

2nd International Conference on Predictive APIs and Apps

Volume 50:

PAPIs 2015



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

Preface

These proceedings contain the 4 papers accepted to, and presented at, the research track of the 2nd International Conference on Predictive APIs and Apps (**PAPIs '15**), held in Sydney, Australia on August 7th, 2015.

Predictive Application Programming Interfaces (APIs) are receiving a lot of interest in the industry as they accelerate the development of predictive apps, by making it easier for application developers to use predictive models in production settings. They are a means of exposing predictive models to other programs, and they can also expose model learning capability, in which case one may speak of Machine Learning (ML) APIs. They exist in commercial offerings, such as **Microsoft Azure ML** and **BigML** presented at the research track, and **Amazon ML**, **Datagami** and **Google Prediction API** presented during the industry track, where ML algorithms run on cloud platforms and are accessed “as a service” (MLaaS). Predictive APIs can also be created from open-source or custom frameworks and self-hosted, as demonstrated in the **Upwork** and **PSI** presentations of the research track, but also in the industry track with **Seldon** and **PredictionIO** (at **PAPIs '14**).

MLaaS seeks to abstract away most of the complexity of building and running predictive models at scale, thus making ML easier and quicker to deploy, but also more accessible to developers. In general, exposing models as REST (http) APIs is a way to “package” them so that they can easily be used by desktop, web or mobile developers—who may be using different programming languages and technologies than those used by ML practitioners to build models (e.g. Python, R, Scala, Octave).

The first paper of these proceedings gives us a behind-the-scenes look at Microsoft Azure ML, an MLaaS environment for authoring predictive models, experimenting with them, running them on a cloud infrastructure and publishing them as web APIs. The Azure ML team presents design principles, challenges encountered and lessons learnt while building the platform.

While it is common for ML practitioners to measure models’ performance via predictions’ accuracy, the second paper of these proceedings by Brian Gawalt of Upwork focuses on concerns of software engineers who are in charge of deploying in production and scalability: models’ throughput and response time. Using today’s MLaaS products can help on those aspects, as they scale automatically up to a certain point. But for applications that require low latency and very high throughput (e.g. web services with millions of users), Gawalt offers an open-source API solution that parallelises data retrieval, featurization, learning and predictions, based on the Actor framework of concurrent programming. His solution provides a mid-point between from-scratch concurrent frameworks and mega-scale industry offerings (e.g. Spark), which is particularly suitable for lone data scientists looking to significantly increase predictions throughput.

In the third paper, Montgomery et al. of the PSI project introduce an open standard for REST Machine Learning APIs. One main advantage of exposing training and usage of predictive models via APIs is separation of concerns, but for these APIs to reach their full

potential, they need to be easily interchangeable and interoperable. This would allow app developers, for instance, to test different APIs by just changing their URIs in their code, and to use combinations of the best of the MLaaS and open-source worlds to create complex predictive features in their apps.

Finally, the last paper of these proceedings by Poul Petersen of BigML re-situates Machine Learning APIs in the history of the field, reviews practical challenges in ML usage in industry, identifies new trends in cloud-based machine learning APIs (such as composability and specialization) and outlines the company's vision for the future of these APIs.

Standards, interoperability and composability were also mentioned in Robert Williamson's invited keynote at the conference, and in the panel discussion on research challenges facing Predictive APIs. Other topics discussed were privacy and security, accountability and transparency, automatism (in model selection, detecting data types, data wrangling, text featurization), trust in APIs' models, and alignment with business concerns. We expect to hear more about these topics at future editions of PAPIs!

Acknowledgments

Program Committee: Erick Alphonse, Sébastien Arnaud, Danny Bickson, Misha Bilenko, Christophe Bourguignat, Mikio Braun, Natalino Busa, Sharat Chikkerur, Matthew Grover, Harlan Harris, Jeroen Janssens, Jacek Kowalski, Nicolas Kruchten, Greg Lamp, Gideon Mann, Francisco Martin, Joe McCarthy, Charles Parker, Mark Reid, Thomas Stone, Keiran Thompson, Slater Victoroff, Zygmunt Zajac. **Student Volunteers:** Johann Bengua, Anantashri Boddupalli, Nazanin Borhan, Joan Capdevila Pujol, Alex Cullen, Min Fu, Yannis Ghazouani, Shameek Ghosh, Asso Hamzehei, Andrew Lee, Rongzuo Liu, Jose Magana, Belinda Qin.

In addition to the entire program committee and student volunteers, we would like to thank the Local Chair Gerry Carcour, who took care of local coordination and arrangements for PAPIs '15, and the Media and Publicity Chair Ali Syed. We are grateful for the financial support provided by our corporate sponsors [BigML](#), [NVIDIA](#), [GCS Agile](#) and [Datagami](#). Finally, we wish to thank the authors for their time and effort in creating such a fine program.

February 2016

The Editorial Team:

Louis Dorard, PAPIs '15 General Chair
louis@dorard.me

Mark D. Reid, PAPIs '15 Research Chair
Australian National University
mark.reid@anu.edu.au

Francisco J. Martin, PAPIs '15 Program Chair
BigML
martin@bigml.com

Table of Contents

Introduction

<i>AzureML: Anatomy of a machine learning service</i> A.T.M. ; JMLR W&CP 50: 1–13 , 2016.	1
<i>Deploying High-Throughput, Low-Latency Predictive Models with the Actor Framework</i> B. Gawalt; JMLR W&CP 50: 15–28 , 2016.	15
<i>Protocols and Structures for Inference: A RESTful API for Machine Learning</i> J. Montgomery, M.D. Reid & B. Drake; JMLR W&CP 50: 29–42 , 2016.	29
<i>The Past, Present, and Future of Machine Learning APIs</i> A. Cetinsoy, F.J. Martin, J.A. Ortega & P. Petersen; JMLR W&CP 50: 43–49 , 2016.	43

AzureML: Anatomy of a machine learning service

AzureML Team, Microsoft *

1 Memorial Drive, Cambridge, MA 02142

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

Abstract

In this paper, we describe AzureML, a web service that provides a model authoring environment where data scientists can create machine learning models and publish them easily (<http://www.azure.com/ml>). In addition, AzureML provides several distinguishing features. These include: (a) collaboration, (b) versioning, (c) visual workflows, (d) external language support, (e) push-button operationalization, (f) monetization and (g) service tiers. We outline the system overview, design principles and lessons learned in building such a system.

Keywords: Machine learning, predictive API

1. Introduction

With the rise of big-data, machine learning has moved from being an academic interest to providing competitive advantage to data driven companies. Previously, building such predictive systems required deep expertise in machine learning as well as scalable software engineering. Recent trends have democratized machine learning and made building predictive applications far easier. Firstly, the emergence of machine learning libraries such as scikit-learn [Pedregosa et al. \(2011\)](#), WEKA [Holmes et al. \(1994\)](#), Vowpal Wabbit [Langford et al. \(2007\)](#) etc. and open source languages like R [R Development Core Team \(2011\)](#) and Julia [Bezanson et al. \(2012\)](#) have made building machine learning models easier even without machine learning expertise. Secondly, scalable data frameworks [Zaharia et al. \(2010\)](#); [Low et al. \(2010\)](#) have provided necessary infrastructure for handling large amounts of data. Operationalizing models built this way still remains a challenge. Many web services (e.g. <http://dataiku.com>, <http://bigml.com>, <http://cloud.google.com/prediction>, <http://datarobot.com> etc.) have attempted to address this problem providing turnkey solutions that enable software developers and data scientists to build predictive applications without requiring deep machine learning or distributed computing expertise. In this paper, we describe AzureML, a web service that provides a model authoring environment where data scientists can create machine learning models and publish them easily (<http://www.azure.com/ml>). In addition, AzureML provides several distinguishing features. These include:

1. Collaboration: AzureML provides the ability to share a workspace of related modeling workflows (called experiments) with other data scientists. Experiments can also be shared with the wider community through an experiment gallery. Experiments in

* Corresponding authors: sudarshan.raghunathan@microsoft.com, sharat@alum.mit.edu (formerly at Microsoft)

the gallery are indexed and searchable. Discovery is enabled by ranking experiments organically based on popularity and community rating.

2. Versioning: AzureML provides the ability to save the state of individual experiments. Each experiment has a version history. This enables rapid iteration without the overhead of tracking and maintaining individual experiments and different parameter settings.
3. Visual workflows: Data workflows can be authored and manipulated in a graphical environment that is more natural and intuitive than scripts. An experiment graph can be composed by joining modules (functional blocks that provide a way to package scripts, algorithms and data manipulation routines).
4. External language support: AzureML supports packaging scripts and algorithms in external languages such as python and R as modules. These modules can be inserted into any experiment graph along with built-in modules. This allows data scientists to leverage existing code along with built-in modules provided by AzureML.
5. Push button operationalization: Experiments containing models and data manipulation workflow can be operationalized into a web service with minimal user intervention. In contrast to other web-services, the prediction service can consist of data pre and post-processing. Operationally, each prediction corresponds to an experiment graph run in memory on a single instance or a groups of instances.
6. Monetization: AzureML provides a market place where authors of models can monetize their published data or prediction services. This opens up new opportunities for modelers in the prediction economy.
7. Service tiers: AzureML currently exposes two service tiers, paid and free. Anyone with a liveID can create a workspace and begin to author and run experiments. These experiments are limited in the size of data that can be manipulated, the size of experiments that can be created, and the running time of these experiments. In addition, free users cannot operationalize models to a production level of service. Paid users have no such limits. In the rest of this paper, we outline the system overview, design principles and lessons learned in building such a system.

2. System Overview

AzureML consists of several loosely coupled services (see Figure 2) which are described in the following.

2.1. Studio (UX)

This is the primary user interface layer that provides tools for authoring experiments Studio provides a palette of available modules, saved user assets (models, datasets) and provides a way to define and manipulate experiment graphs. Studio also provides UX necessary for uploading assets, sharing experiments and converting experiments to published web services. The primary document type within AzureML is an experiment. Operationally, an

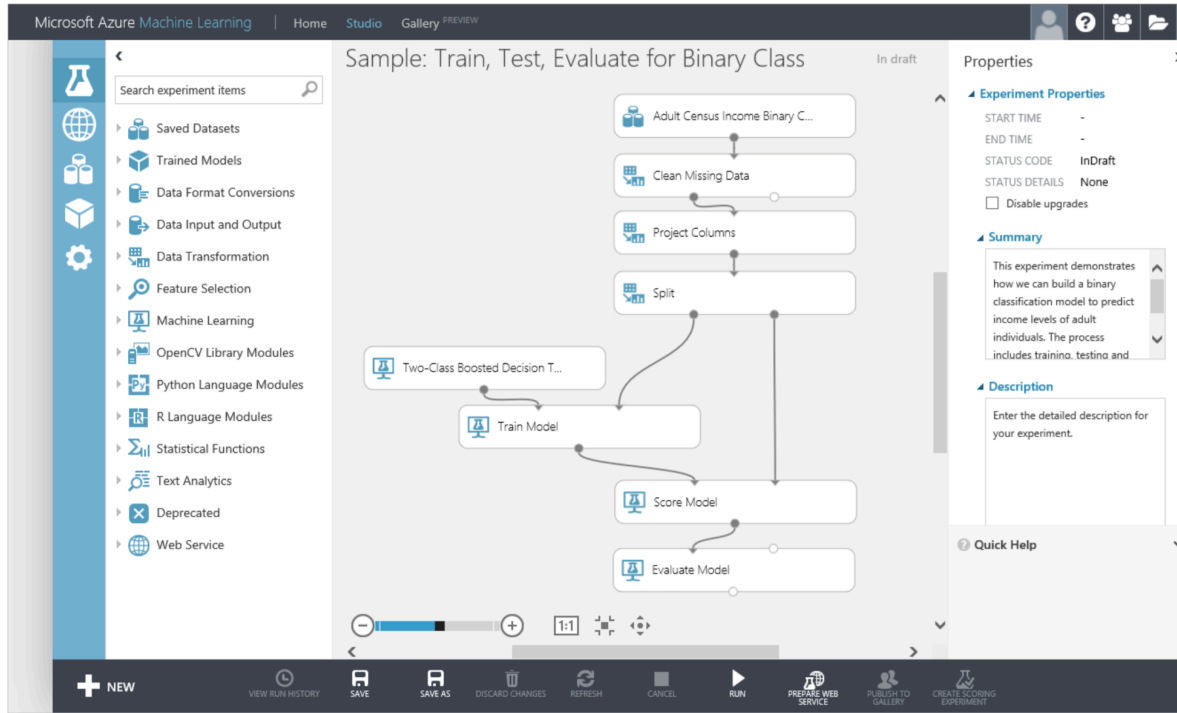


Figure 1: AzureML UX showing an example experimental graph and the module palette

experiment consists of a directed acyclic graph connecting several modules. Modules encapsulate data, machine learning algorithms, data transformation routines, saved models and user-defined code. Modules are divided into many categories such as machine learning, data manipulation, text analytics etc. Such modules can be referenced from multiple experiments and web services.

2.2. Experimentation Service (ES)

ES is the primary backend service that orchestrates interaction among all component services. ES is responsible for handling and dispatching UX events from Studio and communicating the results of experiments back to the user. Interaction between Studio and ES are defined by strong contracts allowing Studio and ES to be developed iteratively and independently. In addition, ES is the repository for all assets (datasets, models, transforms) and information about users and workspaces. Encapsulating all experiment related activities in a service that exposes an API allows programmatic creation, migration and manipulation of experiments. This allows other services to be built on top of AzureML. Internally, we use

this API for end to end tests, runners, and validation of individual modules. The API also enables AzureML datasets to be manipulated by external environments. A Python SDK is available where datasets can be imported (with an auto-generated code snippet), explored, and saved back to AzureML workspaces.

2.3. Job Execution Service (JES)

When an experiment is submitted for execution by the Studio, ES constructs a job consisting of individual module executions and sends it to JES. JES is the scheduler that dispatches these module executions to our execution cluster. Within a workspace, multiple experiments can be scheduled in parallel. Within each experiment, the experiment graph establishes dependencies among individual module invocations. Modules whose dependencies have been satisfied can be executed in parallel. The JES only schedules individual modules and does not execute the task itself. The individual tasks are queued and eventually gets executed on a worker node (SNR). The JES is responsible for tracking the execution of tasks and scheduling downstream modules when results of individual modules are available. In addition, JES enforces resource limits on executions. Each workspace can only run a certain number of jobs in parallel and each job has a limit on the number of tasks that can be executed concurrently. These limits are established to maintain fairness among the different users and also to differentiate among our different tiers of service.

2.4. Singe Node Runtime (SNR)

Tasks from JES are pushed on to a task queue from where SNRs pick them up for execution. Each module task can potentially get scheduled on a separate machine. Therefore the task and resources required for execution of a module are copied over locally to each SNR. Their stateless nature allows us to scale the number of SNRs based on the experiment workload. However, this design also adds some overhead associated with transporting data, dependent and assemblies to the SNR prior to task execution. This overhead can be amortized by careful JES scheduling where adjacent tasks on the graph get executed on the same SNR.

2.5. Request Response Service (RRS)

RRS is responsible for answering prediction requests for single input. It implements web services that are published based on experiments authored in Studio. A distinguishing feature of AzureML is that prediction/scoring workflow can be represented and manipulated as an experiment graph. The presence of web input/output entry points are the only differentiating factors between training and scoring workflows. This implies that that prediction may involve a subset or complete experiment graph including all data transformation modules. A training graph can be transformed to a scoring/prediction graph based using simple rules (which are in-fact automated). Some examples are: (a) replacing model training with a pre-trained and serialized model (b) bypassing data splits in the module (c) replacing input data sets with a web-input port etc. Another difference between training and scoring experiments is that within RRS, modules are executed in memory and on the same machine. Predictions can be made per instance or on a batch basis. When experiments are published to the RRS, test endpoints are created along with an autogenerated help page for the API. This help page describes the format of the requests that are accepted by the endpoint as well as

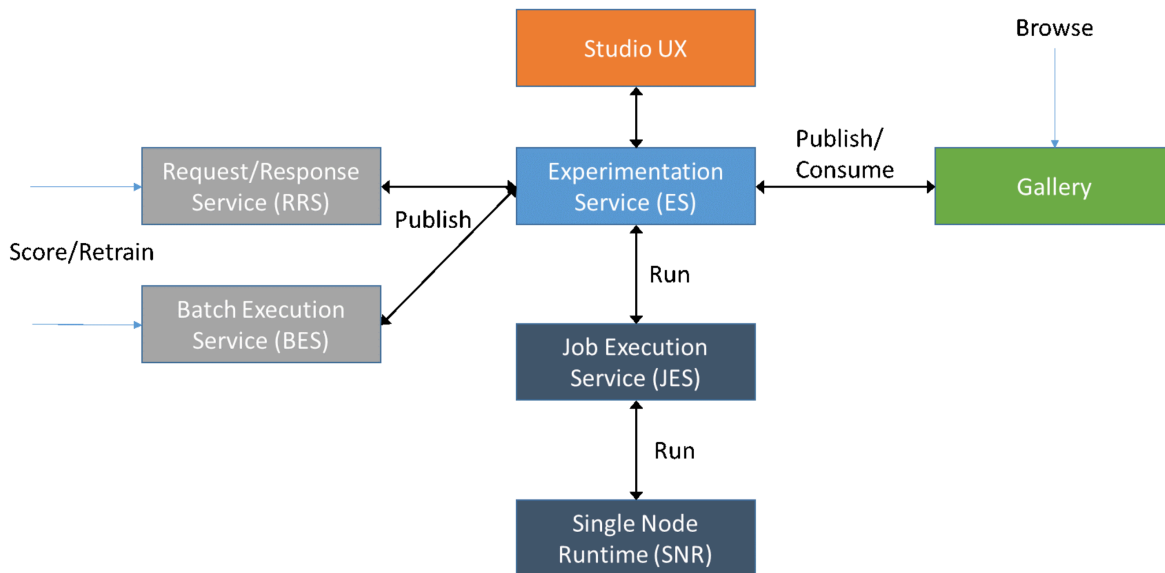


Figure 2: Component services of AzureML

sample code in C#, Python, and R for consuming the endpoints. Each RRS call is stateless and therefore can be replicated on multiple machines to scale with demand. We recently introduced a retraining API where customers can programmatically re-run their training experiments on new data, evaluate the resulting models, and update their web services with the new models if they are satisfactory.

2.6. Batch Execution Service (BES)

BES provides prediction services on larger datasets in contrast to RRS that provides prediction for single inputs. The execution of the graph within BES resembles that of a training workflow.

2.7. Community/Gallery

Authors of experiments can share their workspaces with individual users who can either edit or only view the experiments. In addition, AzureML also provides a gallery where

users can share experiments with all AzureML users. Gallery effectively provides a set of templates that can serve as starting point for experiments. Gallery also supports discovery of experiments through organic ranking as well as through a searchable index. Gallery is available to browse for free on the web and experiments saved there can be imported into any AzureML workspace.

2.8. Marketplace

Marketplace (<http://datamarket.azure.com>) allows users to monetize their published experiments or data. Currently, the marketplace supports web services that were generated using AzureML experiments as well as standalone web services authored externally. These end points can be integrated into regular experiments allowing composition of functionalities.

3. Life of an experiment

In the following, we describe the typical (classification/regression) workflow of an AzureML experiment.

3.1. Ingress

Data can be imported into AzureML in several different ways uploading CSV/TSV/ARFF files, referencing a URL, Azure blob storage, SQL server etc. During ingress, schema from these various formats are reconciled into an internal format (.dataset that will be discussed later). In cases where input schema does not specify the data type of individual columns (e.g. CSV), the types are guessed by looking at the composition of the data. AzureML also provides mechanism to explicitly change data types after ingress.

3.2. Data manipulation

AzureML provides several mechanisms for manipulating data prior to training/predictions.

- Data transformation modules: AzureMLs palette provides modules such as Clean Missing Values Join Columns, Project Columns, SQL transformation etc to allow manipulation of data.
- R/Python scripts: Arbitrary R and python scripts can be packaged as modules to transform the data. AzureML handles marshalling of data and schema.

3.3. Modeling

AzureML exposes both open-source as well algorithms developed at Microsoft to model data. Table 1 provides a list of algorithms that is continuously expanding.

3.4. Parameter tuning and evaluation

Practical machine learning involves some art, those of choosing the appropriate model, algorithm and finally their parameters. Some recent advances [Bergstra et al. \(2011\)](#); [Snoek et al. \(2012\)](#) have automated the task of model selection and parameter optimization. AzureML

Table 1: List of algorithms supported in AzureML

Binary classification	Average perceptron Bayes point machine Boosted decision tree Decision jungle Locally deep SVM Logistic regression Neural network Online SVM Vowpal wabbit	Freund and Schapire (1999) Herbrich et al. (2001) Burges (2010) Shotton et al. (2013) Jose and Goyal (2013) Duda et al. (2000) Bishop (1995) Sha (2011) Langford et al. (2007)
Multiclass classification	Decision Forest Decision jungle Multinomial regression Neural network One-vs-all Vowpal wabbit	Criminisi (2011) Shotton et al. (2013) Andrew and Gao (2007) Bishop (1995) Rifkin (2002) Langford et al. (2007)
Regression	Bayesian linear regression Boosted decision tree regression Linear regression (batch and online) Decision forest regression Random forest based quantile regression Neural network Ordinal regression Poisson regression	Herbrich et al. (2001) Burges (2010) Bottou (2010) Criminisi (2011) Criminisi (2011) Bishop (1995) McCullagh (1988) Nelder and Wedderburn (1972)
Recommendation	Matchbox recommender	Stern et al. (2009)
Clustering	K-means clustering	Jain (2010)
Anomaly detection	One class SVM PCA-based anomaly detection	Schölkopf and Williamsonx Duda et al. (2000)
Feature selection	Filter based feature selection Permutation feature importance	Guyon et al. (2003) Breiman (2001)
Topic modeling	Online LDA using Vowpal Wabbit	Hoffman et al. (2010)

provides a sampling based approach to optimization where a grid of parameters can be fully explored or sampled randomly to optimize metrics. Random sampling is typically more cost efficient than grid search in practice.

3.5. Operationalization

In AzureML, operationalization refers to the procedure of taking a trained model along with associated data transformations and converting it to a web service that can be queried for individual or bulk predictions. The primary challenge in operationalization is optimizing for latency and ability to handle large throughput. Some services export the learned model to a popular format (e.g. PMML [Guazzelli et al. \(2009\)](#)) and yield the choice of operationalization to the model owner. In contrast, AzureML manages the model for the user. The models are replicated based on the volume of prediction requests. Further, AzureML supports machine learning models that cannot be fully expressed in standards like PMML.

3.6. Modules

In AzureML, modules are the individual components in an experiment graph. All machine learning algorithms, data processing routines and custom code are packaged as modules. Each module may consist of one or more input and output ports. Each port has an assigned type (example dataset, ilearner, itransform etc.) which determines how modules are connected.

3.7. Graph Execution

The experiment graph is a directed acyclic graph that also defines the data dependencies between the modules. Modules that do not depend on each other can be executed in parallel. The job execution service (JES) is responsible for this functionality.

3.8. Schema validation

Datasets in AzureML are tagged with meta-data. In addition to column names and column types as in R and python, meta-data in AzureML attaches semantic meaning to individual columns. For instance, the meta-data information keeps track of columns that are used for training vs. those generated by predictions. This schema is propagated across the experiment graph even before the results are computed. This is essential in validating and early reporting of data compatibility. A module missing certain columns can report an error even before the upstream computation can take place.

3.9. External language support

AzureML support mixing of built-in modules as well as custom modules. Custom modules are user defined code that is packaged as a module. Currently, both R and Python language modules are supported. Data is marshalled in and out at the module boundary along with the associated metadata. Security is a primary consideration when allowing user code within a managed environment. Within AzureML, these modules are executed within a sandbox environment that limits system level operations.

3.10. Experiments as documents

AzureML in addition to being a modeling environment is foremost an authoring environment. Experiments are treated as assets of a data science process that need to be tracked, versioned and shared. Each run of an experiment is recorded and saved. Results from previous experiments are cached. If a section of the graph consists of deterministic modules, the modules are re-run only if upstream outputs are changed. Otherwise, cached results are returned immediately. Previous runs of an experiments are frozen can be accessed and copied over as new experiments.

4. Implementation details

4.1. Data types

AzureML experiments can be viewed as transformations on a restricted set of data types. Modules take as input and produce as output objects of the following types:

- DataSets: These are typically CSV, TSV, ARFF, and files in our own proprietary format
- Models: Trained and untrained models such as SVMs, Decision Trees, and Neural Networks.
- Transforms: Saved transformations that can be applied to data sets. Examples include feature normalization, PCA transforms etc.

Because the inputs and outputs to modules are typed, the UX can quickly determine the legality of connections between modules and disallow incompatible ones at experiment authoring time.

4.2. Data representation

After ingress, data is serialized in a proprietary .dataset format. The file is organized as schema, an index, and a linked list of 2D tiles, each tile containing a rectangular subset of data. This flexible representation allows us to decompose at both row and column level granularity. Wide slices are more suitable for streaming where as thin-and-long slices are better suited for compression and column manipulation. The tiled representation allows us to deliver good I/O performance across different access patterns streaming, batch loading into memory, and random access. Conversion modules to other formats (ARFF/CSV/TSV) are provided so that data can be egressed from our system in readable formats.

4.3. Modules

Modules are described by a formal interface what data types they accept and produce, and their parameters. Parameters also have types and are organized into parameter trees where certain parameters are only active if others are set to particular values. These descriptions are loaded into the Studio UX in order to present a rich authoring and viewing experience. Because we desire to support a multiplicity of data set formats (CSV/TSV/ARFF/.dataset and others in the future) we decided that module writers should not be responsible for I/O,

but rather should code against a data table interface. This interface, which includes access to both data and schema takes the place of datasets in our code. When modules are executed, the system converts the input file to an object implementing the data table interface and passes this object to the module code. When module is execution is finished, the system performs the reverse process it saves the object to a .dataset file and in addition creates separate schema and visualization files for consumption by the Studio UX.

4.4. External languages

AzureML supports a growing list of external languages including R and Python. We provide Execute R and Execute Python modules where users can enter scripts for manipulating data. Data internal to AzureML is marshalled as R and Pandas data-frames respectively. AzureML thus provides the ability to compose a graph consisting of built-in as well as user defined modules. In addition to supporting generic Execute Script modules for various external languages, users also have the ability to define their own module implementations in external languages. While the interface of, say, the Execute R module is static, a custom module allows for a writer to fully define an interface, generating a black-box implementation of routines in the external language of their choice. These custom modules interoperate with production modules and can be dragged in to experiments from the module palette like any other firstclass module.

4.5. Testing

All aspects of the product are extensively tested; ranging from usability sessions, mock security attacks and integration testing to an extensive automated test framework. The automated framework includes correctness testing for algorithms, fuzzing techniques to detect boundary cases, end-to-end tests that ensure that the various system parts integrate well together, etc. These tests both gate source code changes as well as act to detect regressions post acceptance.

5. Lessons learned

Implementating a general machine learning web service required several design/performance tradeoffs with a bias towards usability. The subsequent use of the system provided us certain lessons outlined below.

5.1. Data wrangling support is useful

One of the key challenges in data science is data manipulation prior to building models. In fact, it seems to take more effort than building models (<http://nyti.ms/1t8IzfE>). This is reflected in our internal metrics that show that data-manipulation modules form majority of experiment graphs. AzureML provides a rich library of data manipulation modules including specific functionality (e.g. split, join etc.) and also very general modules such as (e.g. SQL transform).

5.2. Make big data possible and small data efficient

It is very tempting to adopt distributed frameworks prior to evaluating its tradeoffs. It has been shown that mapreduce style computing is inefficient for machine learning leading to in-memory solutions such as Spark [Zaharia et al. \(2010\)](#). A typical data science workflow involves rapid exploration and experimentation using a small data set in a development environment followed by productizing the module using large amounts of data. AzureML spans both ends of the size dimension. It makes big-data possible but small data efficient. This is a fluid position that will evolve over time.

5.3. Reproducibility is important, but expensive

AzureML provides the notion of deterministic and nondeterministic modules. Workflows containing only deterministic components is guaranteed to be reproducible across multiple runs. This immutability of module output allows us to cache intermediate results between experiment runs. Further, when executing an experiment graph, only modules downstream of a modified module need to be re-executed. Other results are available immediately due to guarantees of deterministic behavior. Caching allows a data-scientists to rapidly iteration on an experiment. From an implementation perspective this implies that intermediate results have to be serialized during graph execution. Further downstream modules cannot be scheduled until upstream results are serialized. This adds a large I/O overhead to the run-time of an experiment compared to a pipeline or deferred evaluation style execution (e.g. Apache Spark). The design assumes that the amortized cost over many variants will be minimized with checkpoints.

5.4. Feature gaps are inevitable but can be mitigated by user code

AzureML provides a rich set of algorithms and data transformation. The available functionality is expanded on a continuous basis. Users may be drawn to AzureML in order to utilize the authoring environment and versioning capabilities but may not find specific algorithms available in the palette. Because AzureML allows user to package custom code as modules, required functionality can be easily added while feature gaps are eventually closed by the AzureML. This allows users to leverage the rich set of libraries provided by R and Python along with the authoring functionalities provided by AzureML.

6. Conclusion

In this paper, we described AzureML, a web service that allows easy development of predictive models and APIs. Compared to other machine learning web services, AzureML provides several distinguishing features such as versioning, collaboration, easy operationalization and integration of user-code. The paper describes the design of the overall system as well as trade-offs that were made to implement this functionality. Finally, the paper describes some lessons learned that could prove useful for other providers of machine learning services.

References

- Pegasos: Primal estimated sub-gradient solver for SVM. *Mathematical Programming*, 127 (1):3–30, 2011.
- Galen Andrew and Jianfeng Gao. Scalable training of L1 -regularized log-linear models. In *Proceedings of the 24th international conference on Machine learning - ICML '07*, pages 33–40, 2007.
- James Bergstra, Rémi Bardenet, Y Bengio, and B Kégl. Algorithms for Hyper-Parameter Optimization. In *Advances in Neural Information Processing Systems*, pages 2546–2554, 2011.
- Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A Fast Dynamic Language for Technical Computing. *arXiv preprint*, pages 1–27, 2012. ISSN <null>.
- C M Bishop. *Neural Networks for Pattern Recognition*, volume 92. 1995.
- León Bottou. Large-Scale Machine Learning with Stochastic Gradient Descent. *Proceedings of COMPSTAT'2010*, pages 177–186, 2010.
- L Breiman. Random forests. *Machine learning*, 2001. doi: 10.1023/A:1010933404324.
- Christopher J C Burges. From ranknet to lambdarank to lambdamart: An overview. *Learning*, 11:23–581, 2010. doi: 10.1111/j.1467-8535.2010.01085.x.
- Antonio Criminisi. Decision Forests: A Unified Framework for Classification, Regression, Density Estimation, Manifold Learning and Semi-Supervised Learning, 2011.
- R O Duda, P E Hart, and D G Stork. *Pattern Classification*. 2000. doi: 10.1038/npp.2011.9.
- Yoav Freund and Robert E. Schapire. Large margin classification using the perceptron algorithm. *Machine Learning*, 37(3):277–296, 1999.
- Alex Guazzelli, Michael Zeller, Wc Lin, and G Williams. PMML: An open standard for sharing models. *The R Journal*, 2009.
- I Guyon, I Guyon, A Elisseeff, and A Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, 2003.
- Ralf Herbrich, Thore Graepel, and Colin Campbell. Bayes Point Machines. *Journal of Machine Learning Research*, 1:245–279, 2001. doi: 10.1162/153244301753683717.
- Matthew D Hoffman, David M Blei, and Francis Bach. Online Learning for Latent Dirichlet Allocation. *Advances in Neural Information Processing Systems*, 23:1–9, 2010.
- Geoffrey Holmes, Andrew Donkin, and Ian H Witten. Weka: A machine learning workbench. In *Intelligent Information Systems, 1994*, pages 357–361. IEEE, 1994.
- Anil K. Jain. Data clustering: 50 years beyond K-means. *Pattern Recognition Letters*, 31 (8):651–666, 2010.

- C Jose and P Goyal. Local deep kernel learning for efficient non-linear SVM prediction. *Proceedings of the 30th International Conference on Machine Learning*, 2013.
- J Langford, L Li, and A Strehl. Vowpal wabbit online learning project, 2007.
- Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. GraphLab: A New Framework for Parallel Machine Learning. *The 26th Conference on Uncertainty in Artificial Intelligence (UAI 2010)*, pages 8–11, 2010. doi: 10.1.1.167.6156.
- Peter McCullagh. Regression models for ordinal data. *Journal of the royal statistical society. Series B (Methodological)*, pages 109–142, 1988.
- J A Nelder and R W M Wedderburn. Generalized Linear Models. *Journal of the Royal Statistical Society. Series A (General)*, 135(3):370–384, 1972.
- Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, and Others. Scikit-learn: Machine learning in Python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.
- R R Development Core Team. R: A Language and Environment for Statistical Computing, 2011.
- Ryan Michael Rifkin. Everything Old Is New Again: A Fresh Look At Historical Approaches in Machine Learning. pages 1–221, 2002.
- B Schölkopf and R Williamsonx. Support Vector Method for Novelty Detection. *alex.smola.org*.
- Jamie Shotton, Toby Sharp, and Pushmeet Kohli. Decision Jungles: Compact and Rich Models for Classification. *Advances in Neural Information Processing Systems*, pages 1–9, 2013.
- Jasper Snoek, Hugo Larochelle, and Ryan P. R. Adams. Practical Bayesian Optimization of Machine Learning Algorithms. *arXiv preprint arXiv:1206.2944*, pages 1–12, 2012.
- David Stern, Ralf Herbrich, and Thore Graepel. Matchbox : Large Scale Online Bayesian Recommendations. In *Proceedings of the 18th International Conference on World Wide Web (WWW’09)*, pages 111–120, 2009.
- Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark : Cluster Computing with Working Sets. In *HotCloud’10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, page 10, 2010. doi: 10.1007/s00256-009-0861-0.

Deploying High-Throughput, Low-Latency Predictive Models with the Actor Framework

Brian Gawalt

BGAWALT@{UPWORK, GMAIL}.COM

Upwork, Inc.

441 Logue Ave.

Mountain View, CA 94043, USA

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

Abstract

The majority of data science and machine learning tutorials focus on generating models: assembling a dataset; splitting the data into training, validation, and testing subsets; building the model; and demonstrating its generalizability. But when it's time to repeat the analogous steps when using the model in production, issues of high latency or low throughput can arise. To an end user, the cost of too much time spent featurizing raw data and evaluating a model over features can wind up erasing any gains a smarter prediction can offer.

Exposing concurrency in these model-usage steps, and then capitalizing on that concurrency, can improve throughput. This paper describes how the actor framework can be used to bring a predictive model to a real-time setting. Two case-study examples are described: a live deployment built for the freelancing platform Upwork, a simple text classifier with accompanying code for use as an introductory project.

1. The practice of machine learning

Imagine a firm has brought a specialist in machine learning onto a new project. The firm wants a product which can provide a quality prediction about some regular event happening in the course of the firm's business. The specialist is handed a pile of relevant historical data, and asked: Among the new customers seen for the first time today, who's likeliest to be a big spender? Or: of all the credit card transactions processed in the last hour, which are likeliest to be fraudulent? Or: when a customer enters a query into our website's Search tool, what results should we be returning?

The specialist starts with the first of two phases of their project. They have to identify a model that can be expected to fit predictions over both the historical data and in a way that will generalize to new data. The familiar version of the steps involved in supervised learning:

1. Identify raw source data, and break it down into distinct observations of the pattern you're trying to learn and predict.
2. For each raw observation, produce a p -dimensional vector of features and a scalar label.
3. Split this collection into disjoint training, validation, and testing sets.

4. For each candidate model (and/or each of the model’s hyperparameters), fit model parameters to the training vectors and labels, and evaluate the goodness of fit by performing prediction of the validation labels given the validation vectors
5. Select the model whose validation-set predictions came closest to the mark. Use it to then make predictions over the test set. Report this test set performance to vouch for the predictive model you’ve generated and selected.

Note that this first phase doesn’t carry an explicit component of *time urgency*. All else equal, a typical specialist will prefer that the full sequence complete in six hours, and six minutes is better still. But if it takes six days instead, nothing *fundamental* to this first phase has been threatened. The task – finding a model that generates acceptably accurate predictions on new data – is accomplished.

The second phase is to actually put the model’s capabilities to use. Given new events and observation that need scoring by the model – is this new customer likely to be a big spender? is this credit card legitimate? – the above featurization and scoring routines need to be run. And in this real-world deployment, it’s likely that there are also some strict constraints on how long it takes this sequence to run. All predictions go stale, and some use cases need to act on a prediction within milliseconds of the event itself.

There are some cases where these latency constraints aren’t binding. The exact same featurization and scoring routines used to generate and validate the model can be re-run fast enough on new data to produce useful predictions. But this paper focuses on the cases where timeliness requirements exclude the use of the same software developed in the first phase as the backbone of the second phase. What can a lone machine learning specialist do to retool their sequence to run in a production environment?

1.1. Moving to production

If the original software, used to generate and validate the predictive model, is suffering from too-low throughput in producing new predictions, one path forward could be to incorporate more concurrent processing. The three steps to prediction (gathering raw materials, featurizing those materials into vectors, scoring the vectors) can transition from a serial sequence to a pipeline.

Figure 1 demonstrates a modification of the scoring task flow, producing predictions of N events in a sequential and a concurrent pattern. This pipeline offers a few advantages. Scores are produced with less delay after the raw material gathering (useful in case the information in that material is at risk of going stale or out of date).

Most importantly, this redesign provides a clear path forward to speed-up in completing all N scoring tasks. If a system can genuinely perform the concurrent tasks in parallel, as a multicore system might, one can easily picture adding “clones” of this pipeline simultaneously processing more and more partitions of the N events.

1.2. Complexity costs

It can be difficult to put together, from scratch, a high-performance concurrent computing system. It’s easy to fall into traps of false sharing, deadlocking, and other difficulties. It’s

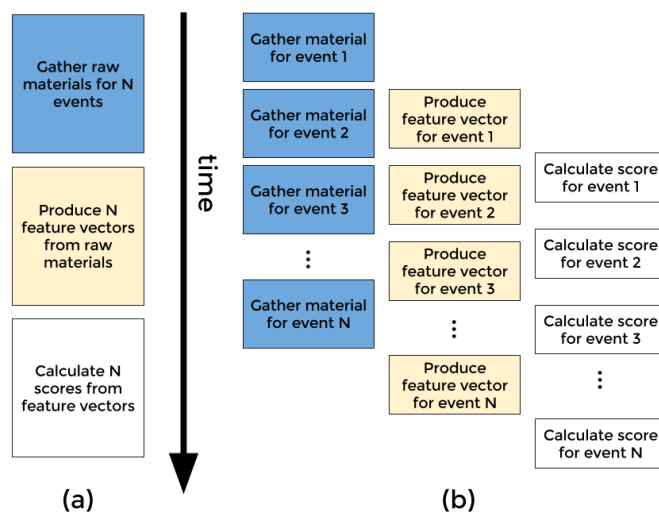


Figure 1: (a) The scoring procedure performed serially. (b) The individual tasks for each event to be scored performed in a concurrent pipeline.

not impossible, but it’s definitely tricky and time-consuming, and building a new concurrent system for a single project might fail a cost-to-benefits ratio test.

Fortunately, lots of great work has produced platforms to take this plan to a truly wide scale implementation. Apache Storm ([The Apache Software Foundation, 2015b](#)), Apache Spark (especially its Spark Streaming library) ([The Apache Software Foundation, 2015a](#)), and Twitter’s Heron ([Kulkarni et al., 2015](#)) all try to distribute this kind of event-driven computing across multiple machines to ensure the highest possible throughput.

Unfortunately, they’re complicated systems. Familiarizing oneself with the API, and managing the underlying infrastructure, requires considerably more expertise above and beyond that of our original model in Figure 1(a). If spreading the load over multiple machines is the only way to meet the required service level of the prediction system, this additional complexity will have to be met with additional resources: maybe the firm will have to start allocating more engineers in addition to more hardware, to what originally was a project of just the single machine learning specialist.

This paper is here to specifically recommend a midpoint between a from-scratch concurrent framework and the mega-scale offerings. The actor framework offers a simple way to reason through concurrent problems, while still being flexible enough to accommodate a variety of approaches to any one problem.

From here, this paper presents a case study where an actor framework was key to bringing a predictive model to production. It interleaves this story with a summary of what an actor framework entails, as well as a description of the framework implementation in the Scala library Akka. It concludes with a reiteration of the advantages (and disadvantages) the actor framework offers and a discussion of why comfort with actors can help machine learning specialists with their work and careers.

2. Case Study: Predicting worker availability at Upwork

Upwork is an online labor platform, connecting freelancers and clients for internet-enabled remote work. There are several patterns for how these work arrangements are struck¹, but one important avenue involves a client using the platform’s search engine to discover new freelancers they think would be well suited to the project the client has in mind.

When the client issues a query, e.g., “java developer”, “restaurant menu design”, “paralegal”, they expect to see a result set filled with freelancers with relevant experience who are likely to perform well on a new assignment. The client can then invite any of these freelancers to interview for an open position.²

This adds a third requirement to the query’s result set: listed freelancers should not only be relevant and high-performing, they should also be receptive to an interview invitation at the time of the query. If the system returns a list of the same ten excellent freelancers for every search for “wordpress template,” it’s likely that those ten freelancers will wind up deluged with opportunities they’re too busy to fill, and for the freelancers ranked just outside that top ten being disproportionately starved for job invitations.

There was an existing heuristic in place to try and capture this, but it was felt a learned model trained on historical data could easily provide higher accuracy.

2.1. Building the predictive model

If the system should only show freelancers available to interview, how should we estimate freelancer availability? This question can be gently reformulated into a question of binary classification: “If Freelancer X were to receive an invitation to interview right now, does their recent use of the site suggest they would accept that invitation, or would they instead reject or ignore it?”

A logistic regression model can help answer this question. Each invitation sent from a client to a freelancer was treated as an example event, labeled as a positive class member if it was accepted and negative otherwise. (There were roughly one million such examples within a relevant and recent time frame.) The goal would be to link the freelancer’s recent site activity – within the month prior to receiving the invitation – to the outcome of an acceptance or rejection of the invite.

The raw materials, and derived features, came in four main varieties:

Job application/invitation history In the previous day, how many job applications did Freelancer X create? How many invitations did X receive? How many of each were cancelled by the client, or by the freelancer? How many of each led to filled jobs? Answer these questions again for the time periods: two days, four days, seven days, 14 days, and 30 days.

-
1. Freelancers can send proposals for a client’s publicly posted job opening; groups of freelancers form agencies to share many client workloads; clients and freelancers, already working together outside the platform, moving their relationship onto Upwork to take advantage of the site’s payment processing, escrow, or other services
 2. Upwork sees the two actions happen in both orders. Sometimes, clients post a job opening, then go searching for freelancers to invite to interview; other times, the client starts by searching and browsing freelancer profiles, and creates the job post once they see someone they’d like to work with.

Hours worked How many hours has Freelancer X billed to any client in the preceding day? two days? ... thirty days?

Server log history In the previous one/two/.../thirty days, how many times has Freelancer X visited pages under the “Job Search,” “Message Center,” and “View Contractor Profile” sections of Upwork.com?

User profile How many jobs has the freelancer worked in each of Upwork’s job categories (e.g., Web Development, Graphic Design, Data Entry)? What is their stated preference for receiving more invitations at this time (“I am available,” vs. “I’m not currently available”)?

These raw materials (along with a few other sources) could be harvested from three services: a Greenplum relational database for assembling job application and hours-worked data, and an Amazon S3 bucket for the server logs. The user-profile information was an interesting case: the historical state of the profile was logged to a table in Greenplum with each invitation, but a freelancer’s present profile state required a call to an internal service’s REST API.

Assembling these feature vectors and performing a training-validation-testing split led to a model with a test-set AUC metric of around 0.81. This sufficiently outperformed the existing heuristic’s accuracy, and the model was deemed ready for production use. Freelancers would start receiving scores from the model, putting them into buckets of low, medium, and high availability (i.e., high propensity to accept an invitation to interview at this time).

2.2. Throughput constraints in production

An interesting aspect of this modeling task is that hour to hour, any individual freelancer’s availability score might swing considerably. Someone who looked dormant three hours ago could decide to log back onto Upwork and start engaging in eager-for-work behavior, sending out job applications and polishing their profile. Another freelancer’s behavior might cause a sudden transition from looks-available to looks-overbooked. The more frequently each freelancer could be scored, the more faith Upwork could have that the bucketing used by the search engine was using reliable information. Keeping freshness of the raw material data under four hours was considered a reasonable goal.

Generating those scores meant providing the routine with those four families of raw material. When developing the model using historical data, all four could be gathered and processed in bulk, in advance of the featurization and scoring routines. In a production setting, this remained the case for data from S3 and Greenplum: all that relevant information, for all registered freelancers, could be collected in under an hour.

However, user profiles posed a challenge: given other demands placed on it by other Upwork.com systems, the internal user profile service could only afford to return one user profile every 20 milliseconds to the availability scorer. Any more rapid than that threatened the health of the service. That placed a maximum of 4.3 million scores to be produced per day, one for each up-to-date profile request. With six million total registered freelancers, this put the squeeze on the preliminary goal of every-freelancer, every-four-hours.

2.3. Concurrency under the actor model

A direct re-use of the routines in the model generation software would involve:

1. bulk collection of the job application and worked-hours data,
2. bulk collection of the server log data,
3. bulk collection of as many user profiles as possible before the data from (1) and (2) could be considered stale,
4. featurization and scoring of each freelancer whose profile was collected in step (3).

Steps (1) and (2) combine to a total of about 40 hours of collection and processing time when performed sequentially. Keeping a buffer of 5 minutes aside for Step (4)'s vectorization and scoring (and saving those scores to disc), that means a 4 hour cycle will involve 195 minutes of harvesting user profiles in Step (3). That means an amortized rate of 146,250 scores produced per hour.

The fastest rate possible (one every 20 ms, as limited by the necessary contributions from the user profile service) would mean 180,000 scores per hour. That leaves room for a potential 23% improvement in throughput by moving to a system where user profiles are harvested concurrently alongside the other routines. This potential upside increases as the stringency of the data-freshness guarantee is increased from four hours, to two hours, to one.

2.3.1. THE ACTOR MODEL

The actor model makes it easy to architect a concurrent version of the availability system. An actor framework involves three components:

The Actors A collection of objects, each holding a message queue, some private state, and some private methods. This state can only be updated, and these methods can only be called, by the actor object itself, and in response to receiving a message from some other actor. The response an actor takes to each message is fully completed before it begins its response to the next message.

The Messages Simple, immutable objects that contain information or instructions one actor wants another actor to notice.

The Execution Context The harness which ensures that each dispatched message reaches the right actor queue, the computation each actor calls for is completed in order, and that the overall workload is balanced as evenly as possible over the available hardware resources. This system also provides methods for creating new actors, as well as for setting up a timed schedule to dispatch messages at regular intervals to any actor.

This message-passing version of concurrency avoids the race conditions and other difficulties that can arise from many execution tasks sharing mutable state. Anything mutable is isolated inside individual actors, who only share with each other immutable message objects.

The actor model rose to prominence in an artificial intelligence context thanks to work from Hewitt, Bishop, and Stieger. ([Hewitt et al., 1973](#)) It's true that compared to classical

imperative programming “it imposes a major mind shift in software design.” (Korland, 2011) But it’s decidedly less intense than the large-scale, distributed alternatives described above, and lets novice or over-worked developers avoid the easy pitfalls of writing one’s own stateful multithreading routines.

2.3.2. THE CONCURRENT AVAILABILITY SYSTEM

The family of actors defined for a concurrent availability prediction system uses a hub-and-spoke arrangement, with a single lynchpin actor in charge of featurization from a freelancer’s raw data. Three “historian” actors are in charge of polling the higher-bandwidth Greenplum and S3 sources, and another is in charge of (rapidly) polling the user profile service; all four send their findings back to the central featurizer for storage as state variables. A sixth actor receives (freelancer ID, feature vector) pairs from the featurizer, applies the model, and stages these scores to be bulk uploaded back to Greenplum, where the rest of Upwork’s freelancer search systems can make use of it.

In more detail:

The featurizer The central hub of the actor system: the featurizer keeps track of each freelancer’s worked-hours, job application, and server log information, updating this background data whenever an update is received from a Historian. Whenever a freelancer’s information is received from the User Profile Fetcher – that profile holding the final piece of information needed for a complete feature set – the featurizer sends the freelancer’s feature vector to the scorer-exporter.

The historians Each is responsible for polling a different kind of raw material resource. The execution context is given a timing schedule for each, and periodically sends them “poll for new data now.”

Worked-hours historian Fetches worked-hours records from Greenplum at a schedule of once per hour, forwarding the results found to the featurizer.

Job applications historian Fetches the job application record from Greenplum at a schedule of once per hour, forwarding.

S3 historian Fetches the latest server logs at a schedule of once per hour, forwarding them to the featurizer.

User profile fetcher Every 20 milliseconds, requests and processes user profile data from Upwork’s internal service, passing the results back to the featurizer.

The scorer-exporter Waits to receive from the featurizer a freelancer ID, and a vector of features capturing that freelancer’s recent use of the Upwork platform. It then applies the (previously trained) logistic regression model to the features, generating a “odds-of-accepting-an-interview-invitation” score for that freelancer. The scorer-exporter stores as many of these as it receives in one hour, then uploads the large collection all at once to a table in Greenplum where it can be read by the rest of the Upwork backend systems and incorporated into search rankings.

This family of actors, along with arrows indicating the principal flow of information from raw material, to feature vector, to availability score, is depicted in Figure 2.

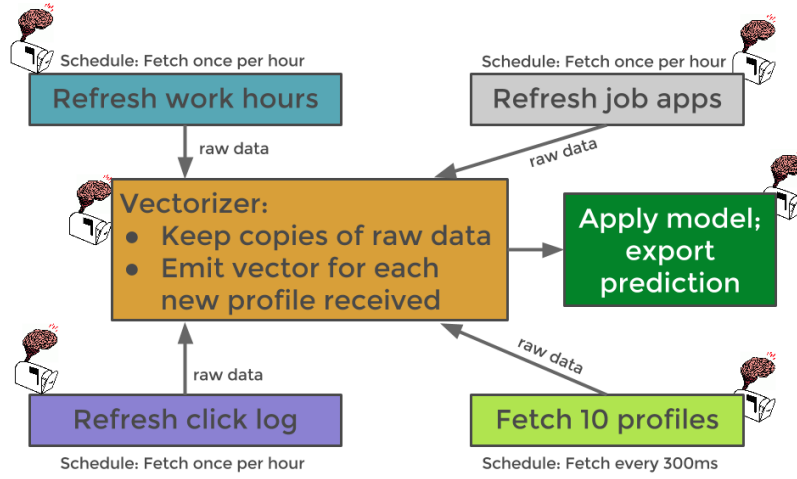


Figure 2: The actor system for freelancer availability scoring. Each actor can be thought of as a mailbox, queuing up messages, with a little brain, holding memories and reacting to those messages one by one.

By dividing responsibility for the full scoring pipeline across several distinct workers, the execution context is able to take their concurrent operations and schedule them for the operating system to perform. When the underlying machine offers multiple CPUs, this can mean the actors operate simultaneously, allowing for the latency-reducing speed ups we were hoping for. With single-file processing of messages we avoid problems where, e.g., one step in the scoring procedure tries to read its stored worked-hours data at the same time another step is updating that same data – trying to square this circle from scratch can easily result in code that deadlocks, livelocks, or just plain drops data.

Figure 3(a) depicts the order of operations from our original, sequential scoring system, where raw material gathering, vectorization, and scoring happen one after another, repeating once an hour. Time spent gathering the Greenplum and S3 resources takes away from the rate-limited use of the user profile service.

Figure 3(b) demonstrates the concurrent availability scorer run on a four core machine. The execution context is able to provide parallel execution of actor routines, meaning the user profile historian never ³ needs to take a break to let the other raw material harvesters use the execution thread. The rate of availability score production can be maxed out by taking full, round-the-clock advantage of the rate-limited resource.

The actually-implemented availability system, now running as a service on Upwork’s backend, achieves this same rate of freelancer score production.

3. This may be a little generous: the Java runtime environment itself may preempt the execution of one or more actor’s operations in order to execute services like garbage collection.

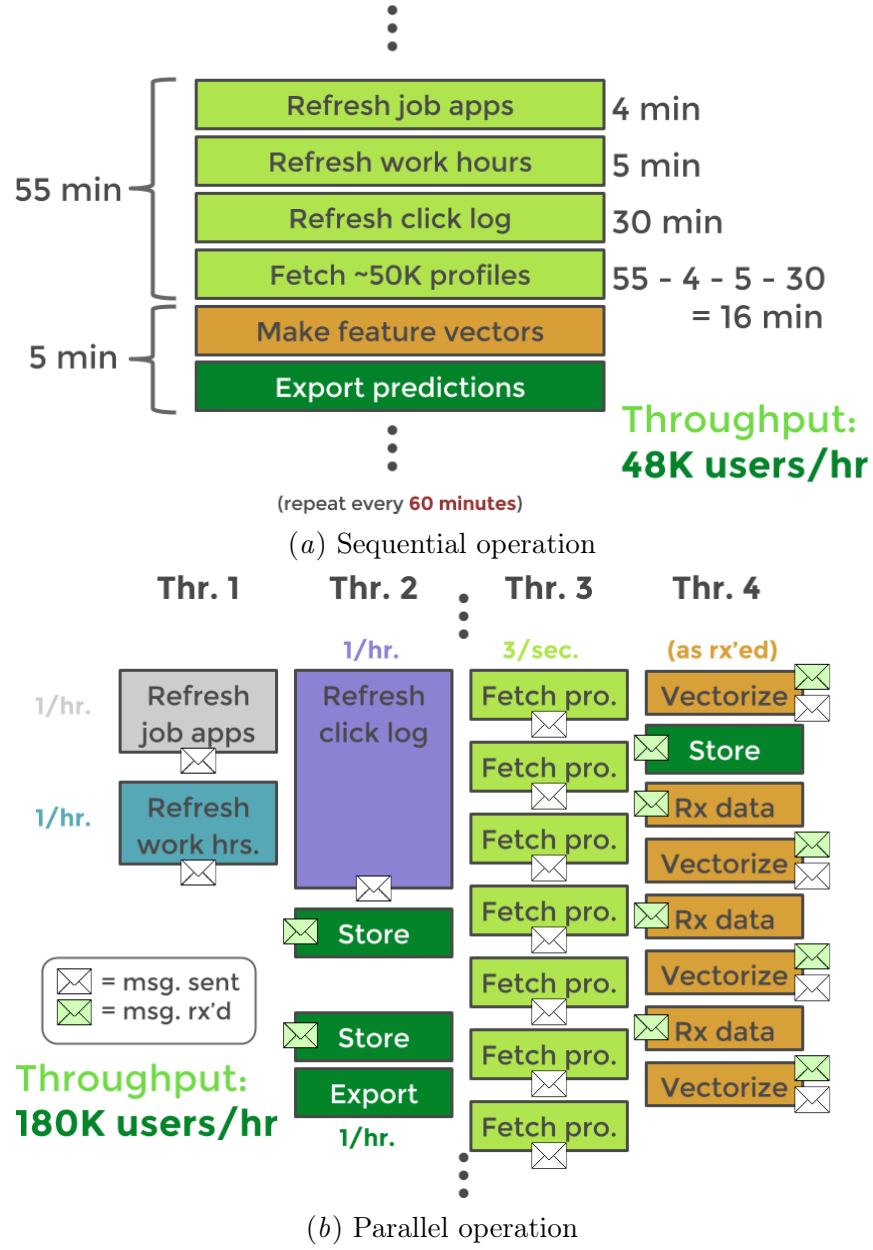


Figure 3: The computation cycle for availability in its sequential and concurrent formulations, on a four-CPU system. When data is kept to within one hour of freshness, the actor model provides an approximately 3.75-fold increase in throughput.

3. Case study: a text classification API

As a hands-on demonstration of powering a machine learning service with the actor framework, a repository of code has been put online as a companion to this paper at <https://github.com/bgawalt/papis-akka-demo>. (Gawalt, 2015) The project is an online learner, a text classifier trained one example at a time to distinguish between positive and negative class documents (e.g., spam versus valid messages).

It is implemented in Scala, making use of the Akka library (Typesafe, Inc., 2015) to handle the execution context and actor classes, and the Spray library (The Spray project, 2015) to connect that execution context with HTTP requests from the outside world.

3.1. Scala, Akka, and Spray

Scala is a high level language, designed to allow for easy productivity when developing in both functional programming and object-oriented styles. The Scala language allows developers to define **traits**, ways to partially define class features. Declared attributes can be implemented right along with the **trait** definition itself, or they can defer the implementation until the trait is finally used to build a legitimate **class**.

They're much like Java's **interfaces**, with the added benefit of mixin capabilities – several **traits** can be defined independently with their own methods and fields, then combined arbitrarily to create new subclasses sharing the union of their attributes.

The Akka library provides a **trait** called **Actor**. To define an actor, declare a new **class** which extends **Actor**, which will require you to implement the method **receive**. This method is of type **PartialFunction[Any, Unit]**, exactly as we'd expect: the actor can receive any kind of message, and then does something. The **match** keyword makes it easy to encode reactions to different expected message types, and provide a default “unexpected command” behavior.

Spray then builds on Akka by offering a trait of **Actor** which can handle HTTP requests and responses in the same asynchronous, message-passing style. It's a convenient way to interact with an actor system.

3.2. Text classification servers

The demo project involves two approaches to the same binary classification task, implemented as two separate API servers: **LoneLearnerServer** and **PackLearnerServer**. These APIs are backed by two types of actor: the request parser and the learner.

The request parser, one per server, inherits from Spray's **HttpServiceActor**, whereby HTTP requests to a particular socket can be forwarded to the parser's **receive** method as a **RequestContext** object. This request context contains useful information, like the URL of the resource requested, and has a **complete** method which should be invoked to send a response to the client via Spray's HTTP mechanisms.

The parser can decode the requested URL and then forward instructions, along with the context to be completed, to the second actor in the system: the learner. Via this API, the learner can perform six tasks: score a new document based on the current model state; update the model state when given a document and a positive or negative class label; report its current state as a string; reset itself to it's original, zero-observations state; synchro-

nize with fellow learners by transmitting information about recently-seen observations, or incorporating that information received from another learner.

The `LoneLearnerServer` has only two actors: one parser, listening to HTTP on a particular socket, and a learner, waiting to receive new documents to judge or labelled documents to learn from. Figure 4 lays out the basic execution flow. There’s a risk that too many requests received in rapid succession could cause steadily longer delays between client request and response.⁴

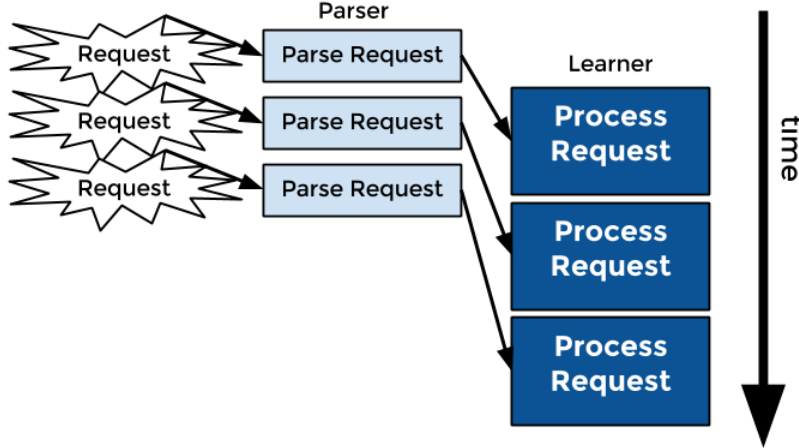


Figure 4: A text classifier system with a lone learner. The learner, with its more computationally intensive tasks, is a potential bottleneck, and latencies can increase as requests pile up.

To break this bottleneck, `PackLearnerServer` has a single parser working with N learner actors. Each HTTP request parsed can be routed to a single learner, selected in a one-after-the-other sequence, or selected pseudorandomly (such as by taking the request’s hash value modulo N), spreading the computational load and increasing the odds that the selected learner fielding the request will be unoccupied and ready to work at the time of the request.

The drawback is that when a new labelled document is observed, it can only update the model housed in the selected learner’s state. Predictions made by the other learners in the pack won’t be able to benefit from the knowledge gained from the update. To patch this, learners are able to send each other update messages after a certain number of observations. This synchronization pulls them offline, unable to respond to new requests until the updates are incorporated.

The machine learning specialist can judge how to trade more frequent updates (and potentially longer delays between request and response) for greater fidelity to the full collection of observed documents. Figure 5 depicts an example of this system’s execution of requests.

4. In practice, it seems fairly rapid even with this obvious bottleneck – on a two-core machine, the system was able to process 20,000 predict-then-update cycles in around 60 seconds, or about 1.5 milliseconds per instruction to the learner.

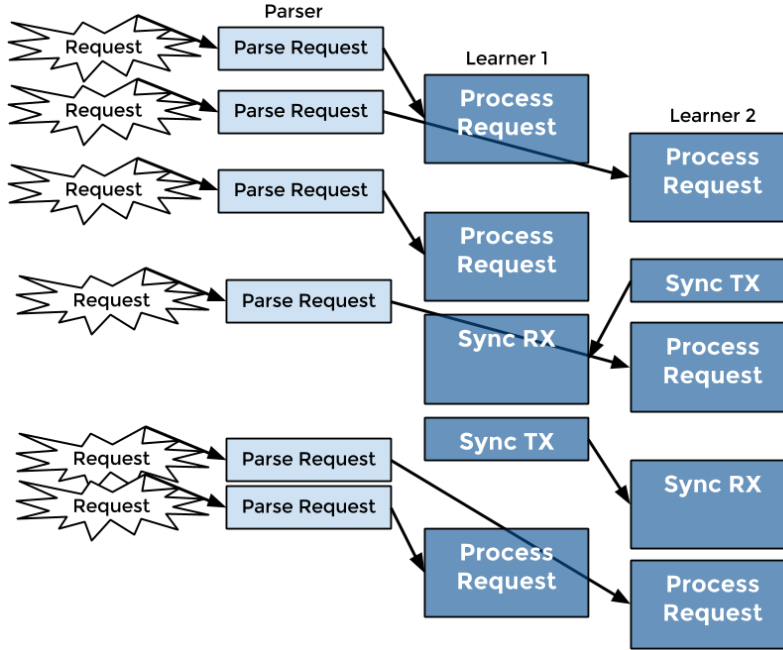


Figure 5: A text classifier system with multiple learners. Request latency can be reduced by sharing the load, though a trade-off has to be made: learners can stay in the dark about recent observations for longer periods (possibly issuing noisier predictions), or more time can be spent synchronizing models between learners (possibly delaying new requests). Note the order in which requests are completed may not match the order in which they were issued.

4. Conclusion

Data science and machine learning are growing as fields and professions, expanding into new domains. That necessarily means engaging firms who find the whole field unfamiliar; and unfamiliarity can inspire risk aversion. These firms in particular are going to greatly appreciate it when a specialist demonstrates a production-ready model at minimal cost, including infrastructure costs, engineering person-hour costs, and maintenance costs.

Exposing concurrency yields efficiencies and reduces costs for any service. Machine learning and data science products benefit especially from the actor framework. It is straightforward to take the standard procedure – gather raw data, produce features from that data, and score those features – and define a collection of actors. That simplicity generates savings in engineering and maintenance efforts.

Upwork was able to take an underperforming sequential software routine, port the particular components to one actor each, and build a system which achieves the maximum possible scoring throughput with much-improved “freshness” guarantees.

The true benefit of the actor framework is its simplicity. The `LoneLearnerServer` text classification API, built on the well-designed Spray and Akka libraries, is under 250 lines of code. And other languages offer their own actor libraries.

For Java, there’s the Quasar platform ([Parallel Universe Software Co., 2015](#)), as well as a Java API for Akka. For C++, there’s CAF, The C++ Actor Framework ([Charousset et al., 2014](#)). For Python, Pykka ([Jodal, 2015](#)) (though to take advantage of parallelism, your program may benefit from Jython ([Python Software Foundation, Corporation for National Research Initiatives, 2015](#)) and it’s lack of global interpreter lock). For Ruby, consider Celluloid ([Arcieri and Keme, 2015](#)) (as with Python, consider JRuby ([Nutter et al., 2015](#)) or Rubinius ([Phoenix and Shirai, 2015](#)) to enable thread-level parallelism).

Having the simple version of a predictive API up and running makes it easy to expose and capitalize on greater amounts of concurrency:

- Which actor routines can swap their I/O library for a non-blocking equivalent (freeing thread resources for another actor to use)?
- Can the workload be smoothed by managing duplicate actors (as with `PackLearnerServer` in our example), or by spawning new actors on-demand?
- More ambitiously, can we start building distributed networks of actors, passing info and sharing tasks?

For smaller scale operations, where the actor framework and its concurrency is overkill. Some machine learning applications will need only produce new predictions infrequently enough – a new batch overnight, say – to allow for the exact same routines to be used “in the wild” as were used for designing, validating, and testing the model “in the lab.” The machine learning specialist might be of more use moving on to generate a new better model in a new domain, than to rig up a highly concurrent solution.

As the load increases on an actor framework increases, greater care and maintenance is required. Without backpressure, overstuffed mailboxes can cause fatal out-of-memory errors. Timed-out operations need to recover gracefully. As these problems mount, the value proposition of large-scale platforms like Spark and Storm become more compelling.

But for that wide and intermediate range of problems between these scenarios, the actor framework helps the machine learning specialist deliver great results. It is capable enough to support medium-scale applications, especially given the affordability and ubiquity of four, eight, sixteen core systems. It is simple enough that the specialist can assemble the concurrent solution on their own: a production deployment from the same person who designed the model.

Empowered, self-sufficient data scientists are going to continue to push the frontiers of their profession. They should consider adding the actor framework to their toolkit.

References

- Tony Arcieri and Donovan Keme. Celluloid project. <https://celluloid.io/>, 2015. [Online; accessed 22-October-2015].
- Dominik Charousset, Raphael Hiesgen, and Thomas C. Schmidt. Caf - the c++ actor framework for scalable and resource-efficient applications. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control, AGERE! '14*, pages 15–28, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2189-1. doi: 10.1145/2687357.2687363. URL <http://doi.acm.org/10.1145/2687357.2687363>.

- Brian Gawalt. PAPIs Akka demo. <https://github.com/bgawalt/papis-akka-demo>, 2015. [Online; accessed 22-October-2015].
- Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.
- Stein Magnus Jodal. Pykka documentation. <https://www.pykka.org/en/latest/>, 2015. [Online; accessed 22-October-2015].
- Guy Korland. *Practical Solutions for Multicore Programming*. PhD thesis, Tel Aviv University, 2011.
- Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’15, pages 239–250, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2758-9. doi: 10.1145/2723372.2742788. URL <http://doi.acm.org/10.1145/2723372.2742788>.
- Charles Oliver Nutter, Thomas Enebo, Ola Bini, and Nick Sieger. JRuby project. <http://jruby.org/>, 2015. [Online; accessed 22-October-2015].
- Parallel Universe Software Co. Quasar documentation. <http://docs.paralleluniverse.co/quasar/>, 2015. [Online; accessed 22-October-2015].
- Evan Phoenix and Brian Shirai. Rubinius project. <http://rubini.us/>, 2015. [Online; accessed 22-October-2015].
- Python Software Foundation, Corporation for National Research Initiatives. Jython: Python for the Java platform. <http://www.jython.org>, 2015. [Online; accessed 22-October-2015].
- The Apache Software Foundation. Apache Spark project. <http://spark.apache.org/>, 2015a. [Online; accessed 22-October-2015].
- The Apache Software Foundation. Apache Storm project. <http://storm.apache.org/>, 2015b. [Online; accessed 22-October-2015].
- The Spray project. Spray documentation. <http://spray.io/documentation/>, 2015. [Online; accessed 22-October-2015].
- Typesafe, Inc. Akka documentation. <http://akka.io/docs/>, 2015. [Online; accessed 22-October-2015].

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 Pre-

diction 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 servers). 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.

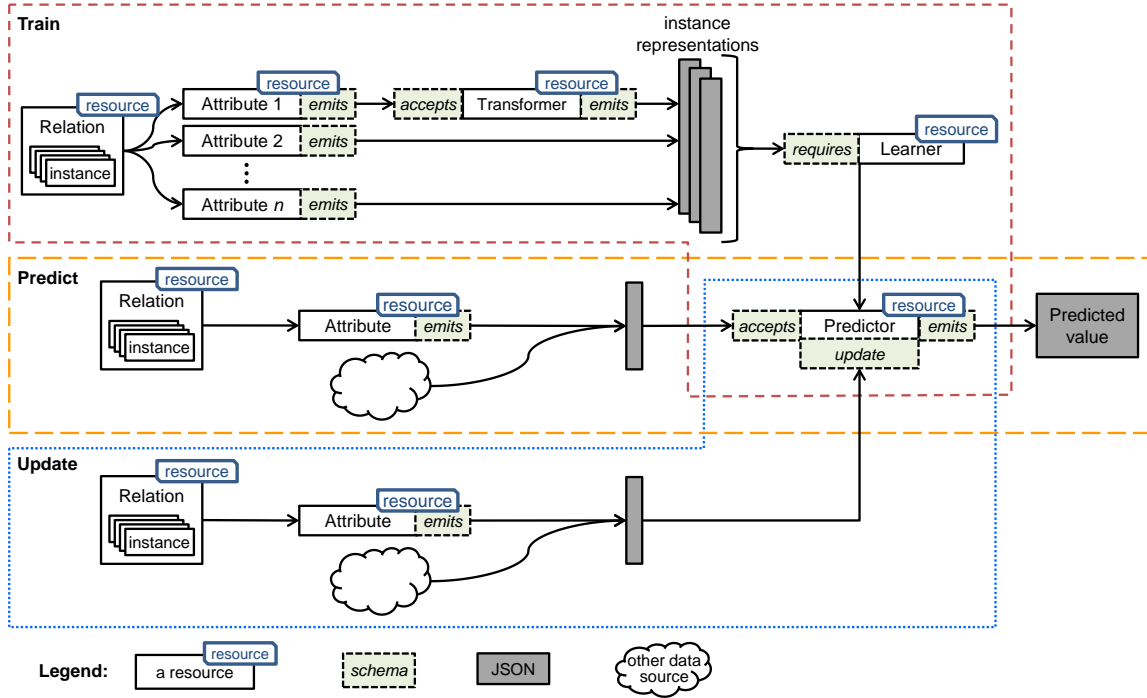


Figure 1: Common data flows through a PSI service. In this illustration the output from Attribute 1 is transformed before it is used in learning

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

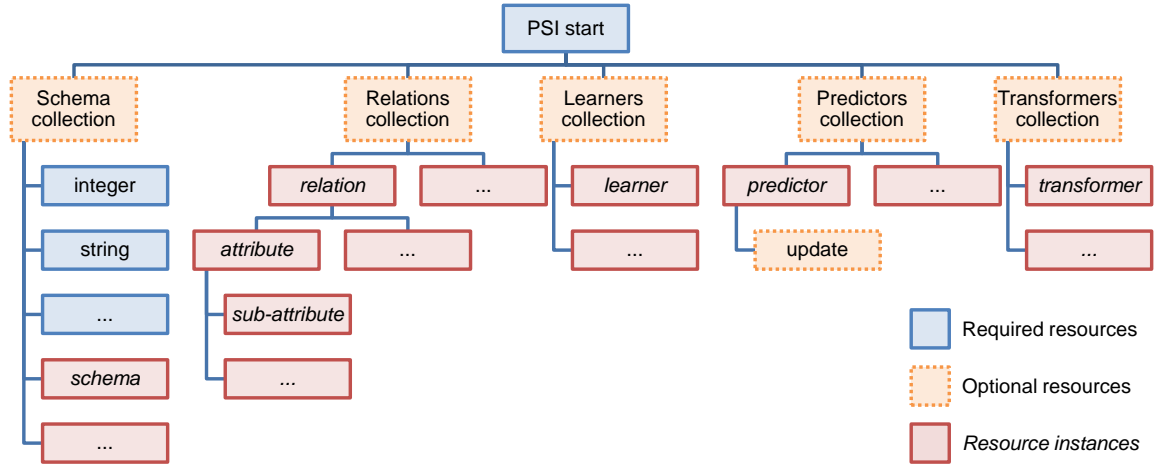


Figure 2: PSI Service Resource Hierarchy. All resource collections are optional and will depend on the nature of the service being offered. A service may also consist of a sole predictor, transformer or relation (and its attributes)

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  $x^2_{a_i} x^2_{b_j}$  for all  $x_i, x_j$  in input and  $0 \leq a+b \leq 2$ ",
  "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-compatible 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 Floor on the diagonal of the covariance matrix to prevent overfitting. (optional)

n_components Number of mixture components. (optional)

covariance_type 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 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 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



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 Jeliaskova, Vedrin Jeliaskov, ..., 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.

The Past, Present, and Future of Machine Learning APIs

Atakan Cetinsoy

Francisco J. Martin

José Antonio Ortega

Poul Petersen

2851 NW 9th, Suite D,

Corvallis, OR 97330

CETINSOY@BIGML.COM

MARTIN@BIGML.COM

JAO@BIGML.COM

PETERSEN@BIGML.COM

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

Abstract

In this paper, we start off by summarizing the key evolutionary turning points of machine learning APIs and conclude by laying out our vision for the future of this key enabling component that can power tomorrow’s ubiquitous intelligent systems.

Keywords: Machine learning, predictive API

Having set out to make machine learning more consumable, programmable and scalable back in 2011, BigML has come to realize the central importance of REST APIs in tackling this big challenge. This has given us reason to believe that as machine learning continues its journey originating from the laboratory into the real-world, APIs will dominate its future footprint more than anything else.

1. The Evolution of Machine Learning APIs

It has been a long time since IBM’s then CEO correctly predicted Professor Arthur Samuel’s public demonstration of his checker playing self-learning program would increase IBM’s stock by 15 points back in the 1950s. The primary actors of machine learning in that early phase were mainly academics and innovators at very large corporations that could afford large R&D departments. Starting in the 1980s machine learning split from the discipline of statistics and became a field of its own. International Machine Learning Society’s first workshop was held in 1980 and it helped spur the formation of a community around the topic.

After the genesis, the areas of focus shifted towards inventing new algorithms and other theoretical research followed by the popular themes of parameter estimation and scalability. Around the turn of the century automation and composability became of interest, while some groundbreaking new algorithms such as automated representation were keeping the ball rolling on the earlier themes (see Figure 1). The attempts to improve existing algorithms by addressing their weaknesses led to the development of more complex techniques such as boosting and ensembles among which bagging and random forest are especially noteworthy.

More recently, the focus has shifted towards applicability and deployability of machine learning systems. Since the ultimate goal is to solve real-world problems, the industry needs to concentrate on what needs to be done to make modern machine learning algorithms ready for real-life challenges. One of the driving factors in this new direction is the data deluge fueled by increasingly powerful cloud storage and computing technologies, that are

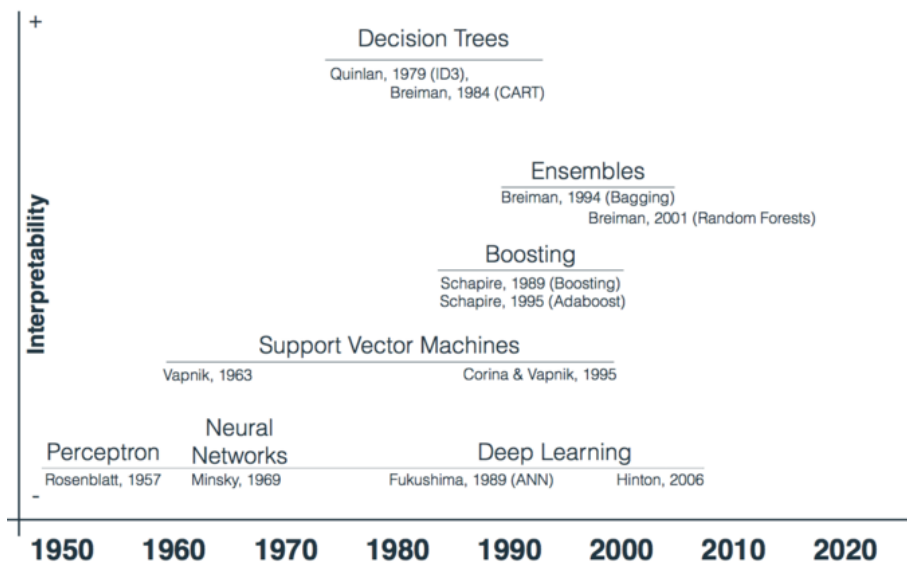


Figure 1: Evolution of Machine Learning Algorithms

able to cost-effectively process magnitudes higher amounts of data of ever greater variety, veracity, and velocity. Business media has promptly caught up on these innovations and the buzz generated in turn created heightened expectations for data-driven decision making powered by smarter enterprise applications. However, this promised land is remains to be fully surveyed for three primary reasons:

1. Real-world machine learning problems call for much more than picking the best algorithm,
2. Scaling machine learning to cope with the data explosion is a challenging process in itself, and
3. To complicate matters even further, existing set of machine learning tools were mainly designed for scientists — not developers.

2. Present Day Challenges

Let us cover these reasons in more depth. If we break down the typical machine learning project into its stages (see Figure 2), it becomes immediately obvious that the learning part is a single step out of many that all together make for a successful launch. High value predictive use cases must be identified with plenty of involvement from business analysts and domain experts. As a first step the chosen use cases must be stated as logical learnable problems. Data from disparate sources must be corralled, cleaned and conditioned before it gets fed to the tools that can efficiently train generalizable models.

As we turn our attention to the learning and the predicting stages of a typical machine learning project, we also need to keep in mind that most incoming data has inherent time-value that decays over time so it is crucial to act on it while it is still relevant. Rapid

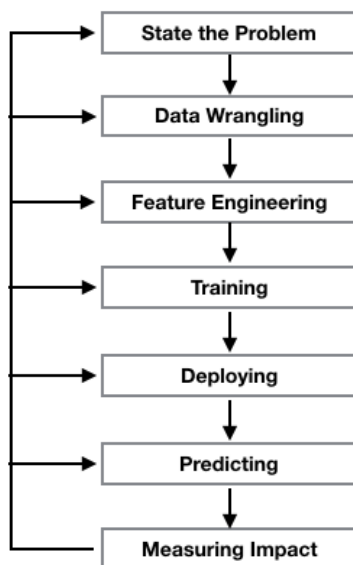


Figure 2: Typical Machine Learning Project Stages

turnaround of single stream of predictions is necessary but not sufficient for machine learning problems that call for a large number of parallel models. Scaling quickly becomes an equally big consideration. A good example is a collection of user specific fraud detection models that have to churn high volumes of accurate predictions with response times in microseconds in order to be deemed useful in a commercial setting. Successfully completing all the stages takes time and often requires multiple iterations before an acceptable predictive performance target is reached. This type of first hand experience leads machine learning experts to not only gain a new appreciation of the team effort required but it also further highlights that machine learning techniques are only as good as the impact it has on the real-world. This maxim shall serve as the guiding light of the machine learning community at large as we move forward. The third present day challenge we mentioned is the assertion that legacy machine learning tools have been built by scientists for scientists so they discriminate against developers that don't have a clue about all the different configuration parameters of a given algorithm. Even if one starts riding the parameter tuning learning curve, he is quickly faced with a 'Paradox of Choice' because of the exploding number of essentially similar algorithms. A recent paper ([Fernandez-Delgado et al., 2009](#)) compared a comprehensive list of 179 classification techniques arising from 17 different families against 121 data sets, which represent the whole UCI database (excluding the large-scale problems). The authors found that the random forest family of classifiers produced the best results overall even though different classifiers did best against different data sets. In a fast-paced business context, it is often more valuable to have a good solution fast instead of the perfect solution at an uncertain future date. Often, this makes it difficult to justify the time spent on optimizing many similar models due to diminishing returns. The fact that the Netflix Prize ([Prize](#)) winning ensemble methods never found their way into the company's real-

life production recommender system is yet another reminder of the need for feasibility in a business context defined by efficiency. Examples of currently utilized machine learning tools include both workstation-oriented pre-Hadoop software (Weka, R, Orange, KNIME, scikit-learn) and post-Hadoop ones (Mahout, Spark MLlib). However, the availability of larger datasets and more sophisticated data pipelines (e.g., AWS Data Pipeline) in the post-Hadoop era has done little to resolve the complexity issue, which still inhibits wholesale adoption. On the other hand, incumbent commercial data mining tools such as SAS and IBM SPSS suffer from the same problems except that they also happen to be much more expensive from a total cost of ownership perspective even as these vendors strive to fully adapt their portfolios to the cloud.

3. Future Scenarios

A promising approach in overcoming these sources of friction is exploring ways to further abstract machine learning by automating algorithm selection and parameter tuning while controlling any negative impact on eventual model performance. If we take a page from the historical arc of web application development, we observe that the evolution of REST APIs and JSON as a data interchange standard were ushered by industry leaders like Ebay, Salesforce and Amazon in the early 2000s. As these technologies reached maturity they increasingly defined the forefront of digital innovation in the commercial realm be it on desktops, mobile devices and for both business-to-consumer and business-to-business applications. Drawing parallels with the future evolution of machine learning tools, it is not too far-fetched to predict related APIs taking a lead role in defining their future trajectory. In fact, in the current decade we have already observed the launch of Google's Prediction API (2010) followed closely by BigML's inception in 2011. One might ask what is Hadoop's likely imprint on these new generation tools? Against the backdrop of massive tech media publicity taking up the wavelengths, it boils down to the provisioning of a more powerful on-ramp that supports more sophisticated data pipelines leading to more capable cloud-based machine learning applications. This means the big promises of distributed storage and distributed processing will remain unfulfilled unless the three speed bumps mentioned earlier are successfully addressed. Rather optimistically, we see this as a matter of time since a steady stream of new investment keeps flowing into Machine Learning as a Service (MLaaS) in the cloud most recently evidenced by the launches of Azure ML (2013) and AWS ML (2014). BigML's approach (see Figure 3) to its API turns each machine learning step to an end-point e.g. creation of a dataset, clustering, anomaly detection, model evaluation, batch predictions etc. Once provided with complete documentation developers feel right at home in accessing the REST API via the corresponding binding for their favorite programming language e.g. Node.js, Python, Java, Objective-C, Swift etc. On the other hand, our proprietary command line language BigMLer packages multiple machine learning tasks into a single command, which helps further abstract machine learning for non-specialists.

With concrete examples like that we can conclude that the confluence of the cloud and machine learning have taken on a new significance as cloud-based machine learning APIs can:

- Keep abstracting the complexity of machine learning algorithms,

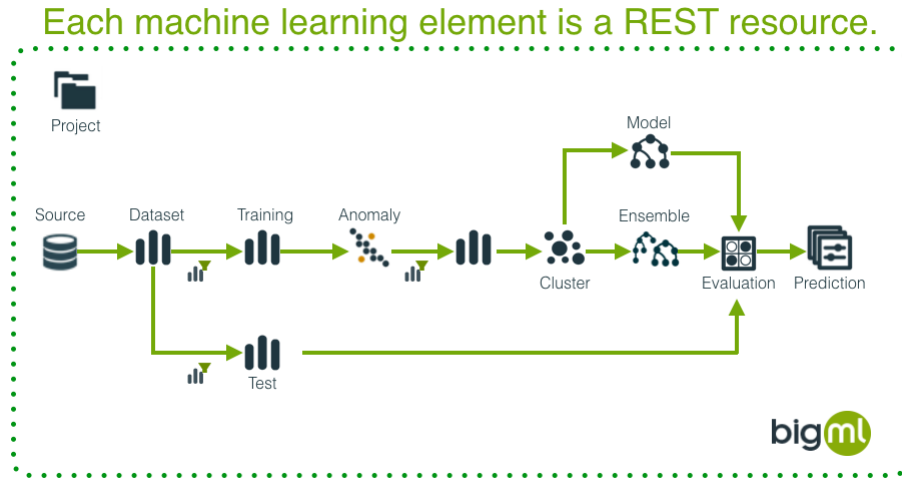


Figure 3: API-first Machine Learning

- Seamlessly manage the heavy infrastructure needed to learn from data and to make predictions at scale i.e. No additional servers to provision or manage,
- Easily close the gap between model training and scoring,
- Empower developers with full workflow automation,
- Add traceability and repeatability to all machine learning tasks for enterprises.

The combination of these enablers are paving the way to the democratization of machine learning. Without a doubt another democratization driver that cannot be overlooked is cost. Luckily, cloud-born machine learning tools with fully featured APIs come at a modest fraction of the total costs for equivalent tools from the previous era since those included many hidden fees and practically mandatory add-on modules to match the same level of capabilities. In the case of BigML, a computational bandwidth-based pricing scheme is on offer to specifically encourage rapid user adoption instead of a typical per user seat enterprise software pricing model.

Finally, freedom from any lock in effects and the preservation of future deployment options must be secured for full democratization to take place. We mean white-box models that can be exported out of the cloud environment they were created in so as to allow for the creators to port them over to a private cloud or any other run time environment of their choice. BigML has followed this flexible design to avoid potential customer backlash due to mounting MLaaS costs pertaining to large scale synchronous applications that require predictions to be served locally e.g. on a smartphone to minimize network latency and/or server-side workload.

We are hoping the arguments made so far have convinced the reader that the future of machine learning requires a mindshift from the obsession on algorithms to a more balanced viewpoint where the API is seen as "the product". Unfortunately, meaningful benchmark studies of current machine learning APIs are still few and far in between. This highlights the

heightened need for independent parties to test the existing set of machine learning APIs for us to accurately mark the industry's departure point circa 2015. Some of the dimensions that we recommend for consideration in such future studies are: number and type of algorithms supported, training speed, prediction speed, performance, ease-of-use, deployability, scalability and throughput, API design, documentation, user interface, SDKs, automation, time to productivity, importability, exportability, transparency, system dependencies, and price.

These design principles also summarize the internal architectural choices that BigML had to make in the last 5 years. Aside from internal architecture considerations a compendium of external and open REST APIs have surfaced (e.g. sentiment analysis, image processing etc.) pointing to a greater degree of specialization among machine learning APIs. This means an unprecedented new level of composability and many more interesting smarter applications made possible simply by mashing up these specialized APIs (see Figure 4). In the not so distant future, the need to start even a custom predictive application from scratch will be fully eradicated. A custom voice and visual assistant application for French hikers (Turner) was demonstrated at Gluecon 2014 as a case in point. This particular application lets hikers answer verbal and visual questions about plants and animals encountered on a hike e.g., "Est-ce toxique?" or "Is this poisonous?".

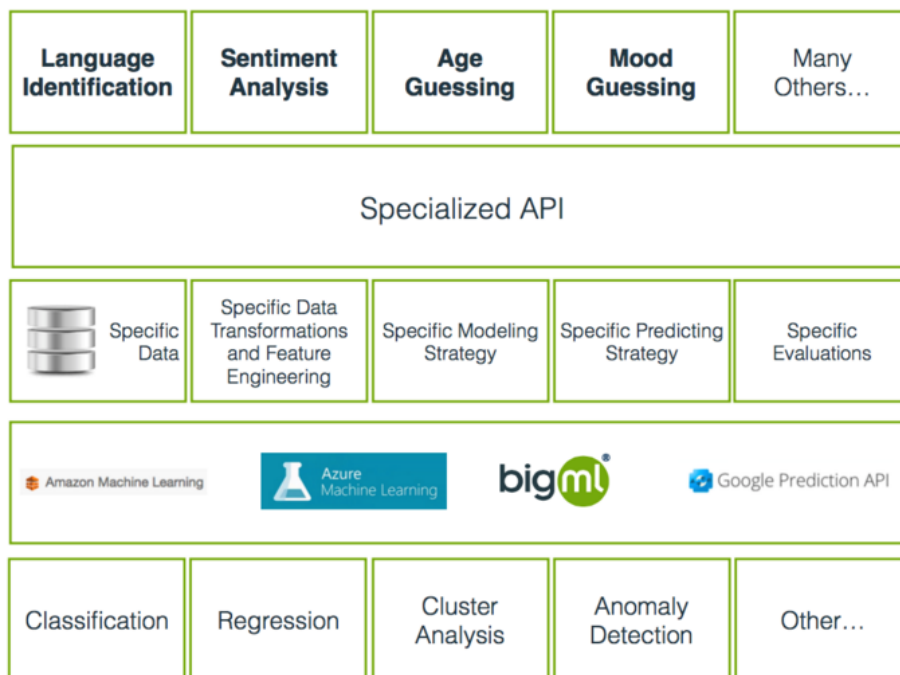


Figure 4: Machine Learning Turned into Specialized APIs

4. Conclusion

What is the end game here? In a sense the evolutionary path ahead may resemble what has happened with Web and mobile applications. They started off with custom scripts,

stove pipe integrations, screen real estate hogging UI controls and half baked million dollar launches. Over time, those disparate components left the center stage for standardized, pliable and robust development frameworks with built in separation of concerns, beautiful and easily customizable UI templates as well as agile and cost effective implementation methodologies. As cloud-based machine learning services mature by adding more and more features, they should stave off the trap of reintroducing unnecessary clutter and complexity that the previous generation of tools have been suffering from. Only then (and to the extent end-users will be given the opportunity to mainly be concerned with their predictive applications instead of the parameters and configurations of the underlying machine learning steps) will we reach full-on commoditization. Besides becoming an integral layer of the new cloud computing stack, machine learning will also transform data architectures such that implicitly predicted values will populate standard data model objects.

The onus is now on the technology providers to build machine learning APIs that are standardized, simple, specialized, composable, scalable yet cost effective in order to realize the goal of completely automated machine learning utilities, which will become an integral part of the modern software application development toolbox.

References

Manuel Fernandez-Delgado, Eva Cernadas, Senen Barro, and Dinani Amorim. Do we need hundreds of classifiers to solve real world classification problems? *Journal of Machine Learning Research*, 2009. URL <http://jmlr.org/papers/volume15/delgado14a/delgado14a.pdf>.

Netflix Prize. URL https://en.wikipedia.org/wiki/Netflix_Prize.

Elliot Turner. Enhancing your cloud applications with artificial intelligence. URL <http://www.slideshare.net/alchemyapi/alchemy-api-enhancing-apps-with-aielliott-turnerjune-2014>. GlueCon 2014.

