Getting Started with WhizzML

The BigML Team

Version 1.17

MACHINE LEARNING MADE BEAUTIFULLY SIMPLE

Last updated March 28, 2024

About this Document

This User Manual provides an introduction to WhizzML, BigML's domain-specific language for the creation of sophisticated, efficient machine learning workflows. We provide an overview of WhizzML's workings and a primer to the language, complete with extended, tutorial examples.

Contents

CHAPTER 1

Introduction

WhizzML is a domain-specific language for the definition of arbitrarily complex machine learning workflows combining BigML's primitives efficiently and executed scalably on our servers. This chapter provides an introduction to the philosophy behind WhizzML, the kinds of problems it addresses and the way in which it solves them. We provide a full introduction to the language in Chapter 2 and a rich set of guided examples in Chapter 3.

1.1 The Need for Machine Learning Workflows

BigML provides a rich set of composable machine learning primitives and algorithms, neatly encapsulated behind an easy-to-use REST API. Sometimes, using a single algorithm (say, a model or an anomaly detector), or a simple combination of them, is enough to solve the problem at hand. However, it is also often the case that the machine learning solution for non-trivial problems requires combining many of the primitives and algorithms into a complex workflow.

Machine learning workflows are most of the time iterative, and typically involve the creation of many intermediate datasets, models, evaluations and predicitions, all of them dynamically interwoven.

Examples of such workflows are common. For instance, in model cross-validation,[1] one generates a sequence of models using different subsections of the input dataset, evaluates them against the rest of the data and computes average quality metrics, in order to asses the adequacy of the modeling algorithm being used. As a second example, in model boosting,[2] a set of weak classifiers is iteratively generated and all of them are finally combined in a better model; each step in the process needs re-weighting of the input data.

There are many other examples of standard composite machine learning procedures and algorithms, including the so-called meta-algorithms that aim at improving existing algorithms by the application of machine learning to the solution space itself.

The list of use cases for workflow automation does not end there: any fully-fledged, customer-facing application that uses machine learning is going to need one or several data pipelines, possibly including feature selection, feature synthesis, data modelization and evaluation, as well as new data scoring in predictive applications.

1.2 The Need for Better Workflow Automation

A common way of expressing these workflows in an executable form is by means of a general purpose programming language, such as Python or Java. A program or script can use BigML's bindings to access, via our REST API, the machine learning primitives that, combined, constitute the sought for solution.

[1] https://en.wikipedia.org/wiki/Cross-validation_(statistics)

[2] https://en.wikipedia.org/wiki/Boosting_(machine_learning)

Automating machine learning workflows by means of standard programming languages is an effective, widely-used solution to the problem of automation, but it presents serious drawbacks in several respects:

Complexity Coding the workflow in a standard programming language implies worrying over details outside of the problem domain and not directly related to the machine learning task.

Reuse REST APIs such as BigML's provide a uniform, language-agnostic interface for solving machine learning problems; however, the moment we codify a workflow using a specific programming language we lose that interoperability. For instance, workflows written in Python, won't be reusable by the Node.js or Java communities.

Scalability Complex, client-side workflows are potentially very hard to optimize, and they lose the automatic parallelism and scalability offered by sophisticated server-side services. Furthermore, there are performance penalties inherent to the client-server paradigm: asynchronously pipelining a remote workflow introduces an extra number of intermediate delays just to check the status of a created resource before moving forward in the workflow.

Client-server brittleness Client-side programs need to take care of non-trivial error conditions derived from network failures. When any step in a complex workflow can potentially fail, handling errors and resume interruptions without losing valuable work already performed becomes a big challenge.

In short, the problem with using programming languages to automate workflows is that it is a too low-level solution: we need to ascend a rung or two in the abstraction ladder, leaving behind the details not directly related to the machine learning problem that the workflow solves.

The underlying REST API is already providing an abstraction layer over computing resources and the nitty-gritty details of many algorithms, but our current scripts are re-introducing inessential details in our workflow specifications.

To better codify our workflows, we need more abstraction.

1.3 Workflow Automation Via a Domain-Specific Language

In the software engineering field, we have long known a very effective method to cure the kind of problems outlined the previous section. To increase the abstraction level of our workflow descriptions and free them from inessential detail, we need to have a way to express our solution in a new language that closer to the domain at hand, i.e., machine learning and feature engineering. In other words, we need a *domain-specific language*, or DSL, for machine learning workflows.

If you are a BigML user, chances are that you are already familiar with another DSL that the platform offers to perform feature synthesis and dataset transformations, namely, Flatline.[3] With that domain-specific language, we offer a solution to the problem of specifying, via mostly declarative symbolic expressions, new fields synthesized from existing ones in a dataset that is traversed using a sliding window. For instance, if you want to create a new field that is the average "age" of every ten instances in you dataset, you can simply write:

```
(avg-window "age" -10)
```

Or, if you want to discretize the field "score" in three segments called, say, "high", "normal" and "low", you would just write:

```
(discretize "score" "low" "normal" "high")
```

The DSL lets you also name things and use those names inside your expression, an archetypal means of abstraction. Here's a simple example where we synthesize the average and the harmonic mean of the fields "nightly" and "daily":

```
(let (d (field "daily")
      n (field "nightly"))
```

[3] https://github.com/bigmlcom/flatline

```
(define acme
  "https://s3.amazonaws.com/bigml-public/csv/nasdaq_aapl.csv")

(define source-id (create-and-wait-source {"remote" acme}))

(define dataset-id (create-and-wait-dataset {"source" source-id}))

(define anomaly-id (create-and-wait-anomaly {"dataset" dataset-id}))

(define input {"Open" 275 "High" 300 "Low" 250})

(define score-id
  (create-and-wait-anomalyscore {"anomaly" anomaly-id
                                 "input_data" input}))

(get (fetch score-id) "score")
```

```
  (list (/ (+ d n) 2)
        (/ (* d n) (+ d n))))
```

Note how in all these expressions we talk only of the domain we are in, namely features (old and new), as well as their combination and composition. Also note how the Flatline language is based on `expressions` for the new features we are creating, rather than in a procedural sequence of instructions of how to synthesize those features.

Now, we want to be able to formulate machine learning workflows of the kind described in the previous sections with a similarly expressive DSL. That is precisely what WhizzML provides: a language specifically tailored to the domain of machine learning workflows, containing primitives for all the tasks available via BigML's API (such as model creation or evaluation or dataset tranformation) and providing powerful means of composition, iteration, recursion and abstraction.

1.4 A WhizzML *Hello World*

WhizzML follows in the footsteps of Flatline and uses symbolic expressions[4] for its core syntax. Let's write a very simple example of a WhizzML workflow to get the gist of the language.

The CSV `nasdaq_acme.csv`, stored in an S3 bucket, contains instances describing features such as the daily opening and closing prices of ACME, Inc. in NASDAQ sessions. We want to use the data to train an anomaly detector that will give us an anomaly score for new instances of the company's opening price and, say, its highest and lowest values during the first half of the morning. The task is easy enough in BigML, but it already comprises several steps:

1. Create a remote source for the CSV file.
2. Create a dataset from that source.
3. Train an anomaly detector on the dataset.
4. For every new tuple of values for the `Open`, `High` and `Low` fields, ask BigML to compute the associated anomaly score.

A good domain-specific language should allow a direct translation of the steps above, using almost exclusively nouns and verbs of the domain (source, dataset, score, and so on), glossing over any details of the underlying mechanism used to compute them (API calls, REST, Python libraries, etc.). Listing ?? shows a fully functional rendering of this example workflow in WhizzML.

[4] https://en.wikipedia.org/wiki/S-expression

```
(define (score-stock name input)
  (let (base "https://s3.amazonaws.com/bigml-public/csv"
        stock (str base "/" name)
        source (create-and-wait-source {"remote" stock})
        dataset (create-and-wait-dataset {"source" source})
        anomaly (create-and-wait-anomaly {"dataset" dataset}))
    (create-and-wait-anomalyscore {"anomaly" anomaly
                                   "input_data" input})))
```

Listing 1: An example WhizzML procedure definition

Here we can already recognize some of the qualities we have been discussing: the predominant keyword is *define*, which introduces definitions, that is, new names for different artifacts: a newly created source, the associated dataset and anomaly detector, values for the new input, etc. We also see how primitives for the creation of BigML resources are readily available (*create-and-wait-source*, *create-and-wait-dataset*, etc.), and take care of their time evolution (in this case, by waiting for their successful completion). The first three steps of the workflow are thus trivially rendered. For the last one, we start by defining the new input, *input*, using a dictionary (or `map`, in WhizzML's dialect) of the form:

```
{"Open" 275 "High" 300 "Low" 250}
```

You can see here how we keep using the problem's domain by directly employing the names of the fields used to score a new instance, which is done in the very next line.

The script finishes by fetching the computed anomaly score, and extracting from its JSON structure the field called "score", which is the quantity that interests us.

Admittedly, careful use of high-level language bindings such as Python's can attain similar levels of readability for workflows as simple as this example's, but, as you'll see in the following chapters WhizzML's readability scales better when the complexity of the workflow increases.

Moreover, there are other immediate advantages of using the WhizzML script for expressing your workflow. For one, it is automatically cross-language and available for any development platform. Even more importantly, it is executable as a single unit fully on the server side, inside BigML's system: there, we not only save the inefficiency of all the network calls needed by, say, a Python script, but also automatically parallelize the job. In that way, we are recovering one of the big advantages of Machine Learning as a service, namely, scalability of the available machine learning tasks, which are mostly lost for composite workflows when using the traditional, bindings-based solution.

Workflow scripts written in WhizzML are reusable server side resources, which makes WhizzML's procedural abstraction capabilities especially powerful. In our running stock example, we can group the individual steps into a parameterized block, and give the new operation a name (e.g., *score-stock*) by means of a procedure definition as shown in listing 1

This new procedure can then be made available, as part of a library, to other scripts. We have thus gained the ability to define new machine learning primitives, going even higher in our abstraction layer, without losing the benefits of performance or scalability associated with a cloud-based service such as BigML.

This section has only scratched the surface of WhizzML's syntax and semantics: Chapter 2 provides a full tutorial on the language, and Chapter 3 shows some non-trivial examples of machine learning workflows written using WhizzML.

1.5 WhizzML REST Resources

As already mentioned, WhizzML scripts are created and managed as BigML REST resources. More concretely, you have at your disposal three kinds of resources:

Library A collection of WhizzML definitions that can be imported by other libraries or scripts. Libraries therefore do not describe directly executable workflows, but rather, new building blocks (in the form of new WhizzML procedures, such as the one in listing 1) to be (re)used in other workflows.

Script A WhizzML script (as the ones we have seen in the previous section) has executable code and an optional list of parameters settable at run time that describe an actual workflow. Scripts can use any of the WhizzML primitives and standard library procedures, as well as import other libraries and use their exported definitions. When creating a script resource, users can also provide in its metadata a list of outputs, in the form of a list of variable names defined in the script's source code.

Execution When a user creates a script, its syntax is checked and it gets pre-compiled and readied for execution. Each script execution is described as a new BigML resource, with a dynamic status (from queued to started to running to finished) and a series of actual outputs. When creating an execution, besides the identifier of the script to be run, users will typically provide the actual values for the script's free parameters (in the same way that, when calling a function in any programming language, you need to provide its arguments' values). It is also possible to pipe together several scripts in a single execution.

All the above resources abide to the rules of any other BigML resource: they're created and owned by users, can be shared, have attached rich metadata (such as parameter descriptions or type annotations) and are essentially immutable to facilitate traceability.

1.6 Running WhizzML Programs Locally

We have seen how WhizzML programs are usually sent for execution to BigML's servers, where they're optimized and parallelized. The WhizzML server-side environment consists of its own virtual machine and runtime that knows how to execute the result of compiling WhizzML libraries and scripts, which are translated and optimized down to a form of assembly language.

An implementation of the WhizzML's compiler and virtual machine running locally on any browser is also availble at `http://bigml.com/whizzml`. That means that you can use them locally in your own machine to compile and execute WhizzML programs, via an online interface.

When running locally, calls in your WhizzML program that create or manipulate BigML resources translate to remote HTTP calls against our API. Not only that; the local compiler and virtual machine lack the automatic paralellization provided by our server side, be it the scalable cloud environment offered by BigML's multitenant service, or the one provided by your own virtual private cloud or on-premise deployment. So of course performance of local executions will be severely impacted.

Nonetheless, the ability to compile and execute WhizzML snippets or full scripts is invaluable when it comes to learning the language and to testing and debugging your workflows. It is in that vein that we provide the option to compile and execute your code using an interactive interface with immediate feedback.

CHAPTER 2

Language Primer

This chapter provides a quick tutorial on the WhizzML language, its basic syntax and some of the key functions in its standard library, in order to get you started writting machine learning workflows. In Chapter 3, we follow up with a few, more detailed, walkthroughs, as a series of progressive tutorials.

Programming with WhizzML is like programming in many other popular languages. You can do arithmetic, define variables and functions, and utilize common data structures like strings, vectors, and maps. Best of all, you get cloud-based machine learning as a first-class operation without any third-party libraries or complex configuration.

In the sections below we often write a WhizzML expression followed by a comment of the form `;; => x`, with `x` the result of the evaluation (the semi-colon starts a comment in WhizzML), as in the following example:

```
(* 2 (+ 2 3)) ;; => 10
```

which means that the WhizzML expression `(* 2 (+ 2 3))` *evaluates* to 10. If you are following along the tutorial using the REPL (see Section 1.6), you'll see the values after inputting the corresponding expression at your `whizzml>` prompt.

2.1 Syntax

One thing that you may have to get used to is that WhizzML uses *prefix notation* for most of its operations. This means that most operations in WhizzML take the form

```
(operator argument1 argument2 argument3...)
```

Adding two and two, for example, looks like this:

```
(+ 2 2) ;; => 4
```

If we want that sum divided by 10, we enclose that expression with a call to the division function, `/`:

```
(/ (+ 2 2) 10) ;; => 0.4
```

Calls to functions that aren't arithmetic operators look the same:

```
(max 2 1 3 7 5) ;; => 7
```

and they can be nested to arbitrary levels (as any other WhizzML expression):

```
(min (max (+ 1 (/ 3 2)) (log 23)) 10) ;; => 3.1354942159291497
```

2.2 Variables and Values

You can bind a value to a variable name by using the `define` keyword:

```
(define x 2)
(+ 2 x) ;; => 4
```

Variable values can be booleans, numbers, strings, lists, maps or sets. You can always refer to a variable's value by its name. Of course, if you try to use an unbound variable name, you'll get an error.

```
(define ready? true)
(define some-value (+ 2 2.5))
some-value ;; => 4.5
another-value ;; Error: Undefined variable 'another-value'
```

We can define lists using `[value1 value2 ...]`, sets using using `#[value1 value2 ...]` and maps using `{key1 value1 key2 value2...}`. There are no restrictions on the types of values, but the keys of a map must be strings:

```
(define x 2)
(define alist [0 x "some-value"])
(define aset #[1 8.3 false])
(define amap1 {"a" x "b" "another value"})
```

The list, set and map types can be used to create data structures as complex as you wish; lists, sets and maps can nest within each other and themselves to arbitrary depths:

```
(define deep {"key" ["value1" {"a" 0 "b" 1} [0 1 [2 3 4]]]})
(define deep2 #[{"key" #[1 2]} [1 {"key2" 43}]])
```

Note that many of the common operations for lists, sets, maps, and strings, such as finding values in the collection, or creating substings, are standard library functions. These are all enumerated in the WhizzML Reference. For example, the standard library function `contains?` tells us if a key is present in a map:

```
(define x {"a" 1 "b" 2})
(contains? x "a") ;; => true
(contains? x "c") ;; => false
```

and `member?` checks membership of an element in a list or set:

```
(define s #[1 8 3])
(member? 1 s) ;; => true
(member? "a" s) ;; => false
(define l [1 9 "2"])
(member? 2 l) ;; => false
(member? "2" l) ;; => true
```

Again, refer to the WhizzML Reference for a full list of these functions.

One can also use a list of identifiers as the first define argument, and assign it to a list of values:

```
(define [a b] [1 2])
a ;; => 1
b ;; => 2
(define alist (list "a" true false "extra"))
(define [c d e] alist)
c ;; => "a"
```

```
d ;; => true
e ;; => false
```

As shown, the assigned value must be an expression that evaluates to a list, but does not need to be an explicit literal. The list value can contain more elements than needed in the assigment: trailing elements are just ignored.

2.3 Local Bindings

Note that `define` forms can only appear at the top level of your programs: in nested contexts, you would use `let` instead: `let` takes a list of bindings as its first argument and an expression as its second. In the expression, you can use the variable names as values. For example, here's an expression to compute the length of a 2-d vector with the given x and y coordinates:

```
(let (x 3
      y 4
      x-squared (* x x)
      y-squared (* y y))
  (sqrt (+ x-squared y-squared)))  ;; => 5.0

;; no x, y, x-squared or y-squared here!
```

As you see in the example above, values are assigned to variables sequentially: first `x` is set to 3, then `y` to 4, after that we can compute `x-squared` and `y-squared` using both `x` and `y`, and finally all four variables are used in the `let` body, which in this case comprises only one expression, namely, `(sqrt (+ x-squared y-squared))`. Also note that outside of the let expression, the mentioned variables are simply not bound.

2.4 Functions

Functions can be defined in two ways in WhizzML, first using the `lambda` keyword, where the syntax is `(lambda (parameter1 parameter2...) expression)`:

```
(define add (lambda (a b) (+ a b)))
(add 2 2) ;; => 4
```

You can also use a modified call to `define` to do the same thing, where the syntax is

```
(define (function-name parameter1 parameter2...) expression)
```

Finally, functions in WhizzML are first class entities, so it's perfectly fine to create a function like this:

```
(define (compose fn1 fn2) (lambda (x) (fn2 (fn1 x))))
```

Here we define a function `compose`, which takes two functions as arguments. It returns another function that takes a single value, `x` as an argument. When the returned function is called, it executes the first function on the input argument, then the second function on the result of the first call. For instance:

```
(define (square x) (* x x))
(define (increment x) (+ x 1))
(define square-and-increment (compose square increment))
(square-and-increment 3) ;; => 10
```

Functions can recursively call themselves:

```
(define (factorial n)
  (if (< n 1)
      1
      (* n (factorial (- n 1)))))
```

and of course it is also possible, although less common, to write mutually recursive functions.

It is also worth noting that list and maps can be used as functions. When maps are used in such a way, the key given as argument will be looked up in the map and its value (if found) returned, as in the following examples:

```
(define m {"a" 3 "b" 32})
(m "a")  ;; => 3
(m "c" "not-found") ;; => "not-found"
(m ["x" "c"] "not-found") ;; => "not-found"
(define mm {"a" 42 "x" m})
(mm ["x" "b"]) ;; => 32
(mm ["x" "b"] "not-found") ;; => 32
```

For lists, the argument must be an integer representing a position in that list, and the value of the element at that position is returned:

```
(["a" 2 {"b" 3}] 0) ;; => "a"
((["a" 2 {"b" 3}] 2) "b") ;; => 3
```

It is also possible to mix keys and element positions in a lookup path:

```
(define m {"a" 3
           "b" [-1
                -2
                {"c" {"a" 42
                      "d" [1 2 3 true]}}]})
(m ["b" 0]) ;; => -1
(m ["b" 2 "c" "d" 3]) ;; => true
```

2.5 Control Flow

Conditional statements work as you might expect by now, with the syntax

```
(if condition then-expression else-expression)
```

E.g.,

```
(if (= 1 2) "They're equal" "They're not equal") ;; =>  "They're not equal"
```

Note the single equal sign test for equality, which is different from most programming languages. WhizzML has a full complement of relational (<, >, ...) and boolean (`and`, `or`, ...) operators that can be found in the WhizzML Language Reference. There are also a number of other control flow tests that we won't go over here (`cond` is particularly nice) that you may find useful.

Explict looping is usually not necessary in WhizzML because there are so many functions that operate on lists of values. For example, the `map` function takes as arguments a function and a list of values, and returns a list with that function applied to each value in turn.

```
(define (square x) (* x x))
(map square [1 2 3 4 5]) ;; => [1 4 9 16 25]
(map (lambda (x) (+ x 1)) [1 2 3]) ;; => [2 3 4]
```

There are several functions like this, such as `reduce` and `filter`. For a full list of them, we again refer you to the language reference. If you must loop explicitly, you can also check there for the `loop ...recur` looping construct.

2.6 Immutability

An important concept in WhizzML (and some other languages) that may take some getting used to is the *immutability* of variables, which means that the value of a variable never changes after its initial assignment. Operations that alter a variable's value return a modified copy of the data. For example, the function `assoc` associates a new key with an existing map, but does not modify the existing map. Instead, it returns a copy of the old map, with the new key added:

```
(define x {"a" 1 "b" 2})
(define y (assoc x "c" 3))
x ;; => {"a" 1 "b" 2}
y ;; => {"a" 1 "b" 2 "c" 3}
```

The same general principle applies to transformations on lists and strings.

2.7 Machine Learning

You can use WhizzML to create any resource that you normally could with the BigML API. For example, let's create a data source using the canonical `iris` dataset, downloaded from the UCI dataset repository, and a dataset from it:

```
(define r
  "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data")
(define iris-source-id (create-source {"remote" r "name" "test"}))
(define iris-dataset-id (create-dataset iris-source-id))
```

The `create-source` function will make a call to the BigML API returning the identifier for the newly created resource. The map of parameters should have keys that match the parameters for source creation listed in the BigML API documentation. Source creation will be handled in the background by BigML, while control is returned to your script. But you can use `iris-source-id` immediately: the platform will make sure that the resource is finished before being used as the parent of, say, a dataset, as shown in the call to `create-dataset` immediately below.

There are functionally similar create methods for all BigML resource types (e.g., `create-ensemble`, `create-sample`, and so on).

You can also `fetch` any resource to pull its JSON structure down into a WhizzML map:

```
(let (src-map (fetch src-id))
  (log-info "The source name is: " (get src-map "name")))
```

Note that WhizzML will `not` automatically wait for a resource to be finished before fetching it: `fetch` will give you back its current status, which may not be finished. If you want to explicitly wait until the resource is finished, you can accomplish it via the `wait` primitive, which returns the same identifier passed as argument, after waiting for the given resource to reach the finished state:

```
(let (src-map (fetch (wait src-id)))
  (log-info "The final source fields are: " (get src-map "fields")))
```

It is possible to update and delete any resource in a similar way:

```
(update src-id {"name" "more testing"
                "description" "a whizzml source"
                "fields" {"000000" {"name" "first field"}}})
(delete src-id)
```

and since updates (like creations, but unlike fetches or deletions), can only happen when the involved resource is finished, WhizzML will automatically wait for the resources to be updated to be finished.

This completes our whirlwind tour of WhizzML's syntax and standard library. We have covered just the basics to get you started reading and writing WhizzML scripts and libraries. In the following chapter we will explore the language and its use further by means of actual examples solving real machine learning problems.

CHAPTER 3

Extended Examples

In this chapter we present some tutorial examples of realistic WhizzML scripts. See also the examples folder[1] in WhizzML's repository for more sample scripts.

3.1 Model and Batch Prediction

Let's whet our appetite with something simple: imagine that you routinely create a model from an input source and use it to predict over a second source that doesn't have its objective field filled in. The workflow to accomplish this task is easy:

- Create a dataset for each source, given the source ids.
- Create a model from the first dataset.
- Perform a batch prediction of the model using the second dataset.
- Clean up unused resources.

Our WhizzML script will need then two inputs, namely, the identifiers of the two sources. Let's call them `model-source-id` and `prediction-source-id`. It will also have a single output, the identifier of the final batch prediction. Thus, our JSON request to create this script will contain the following fields:

```
{"source_code": "...",
 "parameters": [{"name": "model-source-id",
                 "type": "source-id"},
                {"name": "prediction-source-id",
                 "type": "source-id"}],
 "outputs": [{"name": "result",
              "type": "batchprediction-id"}]}
```

where we have yet to fill-in the actual source code for our script. The first thing we need to do is create a dataset for each source; for instance, for the model dataset we would write:

```
(create-dataset model-source-id)
```

and likewise for `prediction-source-id`, which could also be written using the version of `create` that takes a map with the request's options (the two forms are equivalent in this case):

```
(create-dataset {"source" model-source-id})
```

These function calls will return the identifiers of the created datasets, let's call them `model-dataset-id` and `prediction-dataset-id`. With the first one, we'd like to create a model:

[1] https://github.com/bigmlcom/whizzml/tree/next/examples/scripts

```
(define model-dataset-id (create-dataset model-source-id))

(define prediction-dataset-id (create-dataset prediction-source-id))

(define model-id (create-model model-dataset-id))

(define result
  (wait (create-batchprediction model-id prediction-dataset-id)))

(for (id [model-dataset-id prediction-dataset-id model-id])
  (delete id))
```

Listing 2: Model and batch prediction from two sources

```
(create-model model-dataset-id)
```

and we will use the returned `model-id` together with the prediction dataset id to create a batch prediction:

```
(create-batchprediction model-id prediction-dataset-id)
```

Note that our call to `create-model` above will wait until its parent dataset is finished to proceed, and that `create-batchprediction` will do the same for the model and dataset it depends on: we don't need explicit calls to `wait` in our code so far. It is also worth noting that our two datasets will most probably be processed in parallel, and that even the model creation could be started without waiting for the prediction dataset to be complete, since they are independent resources.

We want to return the ouput of this function call in the variable `result` (the name we've declared in our outputs), so we'd better `define` it so:

```
(define result
  (wait (create-batchprediction model-id prediction-dataset-id)))
```

In this case, we explicitly `wait` until the batch prediction is finished, since `result` is not used anywhere else and therefore no other primitive will wait for it.

We're essentially done: all that's needed is to weave the snippets above together using intermediate variables, and delete intermediate resources when we are done, as shown in listing 2.

3.2 Model or Ensemble?

Given a pre-existing source, we want to check whether building an ensemble is actually better than using a simple tree, using the result of their evaluations as arbiter.

To that end, we want to first create a training dataset and a test dataset by splitting the dataset containing all the input data. The standard library already provides the primitive `create-random-dataset-split` to that end. For the sake of illustration, however, let's define a procedure to sample an existing dataset, given its identifier. Then, we can define another procedure that actually generates the two splits from the one dataset:

```
(define (sample-dataset origin-id rate out-of-bag?)
  (create-dataset {"origin_dataset" origin-id
                   "sample_rate" rate
                   "out_of_bag" out-of-bag?
                   "seed" "example-seed-0001"}))

(define (split-dataset origin-id rate)
  (list (sample-dataset origin-id rate false)
        (sample-dataset origin-id rate true)))
```

As you can see, `split-dataset` returns a list of two new dataset ids, the first containing a sampling of the given size, and the second containing all instances not in the first (thanks to the `out_of_bag` flag, and the fact that we are using the same seed).

Now all we need to do is train a model and an ensemble with the first dataset, evaluate both with the second dataset, and compare the results. Creating a model with default configuration parameters from a dataset is trivial using `create-model`. For ensembles, we define a simple function that takes as arguments the parent dataset and the number of models in the ensemble:

```
(define (make-ensemble ds-id size)
  (create-ensemble ds-id {"number_of_models" size}))
```

Let's also define an accessor to the `f-measure` of any evaluation, making sure that the given evaluation has reached status 5 (i.e., is finished) before fetching it:

```
(define (f-measure evaluation-id)
  (get-in (fetch (wait evaluation-id)) ["result" "model" "average_f_measure"]))
```

and now we have all the ingredients to our final function, which takes the identifier of our source and returns either a model identifier or an ensemble identifier:

```
(define (model-or-ensemble src-id)
  (let (ds-id (create-and-wait-dataset {"source" src-id}) ;; full dataset
        ids (create-random-dataset-split ds-id 0.8) ;; split it 80/20
        train-id (nth ids 0)                        ;; 80% for training
        test-id (nth ids 1)                         ;; and 20% for evaluations
        m-id (create-model train-id)                ;; create a model
        e-id (make-ensemble train-id 15)            ;; and an ensemble
        m-f (f-measure (create-evaluation m-id test-id))  ;; evaluate
        e-f (f-measure (create-evaluation e-id test-id))) ;; both
    (log-info "model f " m-f " / ensemble f " e-f)
    (if (> m-f e-f)  m-id e-id)))
```

We could now either create a library exporting `model-or-ensemble`, or just add a call to that function at the end of our script, e.g.

```
(model-or-ensemble "source/56fbbfea200d5a3403000db7")
```

Or we could make the source identifier a parameter of our script. We give it a name, say `input-source-id`, and then use that name in the source of the script, which would then look like the one shown in listing 3.

Thus, the JSON request for creating our script, would look like this:

```
{"source_code": "...",
 "parameters": [{"name": "input-source-id",
                 "type": "source-id",
                 "description": "The source to be analyzed"}],
 "outputs": [{"name": "result",
              "type": "resource-id",
              "description": "Either a model or an ensemble id"}]}
```

where we have omitted the source code (which would be that of listing 3). Once the script is created and assigned an id, we can execute it with a request containing the concrete value of the input source id that we want to use in the arguments of the corresponding JSON request:

```
(define (make-ensemble ds-id size)
  (create-ensemble ds-id {"number_of_models" size}))

(define (f-measure ev-id)
  (get-in (fetch (wait ev-id)) ["result" "model" "average_f_measure"]))

(define (model-or-ensemble src-id)
  (let (ds-id (create-and-wait-dataset {"source" src-id}) ;; full dataset
        ids (create-random-dataset-split ds-id 0.8) ;; split it 80/20
        train-id (nth ids 0)                        ;; 80% for training
        test-id (nth ids 1)                         ;; and 20% for evaluations
        m-id (create-model train-id)                ;; create a model
        e-id (make-ensemble train-id 15)            ;; and an ensemble
        m-f (f-measure (create-evaluation m-id test-id))  ;; evaluate
        e-f (f-measure (create-evaluation e-id test-id))) ;; both
    (log-info "model f " m-f " / ensemble f " e-f)
    (if (> m-f e-f)  m-id e-id)))

;; Compute the result of the script execution
;; - parameters: [{"name": "input-source-id", "type": "source-id"}]
;; - outputs: [{"name": "result", "type": "resource-id"}]

(define result (model-or-ensemble input-source-id))
```

Listing 3: Script choosing the more performant predictor: model or ensemble

```
{"script": "script/56e32430b95b3920960003c3",
 "arguments": ["input-source-id", "source/56fbbfea200d5a3403000db7"]}
```

Please use the noidx option in the documentclass invocation.

bigml®