Deploying Analytics with the Portable Format for Analytics (PFA)

Jim Pivarski
Open Data Group Inc.
400 Lathrop Ave Suite 90
River Forest IL USA

Collin Bennett
Open Data Group Inc.
400 Lathrop Ave Suite 90
River Forest IL USA

Robert L. Grossman[*]
Open Data Group Inc.
400 Lathrop Ave Suite 90
River Forest IL USA

ABSTRACT

We introduce a new language for deploying analytic models into products, services and operational systems called the Portable Format for Analytics (PFA). PFA is an example of what is sometimes called a model interchange format, a language for describing analytic models that is independent of specific tools, applications or systems. Model interchange formats allow one application (the model producer) to export models and another application (the model consumer or scoring engine) to import models. The core idea behind PFA is to support the safe execution of statistical functions, mathematical functions, and machine learning algorithms and their compositions within a safe execution environment. With this approach, the common analytic models used in data science can be implemented, as well as the data transformations and data aggregations required for pre- and post-processing data. PFA compliant scoring engines can be extended by adding new user defined functions described in PFA. We describe the design of PFA. A Data Mining Group (DMG) Working Group is developing the PFA standard. The current version is 0.8.1 and contains many of the commonly used statistical and machine learning models, including regression, clustering, support vector machines, neural networks, etc. We also describe two implementations of Hadrian, one in Scala and one in Python. We discuss four case studies that use PFA and Hadrian to specify analytic models, including two that are deployed in operations at client sites.

Keywords

model producers, scoring engines, Portable Format for Analytics, PFA, PMML, deploying analytics

[*]RLG is also a faculty member at the University of Chicago.

KDD '16 August 13-17, 2016, San Francisco, CA, USA
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4232-2/16/08.
DOI: http://dx.doi.org/10.1145/2939672.2939731

1. INTRODUCTION

The KDD research community has traditionally been focused on developing new algorithms and developing systems for managing and analyzing data. This is the one of the cores of data science. Technology is also required for deploying algorithms and statistical models in analytic products, analytic services, and operational systems. Sometimes the term *analytic operations* is used for this component of the KDD process.

A problem that has haunted the KDD community for the last twenty years is how to efficiently deploy the analytic models developed by data scientists into products and services that must live in operational environments, which usually have stringent service level requirements. Because of these requirements, most organizations do not allow code to be integrated into operations without careful and extensive testing. See Figure 1.

In this paper, we introduce a new language called the Portable Format for Analytics (PFA), describe some of the ways that PFA differs from the Predictive Model Markup Language (PMML), the current dominant standard for describing analytic models [1], and discuss some of the lessons that we have learned working with PFA over the past two years in various deployed analytic applications.

Contributions of this Paper This paper makes four main contributions to the KDD community. The first contribution is that we introduce a language (PFA) for describing statistical and data mining models that overcomes some of the important limitations of PMML. The second contribution is that we have developed and deployed *two* implementations of PFA-compliant scoring engines (one in Scala and one in Python), which are available to KDD community for personal and research use. The third contribution is that we have described two deployments into production sites of the Scala scoring engine and discussed some of the lessons learned from these deployments. The fourth contribution is calling attention to PFA and to the Data Mining Group (DMG) PFA Working Group so that others in the KDD research community can contribute to the further development of the PFA standard.

2. ARCHITECTURES FOR DEPLOYING ANALYTICS

An analytic architecture was developed over fifteen years ago that separated systems, applications and tools that produced analytic models from those that consumed analytic models [2, 3]. Model producers are part of the development

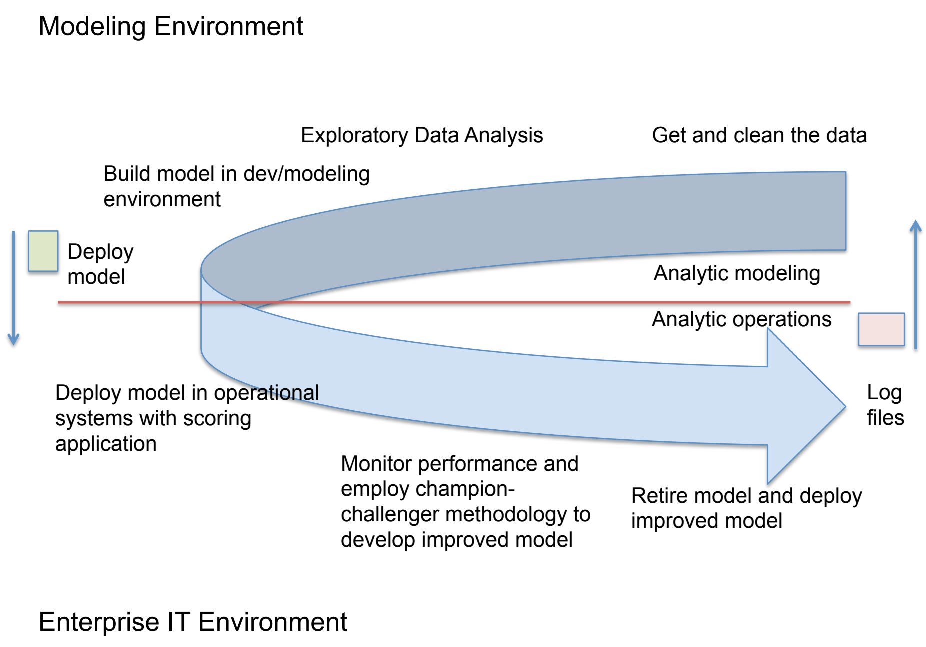

Modeling Environment
Exploratory Data Analysis
Get and clean the data
Build model in dev/modeling environment
Deploy model
Analytic modeling
Analytic operations
Deploy model in operational systems with scoring application
Log files
Monitor performance and employ champion-challenger methodology to develop improved model
Retire model and deploy improved model
Enterprise IT Environment

Figure 1: Analytic model producers and analytic model consumers (scoring engines).

or data modeling environment, while model consumers are part of the operational environment. See Figure 1.

Model producers export models in a language for analytic models (sometimes called a model interchange format), while model consumers import analytic models. With this approach, multiple systems, applications and tools can be used by data scientists for exporting models, and a single scoring engine can consume all of them in an operational environment.

Over time, the term *scoring engine* began to be used as a synonym for model consumer. As shown in Figure 3, the scoring engine is usually embedded in deployment environment that includes the inputs and outputs of the scoring engine. For example, scoring engines may be integrated with web applications and services, with databases, with enterprise service buses, with distributed data processing systems, with real time distributed messaging systems, etc.

It is important to note that with this architecture, the analytic models that are described in the model interchange format are designed to be written and read by programs, not by humans. In other words, models in the model interchange format are designed to be machine readable, not human readable.

3. LANGUAGES FOR DEPLOYING ANALYTICS

The model interchange format that has gained the most adoption is the Predictive Model Markup Language (PMML) [3], which is an XML-based markup language for describing statistical and data mining models. The PMML standard [1, 4] supports the models most commonly used by data scientists, including regression, clustering, support vector machines, neural networks, etc.

PMML represented a significant step forward in terms of providing a mechanism for different systems, applications and tools to exchange analytic models. On the other hand, over time limitations of PMML became apparent. Perhaps, the most common was that PMML was not extensible and did not support all the code required to deploy analytic models, since it has relatively limited functionality for the pre- and post-processing required by most deployed applications, and many deployed applications did not use a single PMML model, but rather workflows built from multiple PMML models.

PMML is a *specification* for a model interchange format. Once the PMML Working Group approves a specification for a new PMML model, PMML-compliant scoring engines are updated by their vendors or developers to add support for that model. Although the PMML Working Group works hard on the standard, overtime the specification grew more and more complicated, raising the work required to develop a PMML compliant scoring engine.

The goals for PFA were to define a language for analytic models with the following properties:

1. The language should be extensible so that users and system developers can define their models and pre- and post-processing code.

2. The language should allow models to be composed so that chains of models and hierarchical models can be supported and allow multiple models to be combined into workflows.

3. The language should be easy to integrate into today's distributed and event-based data processing platforms, such as Hadoop [5], Spark and Storm [6].

4. The language should be "safe" to deploy in IT operational environments.

As mentioned, PMML struggled with the first three requirements. The approach taken with PFA was to define a language in which JSON-based functions could be safely composed and the functions were rich enough to define the common analytic models, as well as the common pre- and post- processing used by data scientists. The language and standard support user defined functions, which supports Requirements 1 and 2. Since the language is JSON-based and uses Avro for data serialization, it integrates easily into today's distributed and event based data processing platforms (Requirement 3).

Although PFA is essentially a programming language, it is limited in ways that make it safer to use than conventional programming languages (Requirement 4). A scoring engine implemented in C, Python, or Java could access the underlying file system, operating system, or network, but a scoring engine implemented in PFA can only transform the data that it is given. Therefore, the security issues involved in running arbitrary code are not raised by PFA.

As we will see below, PFA supports functions for data transformations, data aggregations, data filtering, as well as the basic mathematical and statistical functions. Since these core functions ("data mining or analytic primitives") can be composed, the models in the PMML standard can be easily defined [7]. Specifically, the current version of PFA contains all of the analytic models in the current version of PMML [7]. In addition, compositions of the analytic primitives can be used so that any pre-processing or post-processing code can be expressed in PFA.

PFA is a model interchange format based upon *analytic primitives.* A developer or vendor can add support for a new mathematical or statistical function, model, data transformation or data aggregation at any time simply by expressing it in the PFA language. The more common models can be adapted as a standard by the PFA Working Group. This approach makes adding new functionality to PFA-compliant scoring engines relatively easy and reduces to work required to develop PFA-compliant scoring engines.

To summarize, the main benefit provided by scoring engines is that models can be quickly added to operational environments. With a PMML-compliant scoring engine, any model that is specified in the standard can be quickly added to operational environments. With PFA-compliant scoring engines, any model that can be described in the PFA language can be quickly added to operational environments.

4. OVERVIEW OF PFA

The Portable Format for Analytics (PFA) is a domain-specific language that describes scoring engines. A scoring engine is a high-throughput data processor, usually generated by a statistical analysis. For instance, an analysis might describe a dataset using decision trees; its scoring engine is an algorithm that walks down each tree, classifying data. Another analysis might use k-means to summarize a dataset as a code book of cluster centers; its scoring engine would associate new data to their closest clusters. PFA provides a uniform way to implement and transport these scoring engines.

PFA is portable because it is a simple language whose syntax is a strict subset of JSON. JSON is a widely-used serialization format consisting of only four primitives (string, number, boolean, and null) and two container types (array and string-based map). PFA data transformations and models are all implemented as a syntax tree of nested JSON objects. Any program or library that can manipulate JSON can manipulate PFA. Thus, one only needs to add JSON output to existing data analysis software for it to generate PFA-based scoring engines.

The smallest possible PFA document is:
`{"input": "null", "output": "null", "action": null}`, which inputs nothing, outputs nothing, and does nothing. But, this presents the three necessary parts of a PFA Engine: an input, output, and action. Input and output define the data into and out of the PFA engine. This can be a type, like `double` or a schema defining a structure. The action routine is called once for every input datum. The action can be any PFA primitive or user-defined function. For example, here is a trivial model

```
input: double
output: double
action:
  - {m.round: {"*": [{m.sin:
      {+: [input, 1]}}, 2]}}
```

This PFA engine takes a double as input, adds 1 to that, multiples the sine of that addition by 2 and returns the closest whole number as a double.

Simple engines, like the mathematical function above, transform one input into one output. In practice, you sometimes need to filter data (one input to zero or one outputs) or aggregate data (an entire dataset to one output). To handle these cases, PFA has three methods: `map`, `emit`, and `fold`, which are described later in this section.

Also, PFA has a centralized concept of state that can be shared among multiple instances of a scoring engine. These scoring engines can be stopped, resumed, rolled back to a previous state, or analyzed offline in ways that would be impossible for arbitrary code written in a conventional language. A PFA-based scoring engine is essentially a lightweight, human-readable virtual machine capable only of transforming data.

Persistent storage in PFA is handled by cells and pools. Cells are global variables that cannot be created or destroyed at run-time, only reassigned. Pools are collections of key-value pairs that can be created and destroyed at run-time. This is similar to environments in R. For a pool, the granularity of concurrent access is at the level of a single pool item.

Cells and pools are both specified as JSON objects with the same fields, but initialization is only required for cells.

Extensibility. With PMML, when a function not in the specification is needed, people write a custom extension in their PMML-application and submit to the PMML Working Group the new desired functionality as a Request for Comments. This is a reasonable balancing of a developer's immediate needs and the community's goal of moving from one stable release of the specification to the next. The draw back is that in the interim, the model is not valid PMML and cannot be used in another PMML application.

To achieve PFA's first goal of extensibility and to remove this obstacle to vendor portability, PFA is made of up analytic primitives. Its functions are designed as "building blocks" and as with a general programming language, user-defined functions can be written in PFA, that are themselves valid PFA, and used across any PFA application.

For example, in the PFA time module, there is no function that returns the UTC offset when given a timestamp and timezone. This functionality was requested by a client and will be submitted to the PFA Working Group for consideration, but in the mean time, we wrote the user-defined function in Figure 2 that can run inside any PFA application.

Post-processing. Extensibility can also be considered in terms of pre- and post-processing. PFA has `before` and `end` that allows for hooks to be written that are outside of the scoring process. Sometimes, there is a need to post-process the scored event or its score.

Within a PFA scoring engine, post-processing can be performed by including logic in the PFA action. As a simple example, consider the post-processing logic of only wanting:

1. To alert when at least three scores over some threshold have been seen in a 1 minute time window; and
2. Not alert for a given entity more than once in a 24-hour period.

This can be accomplished by defining a record and pool as follows:

```
types:
 record(Counter,
        event_counter: int,
        alert_counter: int,
        first_timestamp: double,
        last_alert_time: double);

pools:
 window(type: Counter,
        shared: true) = {};
```

and including calls to a check and update in the action, rather than just emitting a score:

```
action:
  window[id] to fcn(old: Counter -> Counter) {
    if u.checConditionMet(old) {
      // send alert and update counter
    } else {
      //update counter
      new(Counter,
          event_counter: 1,
          alert_counter: old["alert_counter"] + 1,
          first_timestamp: ts,
          last_alert_time: old["last_alert_time"]);
    }
```

Composing PFA functions. As mentioned above, an important goal of PFA is supporting the composition of any PFA functions. This means that, unlike PMML where compositions are restricted to certain scopes and between certain components, PFA primitives, functions, and user-defined functions can be composed where ever it makes sense. For example, if you have a decision tree defined by:

```
types:
  record(TreeNode,
         field: enum([x,y, z]),
         operator: string,
         value: double,
         pass: union(TreeNode,string),
         fail: union(TreeNode,string))
```

You can add a regression at each leaf node by defining the regression and embedding it in the leaves:

```
types:
   record(Regression,
          const: double,
          coeff: array(double));

   record(TreeNode,
         field: enum([x,y, z]),
         operator: string,
         value: double,
         pass: union(TreeNode,Regression),
         fail: union(TreeNode,Regression))
```

The models are composed, so the action becomes a 2-step process. Instead of returning a label from the tree walk, the leaf is now a regression that needs to be scored:

```
action:
  var leaf = model.tree.simpleWalk(input, tree,
             fcn(d: Datum, t: TreeNode -> boolean)
               model.tree.simpleTest(d,t));
  leaf
```

becomes:

```
action:
  var leaf = model.tree.simpleWalk(input, tree,
             fcn(d: Datum, t: TreeNode -> boolean)
               model.tree.simpleTest(d,t));

  var vector = new(array(double), input.x,
             input.y, input.z);
  model.reg.linear(vector, leaf)
```

For completeness, in the decision tree PFA, the regression needs to be defined at each TreeNode:

```
  {TreeNode:
     {field: y,
      operator: "<",
      value: 2.2,
      pass: {string: "pass-pass"},
      fail: {string: "pass-fail"}}},
```

becomes, for example,

```
  {TreeNode:
     {field: y,
      operator: "<",
      value: 2.2,
      pass: {Regression:
         {const: 0.0,
          coeff: [0.1,0.2, 0.3]}}}}
      fail: {Regression:
         {const: -1.0,
          coeff: [0.1,-0.2, 0.0]}}}}
```

PFA outputs. PFA views data as a possibly infinite stream of inputs, and produces a corresponding stream of outputs. The author of a PFA document may define a `begin` method, which is evaluated before seeing any data. This author must define an `action` method, which is evaluated on every input, and may define an `end` method, which is evaluated after all the data, if any such time exists. PFA is a stream processor, like the UNIX command `awk`.

Most scoring engines produce one output for every input, like the `map` or `lapply` functor found in array processing languages. Some scoring engines, however, filter the data,

```
fcns:
  getUTCOffset = fcn(x: double, zone: string -> double) {
    // get minuteOfDay in the time zone
    var hourOfDay_zone = time.hourOfDay(x, zone);
    var minuteOfHour_zone = time.minuteOfHour(x, zone);
    var minuteOfDay_zone = hourOfDay_zone * 60 + minuteOfHour_zone;

    // get minuteOfDay in UTC
    var hourOfDay_UTC = time.hourOfDay(x, "");
    var minuteOfHour_UTC = time.minuteOfHour(x, "");
    var minuteOfDay_UTC = hourOfDay_UTC * 60 + minuteOfHour_UTC;

    // difference modulo the number of minutes in a day
    var minuteDiff = (minuteOfDay_zone - minuteOfDay_UTC) % (24 * 60);

    // difference in hours
    var hourDiff = minuteDiff / 60.0;

    // cycled to a number in the range (-12, 12]
    if (hourDiff > 12.0)
      hourDiff - 24.0
    else
      hourDiff
  }
```

Figure 2: A user defined PFA function to supplement the PFA time module.

producing zero or one outputs for each input, while others expand tables, potentially producing more than one output per input (such as Hadoop mappers). Still other scoring engines reduce a data stream to a single value or structure, such as `reduce` functors or `GROUP BY` in SQL. These cases cover all three types of functions in SQL (transformers, table generating functions, and aggregations).

PFA provides for the three cases with a top-level field called `method`, which may be `"map"`, `"emit"`, or `"fold"`. In `"map"` mode, the `action` is an expression that evaluates to the output. In `"emit"` mode, the `action` may call an `emit` function zero or more times to produce outputs. In `"fold"` mode, the `action` returns a partial aggregation for each input and also updates a `tally` that can be used in later steps of the aggregation. To reduce a dataset, one could ignore all outputs except the last. The `"fold"` mode also requires a `merge` method that combines partial aggregations from independent processors, allowing the aggregation to be scaled out, and a `zero` value for starting the aggregation.

5. PFA LANGUAGE DETAILS

A PFA-based scoring engine is a JSON object with fields for general information like model name, version, and metadata, and a single execution path of expressions to be applied to the input data. These expressions consist of constants, such as numbers, data field references, which are JSON strings, and function calls, which are JSON objects with the following structure: `{"functionName": [arguments]}`. Basic features of the langage are special forms with multiple keys: `{"if": predicate, "then": consequent}`.

PFA is a statically typed language with immutable data structures and restricted function objects. This strictness makes it possible to analyze a scoring engine before it is executed. For instance, a semantically valid PFA scoring engine can never pass a string where a number is expected, can never introduce long-range dependencies between data structures, and can modify shared state without the possibility of deadlock.

5.1 Static typing

The PFA type system is inherited from Avro, a serialization format that imposes structure on JSON by requiring explicitly typed number formats (integer, long, single-precision, and double-precision floating point numbers), a distinction between Unicode strings and raw byte arrays, homogeneous arrays and maps, heterogeneous records, enumeration types, and tagged unions of any of the above. An Avro type is specified as a JSON object, with schemae like `{"type": "array", "items": "int"}` to specify an array of integers, for instance.

Avro schemae are included within PFA as type declarations. Input data, function parameters, and global state require explicit types, declared with Avro schemae, and PFA infers the types of all other expressions and checks them against the declared output types.

PFA can downcast types to allow, for instance, a variable declared as a union of null and integer to be passed to a function that expects an integer, but it always splits the program flow into cases— one path is followed if the variable is null and another is followed if it is an integer. Therefore, PFA can use null to signify missing data without ever encountering a null pointer exception at runtime.

Static typing also provides a performance boost: PFA can be compiled into bytecode that does not need to check types at runtime. Only the structure of the input data must be checked; the rest is guaranteed.

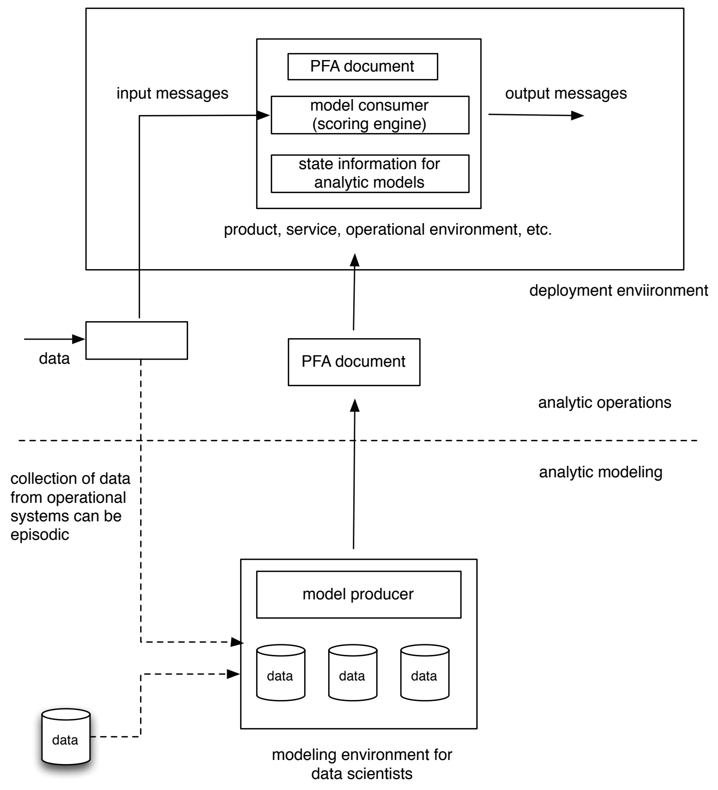

PFA document
input messages
model consumer (scoring engine)
output messages
state information for analytic models
product, service, operational environment, etc.
deployment enviironment
data
PFA document
analytic operations
analytic modeling
collection of data from operational systems can be episodic
model producer
data
data
data
data
modeling environment for data scientists

Figure 3: Analytic model producers and analytic model consumers (scoring engines) exchanging a PFA model.

5.2 Immutable data

Immutable data structures can be replaced but not modified in-place. For instance, items in an immutable array cannot be changed, but the whole array can be replaced with one that has different elements. Immutable data structures have several conceptual advantages: there is no distinction between references and values, so two data structures cannot be linked in such a way that modifications of one implicity change the other, and data can be passed from one scoring engine to another without needing to defensively copy it.

Immutability is particularly useful for sharing data among many scoring engines running in parallel. PFA defines copy-on-write semantics for modifications of shared engine state that allows non-blocking, consistent reads of the shared data while one of the scoring engines is processing a transaction to replace it. Since there is no distinction between references and values, PFA implementations can re-use parts of a data structure among different versions of that structure to avoid deep copying and increase both memory and computational performance.

5.3 Function objects

The ability to define functions and pass them as objects is an incredibly useful aspect of many high-level languages. They allow for generic functional programming, such as transforming one structure into another structure by applying a function to each element (map) or summarizing a structure by iteratively applying an aggregation function (reduce). They also allow the programmer to specify the beginning and ending of a transaction on shared data by wrapping it in a function, perhaps closing over local variables.

However, if functions are freely passed as objects, then it can become impossible to statically determine which functions can be executed in a given context. This is particularly problematic when functions are used as transactions on shared data, since this can lead to deadlock— two scoring engines might end up waiting for each other when attempting to process mutually dependent transactions. Therefore, PFA does not allow unrestricted function objects, but only enough functionality to achieve the benefits of functional programming and shared data transactions without the disadvantages.

Functions can be expressed as inline arguments to functors such as map and reduce, and they can be named as globally accessible user-defined functions that are referenced in arguments to functors. Shared data can be modified by passing it a function that transforms the old value into a new value, but these transactions are not allowed to modify any other shared data.

Since function references can only appear in argument lists, a PFA validator can exhaustively determine which functions can ever be called from a function, thereby ensuring that a transaction can never call anything that, through a chain of other function calls, attempts another transaction. These functions are allowed to read any thread-local variables (as closures) and read any shared global data because PFA's concurrency semantics allow reading without blocking.

With these restrictions, functions do not need to be implemented as objects in compiled PFA, which allows for another performance boost: all user-defined functions could be implemented as inlined bytecode.

6. PFA FOR MODEL DEVELOPMENT

Unlike a conventional program, a scoring engine typically implements a simple or well-known algorithm with a large store of parameter data. For example, a random forest may include megabytes or gigabytes of tree parameters, but the scoring procedure simply walks down each tree and reports the average or most common result. A PFA document wraps up the algorithm and the parameter data into one JSON object, which eliminates the bookkeeping involved in coordinating programs with their parameters.

Usually, scoring engines are produced by other programs—for instance, a library package in R derives a random forest and creates a function that can be used in R to score new data. The only problem is that this scoring engine cannot be used outside of the R environment. However, by iterating over the R function's internal parameters, we can convert it to JSON and insert that JSON into a PFA document.

Even though the algorithmic part of a PFA document tends to be a short list of library function calls, it can be difficult to write an algorithm purely out of function calls. Programmers want to write `x + y` rather than `{"+": ["x", "y"]}`. For this reason, there are several converters from conventional programming languages such as Python and Javascript into PFA, but the most fully featured is PrettyPFA, which provides a C-like syntax for all PFA features. In PrettyPFA, the random forest algorithm is

```
input:
  record(Datum, inputPlaceholder: double)

output: Score

types:
  enum([scorePlaceholder], Score);

  record(TreeNode,
         field: enum([inputPlaceholder]),
         operator: string,
         value: double,
         pass: union(Score, TreeNode),
         fail: union(Score, TreeNode))

cells:
  // declare an empty forest, to be filled in by R
  forest(type: array(TreeNode), shared: true) = []

action:
  // apply a tree-scoring function to each tree
  var scores = a.map(forest, u.score(x: input));

  // output the majority vote
  a.mode(scores)

fcns:
  // tree-scoring composes simpleWalk and simpleTest
  score = fcn(x: Datum, tree: TreeNode -> Score)
    model.tree.simpleWalk(x, tree,
      fcn(d: Datum, t: TreeNode -> boolean)
        model.tree.simpleTest(d, t))
```

The `input` and `output` sections declare data fields and the `TreeNode` type declares the structure of a tree. This scoring engine has one globally shared quantity (a `cell`), the array of trees that comprise the random forest. Its action on input data consists of three function calls:

- `a.map` maps a user-defined tree-scoring function (`u.score`) to each tree in the forest, producing an array of each tree's scores;
- `a.mode` computes the mode of this distribution (a majority vote).

The `u.score` function calls two library functions, `simpleWalk` and `simpleTest` (fully qualified), which together describe the global process of walking over the tree and the local process of deciding how to step from node to node. Different choices of library functions or even user-defined functions can be used to augment this process.

Once written, the PrettyPFA is transformed into a JSON object that is more easily manipulated by a script. The above example translates into the PFA in Figure 6. In this form, the forest produced by R can be inserted into the tree by reassigning

```
pfa["cells"]["forest"]["init"] = myForest
```

and changing the `inputPlaceholder` and `scorePlaceholder` field names. We have developed many tools for manipulating JSON, including regular expression patterns that can map

```
{some: ([...]), thing: (# < 12)}
```

(a JSON object with keys `"some"` and `"thing"`, the first pointing to an array and the second to a number under `12`) into

```
{another: (1), thing: (2)}
```

(a new JSON object with keys `"another"`, `"thing"` and the same values). These patterns transform PFA structures anywhere in a PFA document, and are thus insensitive to modifications upstream.

7. PFA IMPLEMENTATIONS

We have implemented PFA in two environments: Python and Scala (for the Java Virtual Machine). These two libraries implement version 0.8.1 of the PFA specification [7]. The two implementations agree as determined by the PFA conformance tests that are available from the PFA Working Group website [7].

The Python implementation, called Titus, is more often used for model development, so it includes producers such as regression, k-means clustering, and CART trees, converters such as PMML trees to PFA, Python to PFA, and PrettyPFA, and tools for manipulating structure, such as JSON regular expressions and a command-line driven tool called the PFA Inspector.

The Scala implementation, called Hadrian, is more often used for model deployment, so it compiles PFA documents into Java bytecode on-the-fly (without requiring disk access) and is embedded in many systems. To date, Hadrian has been embedded as a command-line tool that transforms data from standard input to standard output (Hadrian-Standalone), a scoring engine container that builds a directed acyclic

```
{"input": {"fields": [{"type": "double", "name": "inputPlaceholder"}], "type": "record", "name": "Datum"}, "output": {"symbols":
["scorePlaceholder"], "type": "enum", "name": "Score"}, "cells": {"forest": {"type": {"items": "TreeNode", "type": "array"}, "init": [],
"shared": true, "rollback": false}}, "action": [{"let": {"scores": {"a.map": [{"cell": "forest"}, {"fcn": "u.score", "fill": {"x":
"input"}}]}}}, {"a.mode": ["scores"]}], "fcns": {"score": {"params": [{"x": "Datum"}, {"tree": {"fields": [{"type": {"symbols":
["inputPlaceholder"], "type": "enum", "name": "Enum_1"}, "name": "field"}, {"type": "string", "name": "operator"}, {"type": "double",
"name": "value"}, {"type": ["Score", "TreeNode"], "name": "pass"}, {"type": ["Score", "TreeNode"], "name": "fail"}], "type": "record",
"name": "TreeNode"}}], "ret": "Score", "do": [{"model.tree.simpleWalk": ["x", "tree", {"params": [{"d": "Datum"}, {"t": "TreeNode"}],
"ret": "boolean", "do": [{"model.tree.simpleTest": ["d", "t"]}]}]}]}}}
```

Figure 4: The PFA for the random forest example.

topology of interacting scoring engines (Hadrian-Actors), a servlet that can be used in Google App Engine or a generic servlet container like Tomcat (Hadrian-GAE), and a map-reduce processor that can score or train models in Hadoop (Hadrian-MR). Any big-data tool that runs on the Java Virtual Machine can be wired into Hadrian.

There is also a version for R, called Aurelius, that is a toolkit for generating PFA in the R programming language. It focuses on porting models to PFA from their R equivalents. However, to validate or execute a execute scoring engines, Aurelius sends them to Titus through rPython (a 3rd party library).

The choice of JSON model serialization allows for easy integration into other big-data toolsets. For instance, the MongoDB database system stores data as JSON-like objects and permits deep inspection of those objects. Titus and Hadrian both have a snapshot feature that serializes the state of a running scoring engine as a new PFA document that could resume processing from the frozen state— MongoDB could be used to store an ensemble of scoring engine states and inspect their contents.

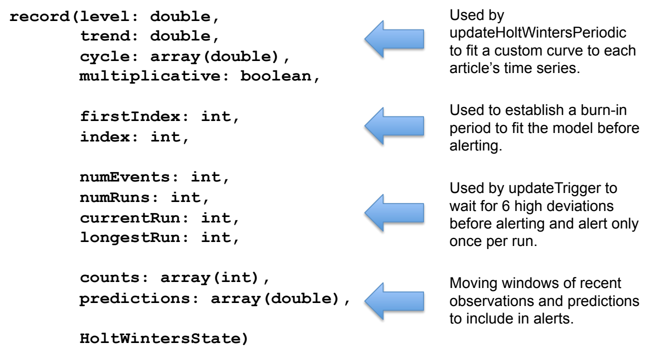
Similarly, the use of Avro as a type system smooths the way to integration with big-data tools. Avro is a popular serialization format in the Hadoop ecosystem, so data transfer to and from these tools is easiest with Avro and possible for any format that can be mapped to Avro. For instance, a CSV file or the tuples passed through a Storm topology could be viewed as a series of flat, heterogeneous Avro records, and the data structures manipulated by Hive SQL are a perfect conceptual match to Avro's nested data types, even though their byte-serializations differ. With a small transformation before and after the scoring engine call, Hadrian could be inserted into a Storm topology or used to write Hive user-defined functions.

Hadrian has been used in high-throughput environments in which hundreds of thousands of records are processed per second. In particular, Hadrian-Actors has been deployed on a 32 CPU machine to perform cluster-based classification and anomaly detection using 4 scoring engines (4 PFA files), distributed as 32 instances, listening to 8 high-speed data pipelines. Map-reduce jobs involving Hadrian-Hadoop and Titus have been used to build models on a MapR cluster, passing data as Avro and JSON.

8. CASE STUDIES

8.1 Case Study 1: Tracking Topical Interest in Proteins Using Wikipedia

In the first case study, we describe a PFA topic tracking model. This model tracked hourly page view statistics (December 2007 through August 2015) for all Wikipedia pages in the category "Human Proteins." There were 11,273 such



```
record(level: double,
       trend: double,
       cycle: array(double),
       multiplicative: boolean,

       firstIndex: int,
       index: int,

       numEvents: int,
       numRuns: int,
       currentRun: int,
       longestRun: int,

       counts: array(int),
       predictions: array(double),

       HoltWintersState)
```

Used by updateHoltWintersPeriodic to fit a custom curve to each article's time series.
Used to establish a burn-in period to fit the model before alerting.
Used by updateTrigger to wait for 6 high deviations before alerting and alert only once per run.
Moving windows of recent observations and predictions to include in alerts.

Figure 5: The PFA record structure for the protein topic tracking.

Wikipedia pages and we built a separate segmented PFA model for each page. The purpose of this study was to identify changes in interest that might be related to scientific discoveries. Part of the project was to detect changes representing an increase of interest by scientists vs an increase in interest from more general non-technical audiences, such as occurs when the general press picks up a story.

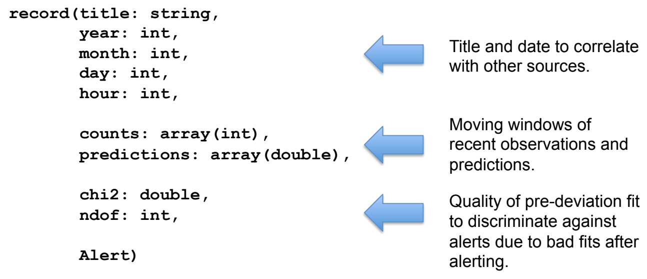
Interest varies wildly from one page (one protein) to another. Some protein's pages were clearly only of interest to practicing biologists, while others, such as insulin, are well-known to the public. It was for this reason that we used a separate model for each page, which we trained and scored independently.

PFA segments models by placing them in arrays (for models we wish to iterate over, such as decision trees in a random forest), key-value maps (for models that we want to look up quickly, such as the example above), and "pools." Pools act like key-value maps, but have a concurrency granularity at the level of individual key-value pairs, not the whole map. This allows for high-speed training and scoring of submodels by a collection of cooperative scoring engines. For this project, we put the page view models for each Wikipedia page into a pool, and let PFA ensure that contributions from different scoring engines would not conflict.

For the submodels, we noticed that page view trends have a strong day-night effect, since the majority of viewers are on one or two continents. For some pages, there is also a week-long trend, since some pages would only be viewed during the work week.

The record structure used by the models is shown in Figure 7.

We used PFA's Holt-Winters model, also known as a triply exponential moving average, to track the periodicity and slow trends. Interesting events are defined to be cases where this model is suddenly a poor predictor of page view behav-



```
record(title: string,
       year: int,
       month: int,
       day: int,
       hour: int,

       counts: array(int),
       predictions: array(double),

       chi2: double,
       ndof: int,

       Alert)
```

Title and date to correlate with other sources.
Moving windows of recent observations and predictions.
Quality of pre-deviation fit to discriminate against alerts due to bad fits after alerting.

Figure 6: The PFA record structure for alerts for the protein topic tracking.

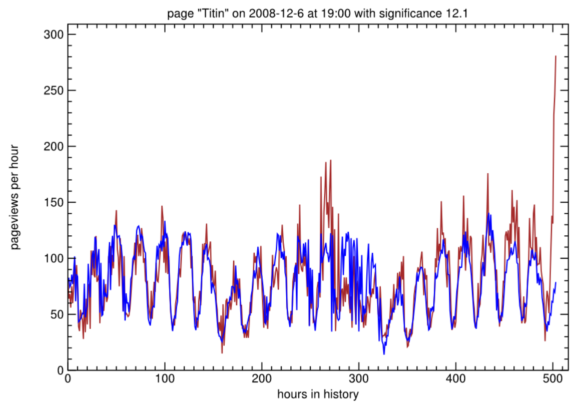


Figure 7: A visualization of the output of the PFA for the segment associated with the titin protein. The blue is the predicted number of page views of the model, the red is the actual observed number of page views. When the observed number of page views differs signifcantly from the predicted view, the model emits an alert.

iors, such as a burst of interest or interest at odd times of day.

Dozens of outliers were discovered, some dramatic and some subtle, and about half could be correlated with relevant news events. The others are potentially interesting cases of rumor or word-of-mouth news in the scientific community.

8.2 Case Study 2: Gaussian Process Model

One of the many models included in the PFA specification is the Gaussian Process [8]. Its inclusion was originally motivated by a request to the PFA Working Group by NIST, but it is a generally useful model— or model component— and therefore should be available as a global library function.

A Gaussian Process essentially smooths a (scalar or vector) function from a given set of sampled points. If these sampled points are taken to be truth from a training phase, the Gaussian Process is a supervised learning technique. However, it could just as easily be a pre- or post-processing step, approximating input data with a curve or reducing noise in scored outputs from another model.

Gaussian Processes also provide a good example of the separation of concerns between the mathematical abstraction presented to the data analyst and vendor-specific optimizations on the backend. The way that it is usually presented, there is no clear separation between training a Gaussian Process model and using it for predictions. Given a training dataset, the Gaussian Process produces a non-parametric fit without an agreed-upon specification for how that PFA must ask for the original training data, not a fit in some vendor-specific coordinates. The PFA `gaussianProcess` function [7, model.reg.gaussianProcess] accepts training data as an array of points and yields predictions.

However, an implementation that re-fits the training data every time a prediction is requested would be very slow. Therefore, Hadrian internally performs the fit once and attaches its non-parametric fit result to the training data array. When the same training data are provided, the implementation first checks for a cached fit result. This fit result never gets out of date because the training data array is immutable.

A similar optimization is provided for k-nearest neighbor models, which are also described by raw training data but are more efficiently searched if the training data are re-cast as a tree. These optimizations are toggled through the top-level PFA field `options`.

The Gaussian Process Model provides a good example of leveraging PFA's ability to compose functions to define new models. In the following code fragment, the math function `m.kernel.rbf` was developed for the Support Vector Machine model, but can be reused to define the Gaussian Process Model as follows:

```
model.reg.gaussianProcess:
    - input
    - {cell: table}
    - null
    - {fcn: m.kernel.rbf, fill: {gamma: 2.0}}
```

Any kernel function could be used here, including user defined PFA functions. Also, the Gaussian Process Model can be used in other PFA functions or in PFA pre- or post-processing code.

8.3 Case Study 3: Supporting Diverse Development Environments

In this case study, we describe a deployed instance of PFA using the Hadrian scoring engine at a client site. The description is high level in order to protect the confidentiality of the client and the project.

The client offered scoring as a service to its customers. A team of data scientists developed models for customers using a variety of different applications and libraries, including R, Scikit-learn and Python. Hadrian was integrated once into the scoring as a service application. PFA exporters were used to export PFA models from the various tools and applications used by the data science team.

Recall that Hadrian is a PFA application written in Scala. For this reason, any J2EE stack to import it as a JAR dependency. For this project, we exposed the scoring function through the Hadrian container's API. Operationally, PFA model files are uploaded to a servlet container, compiled by Hadrian into scoring engines, and exposed to the Java code running in the scoring as a service application. The score function takes an event as a map, validates it against the schema, scores it, and returns the result. Multiple models can be run, and the Java container can provide any pre- or post-processing that is not part of the model. With this approach, a single web-application can be used to score all of the models used by the client in production, independent of what tool or application was used to develop the model and what pre- or post-processing is required.

The exporters used for this project have two parts: the first is written in the native language of the model (R, Python, etc) and goes into the trained models, pulls out the values required in the PFA version, and writes them as JSON while the second takes a PFA template, the input and output schema, and the extract model values as JSON and generates a valid PFA model. This second part is generally written in Python.

After the model is exported, the native and PFA versions are compared on a validation data set and when the model is approved, it is saved with the date and performance metrics so that it can be compared with future candidate models, as part of Champion-Challenger methodology.

8.4 Case Study 4: Scoring over an Enterprise Service Bus

In this case study, we describe a second deployed instance of PFA using the Hadrian scoring engine at a client site. As with the previous case study, the description is high level in order to protect the confidentiality of the client and the project.

In this deployment, MapReduce is used to process large amounts of network data that have been stored in a Hadoop cluster. An application called Hadrian-MR is used to process the data using MapReduce and to produce PFA blocks. These blocks may need to be assembled into a PFA model, depending on the type of MapReduce job run and the statistics calculated. In this way, data scientists can build models over data in the Hadoop cluster and export models in PFA.

The operational systems use a real time distributed messaging platform (an enterprise service bus) that processes billions of messages per day. An application called Hadrian-Actors interfaces to the message platform, listens for messages on the relevant topics, and scores the messages using Hadrian. The output messages are then returned to the messaging platform. In this way, PFA models developed by the data scientists can be easily deployed simply by importing them into the Hadrian-Actor application. The main benefit of this architecture is the ease and speed in which new models can be introduced into the operational environment, even when the models require user defined PFA functions.

9. STATUS OF PFA AND HADRIAN

As of February, 2016, the DMG PFA Working Group has approved version 0.8.1 of the standard. The current standard can be found on the DMG website (www.dmg.org).

To ensure compatibility of PFA implementations, a suite of full-coverage tests were generated for each library function. Since many PFA functions are polymorphic, they are each tested for a broad, representative set of compatible signatures. Each of these signatures are tested for a broad, representative set of values (including infinity, `NaN`, and every possible runtime error). These tests were generated algorithmically, using an XML description of the PFA standard. In all, the tests comprise over 6000 scoring engines, evaluated for more than 1 million sets of values. The JSON file describing these tests is currently 238 MB in size. New PFA implementations can check their compatibility and coverage by evaluating these tests.

Hadrian is available from Github:

`github.com/opendatagroup/hadrian,`

with an open source license. The current version of Hadrian is 0.8.4.

10. SUMMARY AND CONCLUSION

We have described the design of a new extensible language for describing mathematical, statistical and machine learning models called the Portable Format for Analytics (PFA). PFA supports the compositions of models, data transformations, data filtering, data aggregations, and user defined functions, allowing very general user defined models to be described in PFA, as well as the pre- and post-processing required by most deployed models. This represents a significant advance over current standards for describing analytic models. The Data Mining Group is developing the PFA standard and the current version is 0.8.1 and contains over 400 functions. We have also discussed two implementations of PFA conformant scoring engines, one in Scala and one in Python. Finally, we have described four case studies of PFA models that have been deployed using Hadrian.

11. REFERENCES

[1] Data Mining Group, "Predictive Model Markup Language (PMML)," www.dmg.org.

[2] R. Grossman, S. Bailey, A. Ramu, B. Malhi, P. Hallstrom, I. Pulleyn, and X. Qin, "The management and mining of multiple predictive models using the predictive modeling markup language," *Information and Software Technology*, vol. 41, no. 9, pp. 589–595, 1999.

[3] R. L. Grossman, M. Hornick, and G. Mayer, "Data mining standards initiatives," *Communications of the ACM*, vol. 45, no. 8, pp. 59–61, 2002.

[4] A. Guazzelli, W.-C. Lin, and T. Jena, *PMML in action: unleashing the power of open standards for data mining and predictive analytics.* CreateSpace, 2012.

[5] T. White, *Hadoop: The Definitive Guide, 4th Edition.* O'Reilly Media, Inc., 2015.

[6] S. T. Allen, M. Jankowski, and P. Pathirana, *Storm Applied: Strategies for real-time event processing.* Manning Publications Co., 2015.

[7] Data Mining Group, "Portable Format for Analytics (PFA)," www.dmg.org.

[8] C. E. Rasmussen, "Gaussian processes for machine learning," 2006.