, the same could be said of segmentation with a multipleModelMethod of "selectAll."

Another problem with PMML's type system is that new types are defined as part of the declaration of a variable. For instance, consider the following two fields:

```
<DataField name="x" dataType="string"
                                optype="ordinal">
   <Value value="first"/>
   <Value value="second"/>
   <Value value="third"/>
</DataField>
<DataField name="y" dataType="string"
                                optype="ordinal">
   <Value value="first"/>
   <Value value="second"/>
   <Value value="third"/>
</DataField>
```

It would seem that "x" and "y" belong to the same type—a 3-element space in which "first" preceeds "second," which preceeds "third." However, unless a special rule were introduced concerning ordinal strings with identical enumerations, there would be no way to infer that. This makes it difficult to say whether comparisons like

```
<Apply function="lessThan">
   <FieldRef field="x"/>
   <FieldRef field="y"/>
</Apply>
```

are legal without introducing many new rules. To make this example more acute, should this comparison be allowed if "y" didn't have a "first?" The same could be said of types involving intervals of validity. In most programming languages, types are defined separately from variables, which would resolve problems like the above: "x" and "y" would either be declared as instances of the same type or as two different types with the same display values.

Data type annotations are required for <DataFields> and <DerivedFields>, but not <DefineFunction>. Constants require a dataType but not an optype, which makes it difficult to say if

```
<Apply function="lessThan">
   <FieldRef field="x"/>
   <Constant dataType="string">fourth</Constant>
</Apply>
```

should be allowed (using "x" from the example above).

In most expressions, types can be inferred to allow a static type check of the PMML document. However, there are a few exceptions that make a static type check impossible in general. <MapValues>, for example, introduces strings from an XML table. The validity of an expression like

```
<Apply function="lessThan">
   <FieldRef field="x"/>
   <MapValues> ... </MapValues>
</Apply>
```

depends on the assumed dataType and optype of the values in the table.

Ideally, PMML's type system should be clarified in two ways:

1. types must be defined separately from fields;

2. all expressions and models should have type signatures that indicate the output type for a given combination of input types.

The latter would allow for a systematic, static type check, greatly enhancing PMML's safety for production use. Since all types could be inferred from input fields and constants, it would not be necessary to declare types on derived fields and user-defined functions. These type annotations could instead serve as type assertions whose failure could be caught before deploying the PMML document. We understand, however, that assigning type signatures to every computable function in PMML would be a large task.

4.2 Other issues

The other issues are relatively minor:

(a) It is difficult to tell, in general, which elements and attributes affect a model's score and which are for information only.

(b) While <MiningSchemae> are useful for describing how a model is produced (e.g. which fields are dependent and which are independent), they do not affect a model's score, other than mapping MISSING and INVALID values and outliers. Those functions would be more useful if they became independent transformations.

(c) The element has been re-used for purposes well beyond its semantic meaning, e.g. random forests and model chaining. This makes it more difficult to use them in a partitioned space (the original meaning of a).

(d) There are many different ways for results to exit a model: derived fields in the top-most lexical scope, model scores (and auxiliary scores like "entity affinity"), and as <OutputFields>. Moreover, the rules that map quantities onto <OutputFields> are very complex.

5. SUMMARY

This paper describes a new version of Augustus that implements PMML in a hot-swappable way. This feature can be helpful for defining new PMML elements or performing side-by-side comparisons of PMML implementations. It also consolidates the execution state of the scoring engine into a single object that can be serialized and arranges the computation in a column-wise way to take advantage of Numpy optimizations. Algorithms to produce PMML models were designed with MapReduce in mind.

Developing a new PMML implementation gave us new perspective on PMML as a language. Many features of the language work well together, but its system of describing data types is hard to implement and use consistently.

References

[1] John Chaves, Chris Curry, Robert L. Grossman, David Locke, and Steve Vejcik. Augustus: the design and architecture of a PMML-based scoring engine. In *Proceedings of the 4th international workshop on Data mining standards, services and platforms*, DMSSP '06, pages 38–46, New York, NY, USA, 2006. ACM.

[2] Guido van Rossum et al. The python language. `http://www.python.org`. Accessed: 2013-06-03.

[3] The Data Mining Group. Predictive Model Markup Language (PMML). `http://www.dmg.org`. Accessed: 2013-06-03.

[4] Travis Oliphant et al. NumPy: Open source numerical array processing for Python. `http://www.numpy.org`. Accessed: 2013-06-03.

[5] Martijn Faassen et al. lxml: Open source XML processing for Python. `http://lxml.de`. Accessed: 2013-06-03.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems*

Design & Implementation - Volume 6, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[7] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[8] John Carmack. Parallel implementations. `http://www.altdevblogaday.com/2011/11/22/parallel-implementations`, November 2011. Accessed: 2013-06-03.

[9] Hadoop streaming. `http://hadoop.apache.org/docs/stable/streaming.html`. Accessed: 2013-06-03.