Protocols and Structures for Inference: A RESTful API for Machine Learning

James Montgomery JAMES.MONTGOMERY@UTAS.EDU.AU
School of Engineering and ICT, University of Tasmania, Hobart TAS Australia

Mark D. Reid MARK.REID@ANU.EDU.AU
Research School of Computer Science, Australian National University, Canberra ACT Australia & NICTA

Barry Drake BARRY.DRAKE@CISRA.CANON.COM.AU
Canon Information Systems Research Australia, Sydney NSW Australia

Editor: Louis Dorard, Mark D. Reid and Francisco J. Martin

Abstract

Diversity in machine learning APIs (in both software toolkits and web services), works against realising machine learning's full potential, making it difficult to draw on individual algorithms from different products or to compose multiple algorithms to solve complex tasks. This paper introduces the *Protocols and Structures for Inference* (PSI) service architecture and specification, which presents inferential entities—relations, attributes, learners and predictors—as RESTful web resources that are accessible via a common but flexible and extensible interface. Resources describe the data they ingest or emit using a variant of the JSON schema language, and the API has mechanisms to support non-JSON data and future extension of service features.

Keywords: RESTful API, web service, schema

1. Introduction

Machine learning algorithms are implemented across a wide array of toolkits, such as Weka (Hall et al., 2009), Orange (Demsar and Zupan, 2004), Shogun (Sonnenburg et al., 2010) and scikit-learn (Pedregosa et al., 2011), as well as custom-built research software. Although each toolkit or one-off implementation is individually powerful, differences in the programming language used and supported dataset formats make it difficult to use the best features from each. These differences limit their *accessibility*, since new users may have to learn a new programming language to run a learner or write a parser for a new data format, and their *interoperability*, requiring data format converters and multiple language platforms.

One way of improving these software packages' accessibility and interoperability is to provide an abstracted interface to them via web services. Resource-oriented architectures (ROAs) (Richardson and Ruby, 2007) in the REpresentational State Transfer (REST) style appear well suited to this task, given the natural alignment of REST's design philosophy with the desire to hide implementation-specific details. There has been recent, rapid growth in the area of machine learning web services, including services such as the Google Prediction API, OpenTox (Hardy et al., 2010), BigML, Microsoft's Azure ML, Wise.io, and many more. However, despite this wide range of services, improvements in accessibility and interoperability have not necessarily followed. The language each service speaks—HTTP and JSON—may now be consistent, but each service presents its own distinct API while necessarily offering only a subset of inference tools. For instance, the Google Prediction API can perform only classification or regression while, in contrast, OpenTox's range of learning algorithms is extensible but restricted to the toxicology domain. Composing these existing services requires the client to understand each distinct API and to perform a considerable amount of data conversion on the client-side.

The *Protocols and Structures for Inference* (PSI) project has produced a specification for a RESTful API in which each of the main inferential entities—datasets (known as relations), attributes, data transformers, learners and predictors—are resources, with an overall interface that is sufficiently flexible to tackle a broad range of machine learning problems and workflows using a variety of different algorithms. The key feature of our approach is an emphasis on resource *composition* which allows the creation of *federated* machine learning solutions (e.g., allowing for workflows that use learning algorithms and data hosted on different severs). In order to promote composition, we introduce a *schema language* that describes those parts of resources' interfaces that vary depending on the data or learning algorithm. PSI is thus not an implementation but an attempt at building a *standard* for presenting machine learning services in a consistent fashion. This paper presents an overview of the API's design, with the full specification and additional examples available at `http://psikit.net/`. An example PSI service that exposes learning algorithms from Weka and scikit-learn is available at `http://poseidon.cecs.anu.edu.au` and a demonstration in-browser JavaScript client that can be used with any PSI-compliant service is available at `http://psi.cecs.anu.edu.au/demo`.

2. A Resource-Oriented Architecture for Machine Learning

There are a diverse range of problems within machine learning that can be cast in a *data-learner-predictor* framework, in which a learning algorithm is applied to a dataset of *instances* of some phenomenon of interest in order to produce a predictor that can make inferences about additional, previously unseen instances. Applicable problems include classification and regression as well as more varied problems such as ranking, dimensional reduction and collaborative filtering.

In the PSI architecture, datasets, known as *relations* (borrowing from database terminology), the *attributes* of those relations, *transformations* that may be applied to that data (and to predictions), *learners* and *predictors* are all resources that can be composed to perform different inference activities. Each resource is identified by its URI and interaction is via the HTTP methods GET, POST and DELETE. Resource representations, request and response messages, and attribute and predicted values are all represented in JSON.

Service providers are free to offer any subset of PSI resources that suits their purposes. For instance, one service may provide data through relation and attribute resources, while another service could offer learning algorithms and the predictors they produce. This flexibility allows federated machine learning solutions to be created.

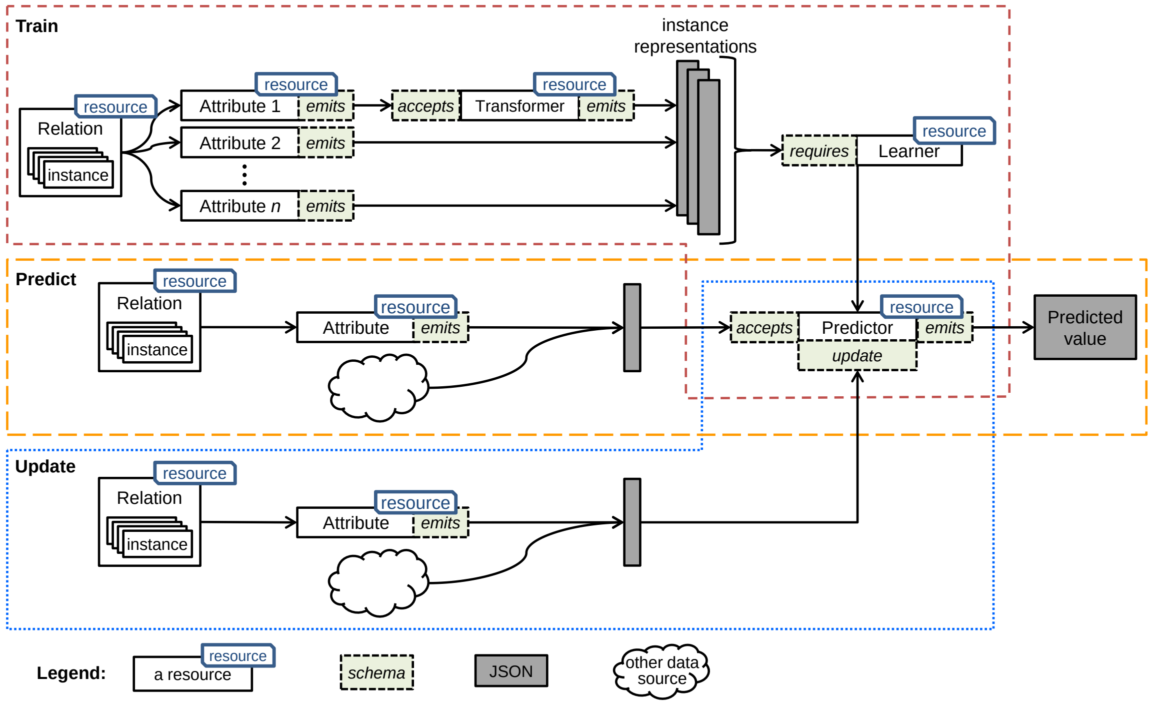


Figure 1: Common data flows through a PSI service. In this illustration the output from Attribute 1 is transformed before it is used in learning

2.1. Flexibility Through Schema

To support a range of learning activities through a common interface, different learner resources must be able to specify their requirements in terms of data they ingest and the parameters they may need. On the other side, relation attributes must be able to express the structure of the data they provide. To serve both needs, PSI uses a custom *schema language* derived from, and compilable to, JSON Schema (Galiegue et al., 2013a,b; Luff et al., 2013). It is this use of schema that allows PSI to safely connect instances (represented by their attributes) with the learners that will process them. The inputs and outputs of (general-purpose) transformers and predictors are also described by schema. Figure 1 illustrates the three primary activities within the framework: training, prediction, and updating. In each activity, schema define the output characteristics or input requirements of resources in the workflow. Further, schema can support the generation of custom controls in client software, in a manner similar to HTML form controls (see Section 4 below).

Given the potential complexity of JSON Schema expressions, the PSI Schema language defines a number of abbreviations, including a set of common named schema that each PSI service should understand. Each of these common schema is also a resource, and may accept URI query string arguments that augment its representation with additional details such as a default value, bounds and description. In PSI's variant of JSON Schema, keys and values

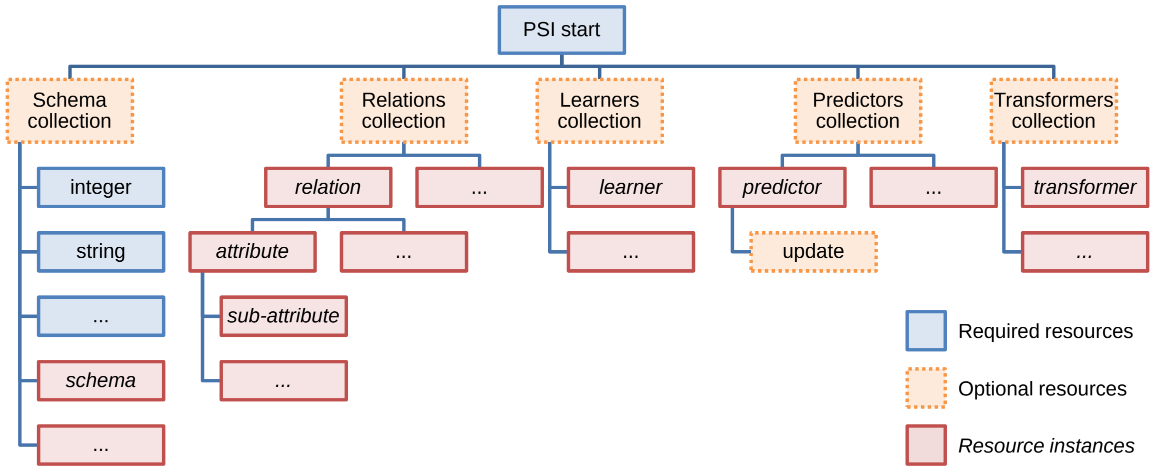


Figure 2: PSI Service Resource Hierarchy. All resource collections are optional and will depend on the nature of the service being offered. A service may also consist of a sole predictor, transformer or relation (and its attributes)

prefixed with $ generally refer to these predefined schema, with many examples appearing in the following sections.

The following sections describe each of the other PSI resource types, illustrated using hypothetical PSI services: one offered by a research team and another offered by a commercial machine learning service, against which the research team wants to compare its learning algorithms. The first could be of particular benefit to the research community: a publication describing a new learning algorithm becomes interactive by including the URI of a learner resource implementing that algorithm. Further, trained predictors can be made available as isolated services.

2.2. Service Discovery

Each PSI service has a single published entry URI, the JSON representation of which includes links to collections of the relations, learners, predictors, transformers and schema provided by that service.[1] Each collection has the same representation, which includes an array of links to the resources it contains. In this way the resources of a PSI service naturally form a hierarchy, which is illustrated in Figure 2 and explored in more detail over subsequent sections.

Consider an example in which a research team has data and learning algorithms it wishes to share. The team could offer these as a PSI service located at `http://example.org`, which responds to a GET request with the following:

1. As a PSI service may also just consist of an isolated predictor or relation, this top-level discovery resource and the collections beneath it is not mandatory, although the overhead for providing it is extremely small.

```
{
  "psiType":     "service",
  "uri":         "http://example.org",
  "relations":   "http://example.org/data",
  "schema":      "http://example.org/schema",
  "learners":    "http://example.org/learn",
  "predictors":  "http://example.org/infer",
  "transformers": "http://example.org/transform"
}
```

indicating that the service potentially offers all PSI resource types. The `psiType` attribute, a part of all PSI resource representations, serves as a substitute for more specific media types for each distinct kind of resource.

Following the `relations` link (i.e., performing a GET operation on its URI) returns the representation:

```
{
  "psiType":     "resource-list",
  "resources":   [ "http://example.org/flowers" ]
}
```

indicating that there is one dataset available, which can be interrogated further for details.

2.3. Relations and Attributes

Datasets in PSI are known as *relations*, a label borrowed from database terminology to connote that they are collections of instances that share the same set of attributes. Each of a relation's attributes is a resource that transforms the underlying instance data—stored in a database table, flat file, etc.—into representations that can be consumed by other PSI resources or client software. Although the term "attribute" in machine learning often indicates a single, atomic value, PSI attributes need not be only atomic-valued and may instead represent object and list structures. A single PSI attribute may thus represent all the data on which learning will take place. Further, although PSI represents values in JSON, atomic values are not limited to the atomic data types of JSON, but may be any data with an associated media type. The mechanism for handling such "rich values" is described in Section 3.1 below.

The initial set of attributes associated with a relation is at the discretion of the service provider, and may be a set of atomic-valued attributes or a single structured-attribute that describes an entire instance. Each structured attribute is composed of other attribute resources, so structured values may be decomposed as needed. New structured attributes can be created by composing existing attributes (referred to by their URIs) in an attribute definition POSTed to the relation. Thus if the "shape" of data produced by a relation's initial set of attributes is not compatible with a particular learner then new attributes can be defined to reshape the data into a compatible form.

Continuing the example from above, the hypothetical research team has a dataset describing flowers in terms of their physical dimensions and species. The team uses this data to train and test its own learning algorithms, but also wishes to share it with others, so makes it available as a PSI relation resource, located at `http://example.org/flowers`, which may be published independently or discovered through the service's main entry point at `http://example.org`. This relation's attributes consist of an atomic-valued species and an array-valued attribute of flower dimensions, which reside below the relation's URI at `/species` and `/measurements`, respectively:

```
{
  "psiType":    "relation",
  "uri":        "http://example.org/flowers",
  "description":"Flower species & physical dimensions",
  "size":       150,
  "defaultAttribute":
      "http://example.org/flowers/measurements",
  "attributes": [
    "http://example.org/flowers/species",
    "http://example.org/flowers/measurements"
  ],
  "querySchema": { ... }
}
```

Relations may optionally support service-specific queries (for instance, to select a subset of the data for use in k-fold cross validation), the format of which is given by the `querySchema` property of their representation (details omitted from this example for brevity).

Examining the `measurements` attribute of this relation reveals the structure of the values it produces and URIs for its four sub-attributes corresponding to the elements of the array values it emits (URIs omitted for brevity):

```
{
  "psiType":  "attribute",
  "uri":      "http://example.org/flowers/measurements",
  "description":  "A structured attribute for presenting flower dimensions",
  "relation": "http://example.org/flowers",
  "emits":  {
    "$array": { "items": [ "$number", "$number", "$number", "$number" ] }
  },
  "subattributes": { ... },
  "querySchema": { ... }
}
```

The value of a particular (indexed) instance or of all instances may be obtained by appending the URI query string `instance=n` or `instance=all`, respectively, which results in PSI value representations such as:

```
{
  "psiType":  "value",
  "value":    [ 3.5, 5.1, 0.2, 1.4 ]
}
```

if a single instance is requested, or

```
{
  "psiType":  "value",
  "valueList": [
    [ 3.5, 5.1, 0.2, 1.4 ], ...
  ]
}
```

if all instances are requested (the ellipsis indicates the 149 elided values).

The PSI "value" representation is a generic container used for all values emitted by attributes, transformers (discussed next) and predictors, which increases the number of ways in which different value-generating resources may be connected.

2.4. Transformers: Functions as Services

A common need in machine learning is to transform values prior to learning, either to generate additional attributes (for instance, when performing linear regression on higher order polynomials) or to convert a value into the appropriate type for a learning algorithm (for instance, transforming a web page into a bag of words for document classification). The PSI framework abstracts the notion of a function as a *transformer* resource, which accepts and emits JSON values (the structure of which is defined by schema). As with mathematical functions, transformers may be joined (i.e., composed) to produce arbitrarily complex transformation pipelines, provided that the input and output schema of the transformers in the chain are compatible. A transformer resource responds to a join request with the URI of the joined transformer resource, whose *accepts* schema is that of the first transformer and *emits* schema is that of the second.[2]

Extending the earlier example, the research team may have, among its advertised collection of transformers, one that calculates second-order combinations of feature values:

```
{
  "psiType":  "transformer",
  "uri":      "http://example.org/transform/quadratic",
  "description":  "Computes x^a_i x^b_j for all x_i, x_j in input and 0<=a+b<=2",
  "accepts":  { "$array": { "items": "$number" } },
  "emits":    { "$array": { "items": "$number" } }
}
```

Transformers can be applied directly to values encoded as part of the query string in their URI. For instance, a GET request to this transformer's URI with the query string `value=[-2,3]` would produce the PSI value:

```
{
  "psiType":  "value",
  "value":  [ 1, -2, 3, 4, -6, 9 ]
}
```

Attribute resources may also be joined with transformers, which fulfills the common need to transform an entire relation prior to training. However, it is not the *values* of attributes that are submitted to learner resources but the URIs of the attributes themselves, as part of a learning task.

2.5. Learners and Learning Tasks

Learners are resources for generating predictors from relations. They do so by processing *tasks*, which include algorithm parameter settings and, in a *resources* property of a task, representations of one or attributes that will provide the training data. As different learning algorithms have differing restrictions on the type of information they can process (and their parameters), each learner reports a schema defining the structure of valid tasks. Part of this schema defines the valid structure of the attributes' descriptions such that their *emits* schema are compatible with the data needs of the learner. The PSI specification defines schema for a number of common attribute types, which reduces the complexity of both defining and interpreting task schema.

2. The new, joined transformer resource may or may not actually exist within the service. In the prototype PSI service implementation, joins are encoded in the query string fragment of a transformer's URI.

The hypothetical research team introduced earlier wants to test its own algorithms against a commercial machine learning service's offerings. The company in question doesn't want to distribute copies of its proprietary learning algorithm, so makes it available as a PSI learner resource at `http://example.com/tryusout/knn`. The research team GET the learner's representation and discover it has the following task schema:

```
{
  "?k" : { "$integer": { "default": 1, "min": 1, "description": "The number of
      nearest neighbours to examine" } },
  "/resources": {
    "/target":  { "$nominalAttribute" : { "allItems" : "$string" } },
    "/source":  { "$arrayAttribute" : { "allItems" : "$numericValueSchema" } }
  }
}
```

The schema indicates that the learner (apparently a variant of the k-nearest neighbour algorithm) has an optional integer parameter k, the number of neighbours to consider, requires an attribute to read each instance's nominal class and another attribute that describes an instance's features as a list of numbers.

Using this task schema the research team constructs the following learning task that tells the company's `knn` learner to use $k = 3$ neighbours and to obtain training instances from the team's dataset using its attributes for reading instances' species (target) and measurements (source):

```
{
  "k": 3,
  "resources" : {
    "relation" : "$http://example.org/flowers",
    "target" : "$http://example.org/flowers/species",
    "source" : "$http://example.org/flowers/measurements"
  }
}
```

where the $ prefix indicates that the URIs are references that should be resolved, that is, their values should be replaced by the result of GET requests to those URIs. The company's service responds with the URI of the newly trained predictor: `http://example.com/user/thx1138`.

2.6. Predictors

A predictor is a transformer that is constructed by a learner. It thus has the same representation and behaviour as any other transformer, and so may be composed with other transformers or with an attribute (to provide predictions over a relation). Both predictors and (untrained) transformers may include provenance information in their representations. In the case of predictors this can include the URI of the PSI learner that produced them, but the structure is otherwise open to be used however a service provider wishes.

Some learning algorithms, and hence the predictors they produce, support retraining with additional examples. Such "updatable" predictors include an additional URI in their representation that may be queried for the schema of values that may be used in retraining. One or more values conforming to this schema may then be POSTed to the update URI, after which the service will respond with the URI of the retrained predictor. Whether this is the same as the original predictor or a new resource is left to the service provider's discretion.

To conclude the example begun earlier, the research team now has, in addition to its own internally-trained predictor, a predictor that resides at `http://example.com/user/thx1138`, and wishes to compare the performance of the two. Evaluation of predictor performance is currently outside the specification, a deliberate design choice given the wide variety of learning tasks and corresponding evaluation metrics. However, it would be straightforward to develop "PSI-aware" evaluation services that could be applied to a variety of PSI-compatiable learners and relations. In this hypothetical case, the research team has a separate dataset for testing, which is located at `http://example.org/test-flowers`. Both its own and the `http://example.com`-trained predictor can be applied to this dataset (by joining the relevant attribute with each predictor in turn) to produce predictions that can be compared against the correct answers. Satisfied that its own flower species predictor is competitive with the commercial offering, the research team publicises the predictor's URI so that it may be freely used by the public.

3. Service Flexibility

A guiding principle in the design of the PSI framework is that it does not, as far as practicable, preclude any particular mode of machine learning. The use of schema for specifying data characteristics and the needs of learning algorithm is a core part of this approach. Two additional features of the framework that assist in achieving this goal are its mechanisms for dealing with non-JSON data and for providing additional points of extension that are discoverable by client software.

3.1. Rich Values

While the JSON data format can support structured values through dictionaries (key–value pairs) and lists, atomic data is fundamentally limited to integers, real numbers, Boolean values, and strings. Although complex data such as images may be transformed into some representation using these data types (e.g., an image as a list of RGB triples, one per pixel), the extra work in doing so presents a clear barrier to adoption of the approach. To support non-JSON data, the PSI framework introduces the notion of "rich values", in which data that cannot be neatly represented in JSON is encoded as either an HTTP or Data URI. In the first case the data may be obtained by GETting a representation of the resource at the nominated URI, while in the second case the data is encoded (using Base64, for instance) as a Data URI string. The space overhead this entails (33% in the case of Base64) is offset by the benefits of greatly extending the data formats PSI services may work with and, in the future, may be offset by the use of binary JSON encodings.

Rich values are described in PSI's schema language by referring to the data's media type. Consider again the hypothetical research team, which has augmented its `flowers` dataset with images of the flowers described, with the following new attribute:

```
{
  "psiType":  "attribute",
  "uri":      "http://example.org/flowers/image",
  "relation": "http://example.org/flowers",
  "emits":    "@image/jpeg"
}
```

Requesting the image of the first instance produces a data URI-encoded JPEG image (in which the ellipsis indicates the 26,500 character URI has been truncated):

```
{
  "psiType": "value",
  "value":   "data:image/jpeg;base64,/9j/4AAQSkZJRgABAQEA..."
}
```

A classification learner that accepts JPEG images as training data could be trained using this additional data from the `flowers` relation. The predictor produced could then be made available for use by members of the public who wish to identify flowers they have encountered. Suitable transformer resources could also be made available to deal with non-JPEG data.

3.2. Linking to related services and actions

A key characteristic of RESTful services is that resource representations indicate (either via their media type or explicitly via embedded links) the actions that may be taken with that resource. To this end, most PSI resource representations may contain a `relatedResources` property that is a collection of Link Description Objects (LDOs), which are part of the JSON Hyperschema Luff et al. (2013) specification with a role similar to the `link` element of HTML documents. This allows service providers to give additional assistance to clients (for instance, providing links to learners that are compatible with a particular relation and its attributes) or to link to functions that are outside the core PSI specification.

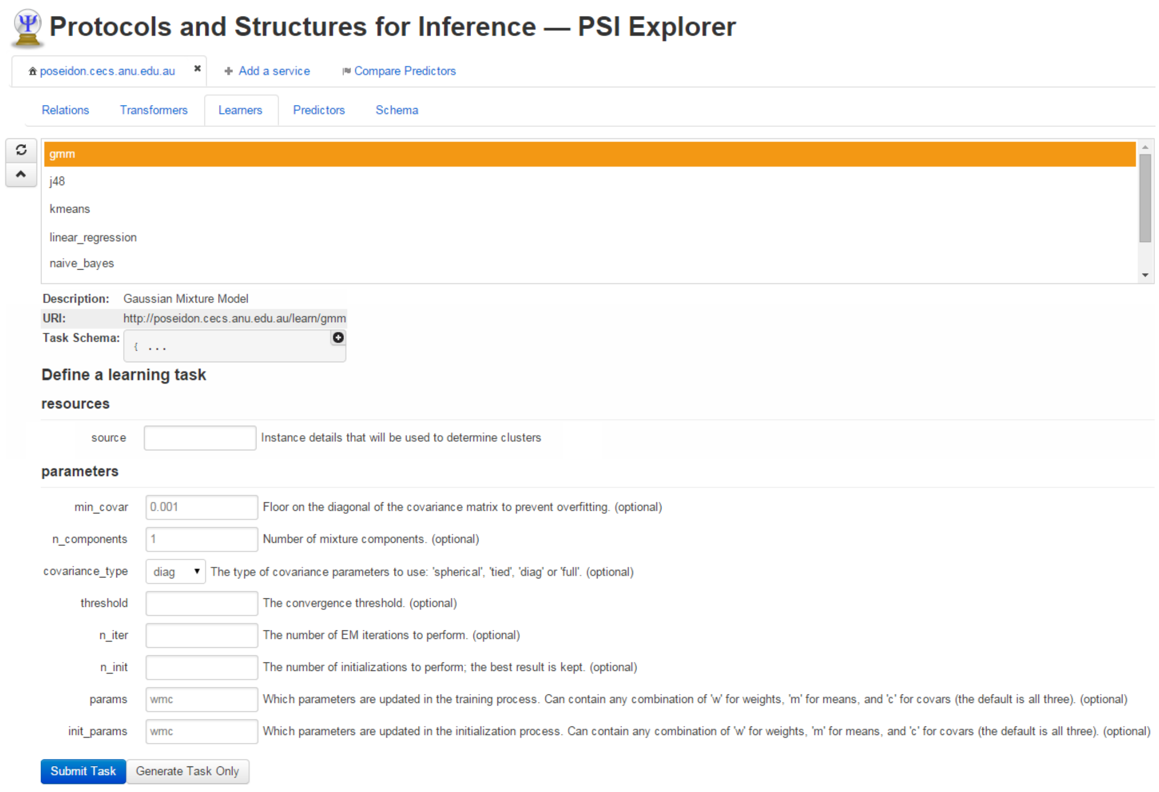
4. Demonstration Implementations

Prototype implementations of a PSI service and client have been developed, the sources of which are available at `https://github.com/orgs/psi-project`. The demonstration service was developed in Java, and an instance that exposes learning algorithms from the Java-based Weka library (Hall et al., 2009) and Python-based scikit-learn (Pedregosa et al., 2011), as well as providing sample datasets and transformers, is available at `http://poseidon.cecs.anu.edu.au`.

The client is a browser-based JavaScript application, available at `http://psi.cecs.anu.edu.au/demo`. It provides basic access to the entire service API and can communicate with any service adhering to the PSI specification. HTML forms for both querying relations and constructing learning tasks are generated from the schema returned by relation and learner resources. This is done in two steps: first the PSI Schema is compiled to JSON Schema, then a third-party JavaScript library is used to generate the form elements, including data validation based on constraints expressed in the schema.

Figures 3 and 4 present two screen captures from the client. Figure 3 shows the generated form for defining a learning task for the Gaussian Mixture Model learner from scikit-learn. The client also includes a demonstration of using the API to compare predictor performance on classification tasks, given the URIs of the label and data attributes and URIs for the predictors to compare. Part of the output from this extension is shown in Figure 4. This illustrates that, while performance evaluation is not explicitly part of the PSI service API, it is supported by interactions between a client and one or more PSI services.

A light-weight Python implementation of PSI is currently in development, and will include demonstrations of rich data transformers for converting various document types into bags of words. An Amazon Machine Image (AMI) of the Java implementation will also be available soon.



Protocols and Structures for Inference — PSI Explorer

poseidon.cecs.anu.edu.au ✖ | Add a service | Compare Predictors

Relations | Transformers | Learners | Predictors | Schema

gmm
j48
kmeans
linear_regression
naive_bayes

Description: Gaussian Mixture Model
URI: http://poseidon.cecs.anu.edu.au/learn/gmm
Task Schema: { ...

Define a learning task

resources

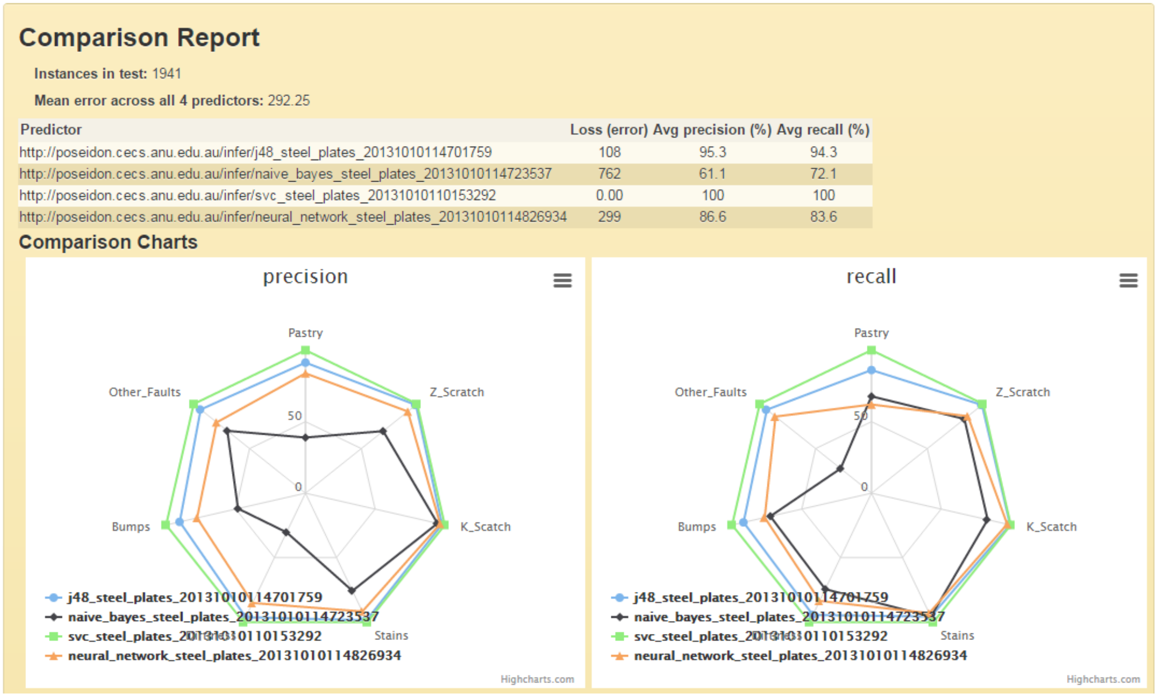
source		Instance details that will be used to determine clusters

parameters

min_covar	0.001	Floor on the diagonal of the covariance matrix to prevent overfitting. (optional)
n_components	1	Number of mixture components. (optional)
covariance_type	diag	The type of covariance parameters to use: 'spherical', 'tied', 'diag' or 'full'. (optional)
threshold		The convergence threshold. (optional)
n_iter		The number of EM iterations to perform. (optional)
n_init		The number of initializations to perform; the best result is kept. (optional)
params	wmc	Which parameters are updated in the training process. Can contain any combination of 'w' for weights, 'm' for means, and 'c' for covars (the default is all three). (optional)
init_params	wmc	Which parameters are updated in the initialization process. Can contain any combination of 'w' for weights, 'm' for means, and 'c' for covars (the default is all three). (optional)

Submit Task | Generate Task Only

Figure 3: The demonstration JavaScript client allows learning tasks to be defined using an HTML form generated entirely from the task schema



Comparison Report

Instances in test: 1941

Mean error across all 4 predictors: 292.25

Predictor	Loss (error)	Avg precision (%)	Avg recall (%)
http://poseidon.cecs.anu.edu.au/infer/j48_steel_plates_20131010114701759	108	95.3	94.3
http://poseidon.cecs.anu.edu.au/infer/naive_bayes_steel_plates_20131010114723537	762	61.1	72.1
http://poseidon.cecs.anu.edu.au/infer/svc_steel_plates_20131010110153292	0.00	100	100
http://poseidon.cecs.anu.edu.au/infer/neural_network_steel_plates_20131010114826934	299	86.6	83.6

Comparison Charts

precision
Pastry
Z_Scratch
K_Scatch
Stains
Bumps
Other_Faults
50
0
j48_steel_plates_20131010114701759
naive_bayes_steel_plates_20131010114723537
svc_steel_plates_20131010110153292
neural_network_steel_plates_20131010114826934
Highcharts.com
recall
Pastry
Z_Scratch
K_Scatch
Stains
Bumps
Other_Faults
50
0
j48_steel_plates_20131010114701759
naive_bayes_steel_plates_20131010114723537
svc_steel_plates_20131010110153292
neural_network_steel_plates_20131010114826934
Highcharts.com

Figure 4: Part of the output from the demonstration JavaScript client's classifier comparison tool. Its inputs are URIs for the testing data attributes and predictors to compare

5. Conclusions & Future Work

A number of extensions are planned for future versions, including: support for encryption and authentication; and the use of compressed JSON formats for large instance representations. We also plan to make available virtual machines images with the code at `https://github.com/orgs/psi-project` and its dependencies pre-installed so that others may easily write PSI-compliant front-ends for their learning algorithms and data sets.

In its present version the PSI RESTful API already offers a general-purpose interface to a range of machine learning activities and may be freely implemented by data, algorithm and predictor providers. It uses schema to define attribute values and learner requirements, and attribute composition to reshape data, both of which can support client software in generating algorithm-specific controls and in composing learning tasks. We envisage a future in which many varied PSI services exist, which we believe would be of benefit to algorithm reuse and evaluation.

Acknowledgments

This research was supported under Australian Research Council's *Linkage Projects* funding scheme (project number LP0991635) in collaboration with Canon Information Systems Research Australia, and by Amazon's AWS in Education Grant award. Mark Reid is also supported by an ARC Discovery Early Career Researcher Award (DE130101605).

References

Janez Demsar and Blaz Zupan. Orange: From experimental machine learning to interactive data mining. White paper, Faculty of Computer and Information Science, University of Ljubljana, 2004. http://www.ailab.si/orange.

Francis Galiegue, Kris Zyp, and Gary Court. JSON schema: core definitions and terminology. IETF draft 04 (work in progress), 2013a. http://tools.ietf.org/html/draft-zyp-json-schema-04.

Francis Galiegue, Kris Zyp, and Gary Court. JSON schema: interactive and non interactive validation. IETF draft 00 (work in progress), 2013b. http://tools.ietf.org/html/draft-fge-json-schema-validation-00.

Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: An update. *SIGKDD Explorations*, 11:10–18, 2009.

Barry Hardy, Nicki Douglas, Christoph Helma, Micha Rautenberg, Nina Jeliazkova, Vedrin Jeliazkov, . . . , and Sylvia Escher. Collaborative development of predictive toxicology applications. *Journal of Cheminformatics*, 2, 2010.

Geraint Luff, Kris Zyp, and Gary Court. JSON hyper-schema: Hypertext definitions for json schema. IETF draft 00 (work in progress), 2013. http://tools.ietf.org/html/draft-luff-json-hyper-schema-00.

Fabian Pedregosa, Gaë Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, . . . , and Édouard Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly, 2007.

Soeren Sonnenburg, Gunnar Raetsch, Sebastian Henschel, Christian Widmer, Jonas Behr, Alexander Zien, . . . , and Vojtech Franc. The SHOGUN machine learning toolbox. *Journal of Machine Learning Research*, 11:1799–1802, 2010.