Picarus Documentation

Release 0.2.0

Brandyn White

February 21, 2014

CHAPTER 1

About

Picarus is a Computer Vision and Machine Learning web service and library for large-scale visual analysis. Behind the scenes it uses Hadoop/HBase and the front end provides an easy to use REST interface and web app. This is a broad project and is under active development, contact us if you are interested in using it or would like to take part in the development. Visit https://github.com/bwhite/picarus/ for the source.

CHAPTER 2

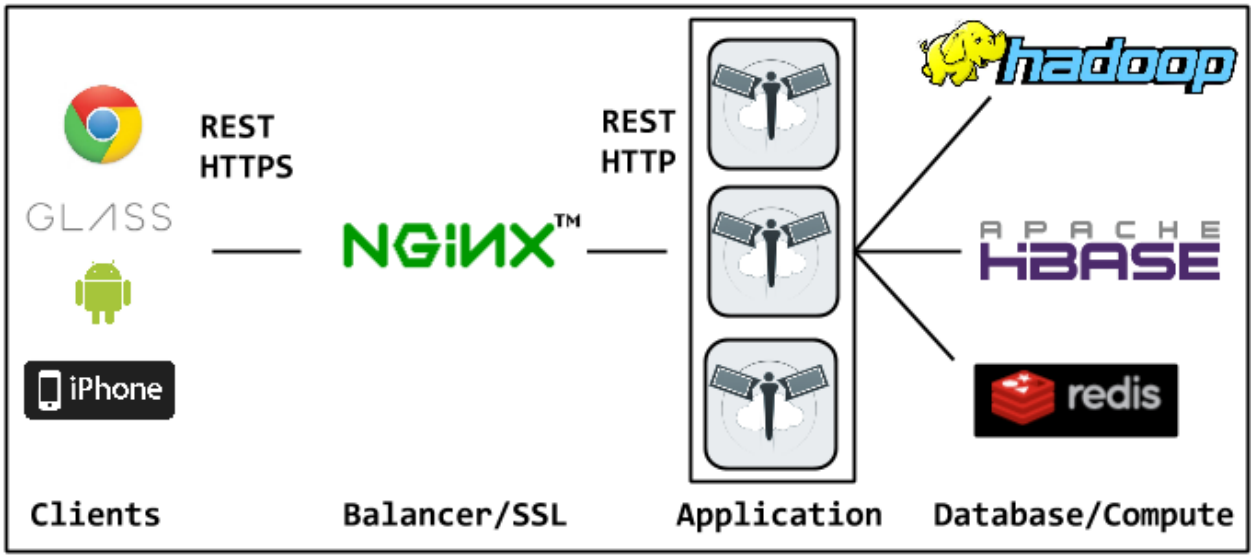
Who

Picarus is developed by Dapper Vision, Inc. (Brandyn White and Andrew Miller). We are PhD students at UMD and are interested in Computer Vision, Web-Scale Machine Learning, HCI, Cryptography, and Social Networks.

CHAPTER 3

Contents

3.1 Overview



3.1.1 Capabilities

Below is an concise (and incomplete) list of the "things you can do", more detail is available throughout the documentation.

- Crawling: Flickr
- Annotation: Standalone or Mechanical Turk. Several operating modes.
- Visualization: Thumbnails, Metadata, Exif, and Location
- Image features
- Classifier Training
- Binary Hash Function Learning
- Search Indexing

3.1.2 How Picarus fits in

Picarus is designed to be used in a variety of capacities from data warehouse, execution engine, web application, REST API, and algorithm factory; however, these are intended to be optional and the level of integration and involvement should be determined by the user.

Data Warehouse

Picarus uses two data models: row access (i.e., table/row) and contiguous row slice access (i.e., table/startRow/stopRow). Each row contains a columns of values. Essentially any datastore that can be modeled in this form can be easily integrated in Picarus. Picarus currently uses HBase as the primary datastore and Redis for metadata. Picarus can be used effectively as a convenient way to interface with data as it provides a convenient server with authentication, user permissions, and sharing.

Execution Engine

Processing occurs either at the web application level or on Hadoop depending on the job submitted. No state is kept on the Picarus application servers, which allows multiple instances to be run on separate machines behind a load balancer (e.g., nginx, haproxy). Hadoop is powerful but often difficult for new developers to work with. Picarus provides a simple interface for Hadoop algorithms.

Algorithm Factory

New classifiers can be trained, search indexes built, and algorithms instantiated. These can all be composed into fairly complex algorithms and executed using Picarus; however, there is no reason why after the algorithm is trained that it needs to be executed solely by the Picarus environment. Consequently, we designed the system so that every algorithm can be 'taken out' as a config file and executed by a standalone binary (see the picarus_takeout project for details). The 'takeout' functionality is implemented in standard C/C++ with as few (and optional) dependencies as possible to enable the broad compatibility. What that means is that you can use Picarus after algorithm creation for execution or you can extract the algorithms and use them as you wish (e.g., mobile apps, compiled to javascript, offline applications).

REST API

The API is provided at a RESTful protocol following standard conventions where possible. Some of the Picarus functionality, such as slice level access, is unique to Picarus and there don't exist common conventions; in these instances we attempted to be consistent and use the 'least surprising' solution.

Web Application

The web interface is implemented using Backbone.js as a single page that provides access to Picarus through the same REST api described in this documentation. The web application is designed to be modular, it is very easy to add a new page for specific funtionality that you may want.

3.1.3 Models

Picarus abstracts data analysis as a sequence of models with a defined flow from source data through them. Each model describes how to instantiate it, what inputs it took (either other models or raw source data), what parameters it has, user notes/tags, and permissions (i.e., who can use/modify it). The result is that given a column name (i.e., data stored for a specific image) we can determine exactly what process of steps occured to create it; moreover, this allows

for natural utilization of pre-processed data, re-use of components, and parallel computation. Models are immutable once created to ensure consistency during analysis.

Examples of models are preprocessors (i.e., take source data, condition is based on specific rules), features (i.e., take processed image and produce a visual feature), classifiers (i.e., take a feature and produce a confidence value (binary) or a ranked list of classes (multi-class)), hashers (i.e., take feature and produce a binary hash code), and search indexes (i.e., take a binary hash code and produce a ranked result list). An output column in the images table corresponds to a row in the models table.

3.2 Algorithms

Picarus supports a variety of algorithm types and instances of them. The computational atoms in Picarus are 'models' which are essentially parameterized functions that are stored in the models table. These parameters are either user defined or are computed algorithmically from training data. Models computed from training data have a 'factory' which performs any necessary training on input data and then ultimately produces a model. A model that does not require training data is fully parameterized by the user. Both factories and models themselves are specified by the 'parameters' table that defines what parameters a factory/model takes in.

3.2.1 Less is More

A natural urge is to pack as many algorithms as possible into a system like this; however, what ends up happening is you have very thin code usage for the majority of the codebase and very heavy usage for a small part. Part of the philosophy of the core Picarus library is to aim for uniform usage across the available methods. It is simple to integrate new algorithms and those on the 'long tail' end will be maintained out of the main tree. This builds confidence and user density in the core algorithms and clearly signals which ones are more experimental. To achieve this it must be easy enough to add new algorithms that this is an acceptable policy. Ideally we'll be able to keep statistics on each algorithms usage and use that to inform our selection decisions.

3.2.2 Image Preprocessors

Takes in a raw image and constrains it (primarily by size and compression type). This ensures that subsequent modules that take in images have a standardized view and can assume that is the case.

3.2.3 Image Feature

Computing image features is the primary task that is 'computer vision' specific, with the majority of Picarus's functionality being applicable to general machine learning tasks. Image features can be categorized into three types: fixed-sized vectors (e.g., histograms, bovw, gist), matrices with one fixed dimension (e.g., feature point descriptors), and three dimensional masks with one fixed dimension (e.g., confidence maps, dense 2d features). Picarus supports each of these feature types and has several instances of each.

Name	Type	Status
GIST	Vector	Ok
Pyramid Hist.	Vector	Ok
HoG+BoVW	Vector	Ok
Pixels	Vector	Ok
HoG	Matrix	Ok
Brisk	Matrix	Ok
Texton Forest	Mask	Integrating

3.2.4 Classifiers

Binary classifiers (i.e., have a concept of negative/positive) take in a feature (often as a vector) and produce a real-valued confidence where towards positive infinity is 'more positive' and towards negative infinity is 'more negative', with the interpretation of positive/negative depending on what the original classifier was trained on. Multi-class classifiers are more varied and harder to generalize over; however, one approach is to have a ranked list of classes with a distance value. Lower distance is 'better', and the list itself is sorted. Some classifiers don't have a clear numerical interpretation of "better' (e.g., nearest neighbor) and the usefulness of the distance value itself will depend on the method used. The ranked list may be complete (i.e., all known classes are present in the ranking) or partial (i.e., only the top performing classes are reported).

Name	Type	Status
Linear SVM	binary	Ok
Kernel SVM	binary	Ok
LocalNBNN	multi	Ok

3.2.5 Hashers

Features (for images and in general) are essentially a raw representation of some trait that we want to capture from the source data; however, they are not, in general, compact or designed for in memory storage. A standard technique (hashing) is to convert features (often represented as large floating point vectors) into compact binary codes that attempt to preserve the majority of their information.

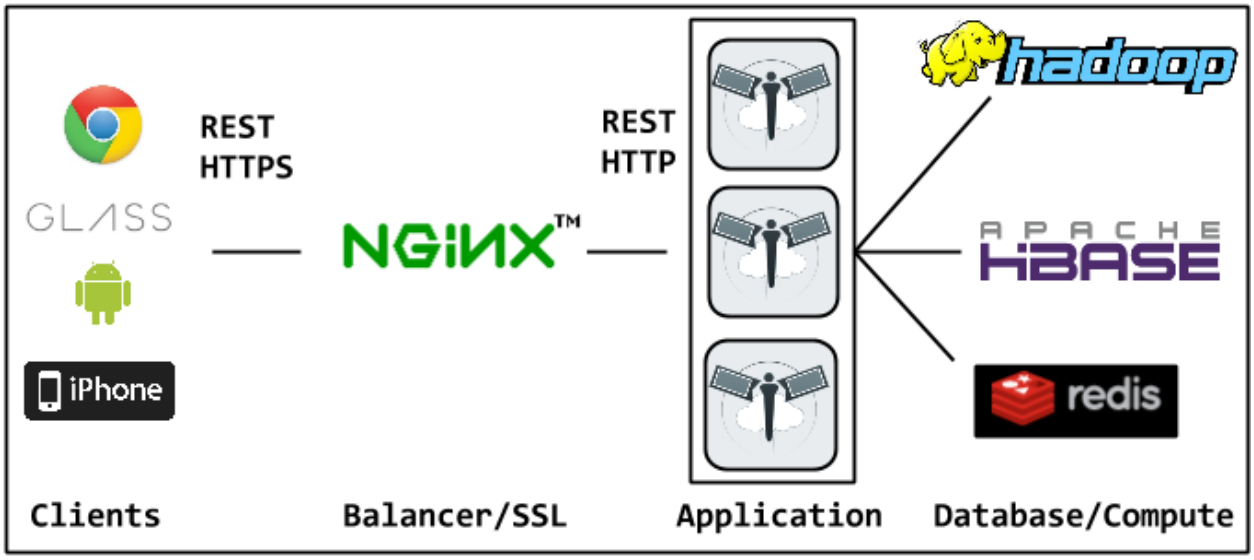
Name	Status
Spherical	Ok
Random Rotation	Integrating

3.2.6 Indexes

Given many features or hashes, we would like to have an efficient data-structure (i.e., an index) for consolidating and retrieving them given a query. The index compares the query to the existing data, producing a ranked list of candidate matches. To compare two features one must define a distance metric (e.g., euclidean (L2), manhattan (L1), cosine, histogram intersection), for binary codes one must define a corresponding operation. This is often a Hamming distance, but may be an operation between the two bit vectors (e.g., spherical hashing used a different metric).

Name	Status
hamming2d	Ok
Spherical	Ok

3.3 Architecture



3.3.1 Overview

The Picarus REST server stores state in Redis, launches jobs on Hadoop, and manages data on HBase/Redis. Because all state is stored in Redis it is safe to have multiple instances running with a load balancer distributing requests between them.

3.3.2 REST Server

The server is written in Python using the gevent socket library and the bottle micro web-framework. Gevent allows the server to process many connections simultaneously.

3.3.3 Load Balancing

Load balancing is optional, Nginx works great for this and also can perform SSL termination.

3.3.4 Hadoop Cluster

Hadoop CDH4 (with mr1) is used on the cluster. The Hadoopy library is used for job management along with Hadoopy HBase which actually launches the jobs using its HBase input format.

3.3.5 HBase Cluster

HBase/Zookeeper CDH4 are used and communication with the REST server is done using the Thrift v1 protocol. Thrift v2 was recently added to HBase but lacks filters while adding check-and-put, for now we are waiting for that interface to stabilize. Each node runs a Thrift server and all communication is done with the local server to spread load and reduce network traffic for cached requests. The Hadoopy HBase library is used to create the Thrift connections and provides helper functions for scanners.

3.4 HBase Tables

3.4.1 Images

Row

Each row corresponds to an "image" along with all associated features, metadata, etc. The image itself is stored in data:image.

Permissions

Users can read all columns and write to data:image and meta: (i.e., anything under meta:).

Column Families

Column Family	Description
data	Image data. data:image is where the "source" image goes. Preprocessors place other copies in data:
thum	Where visualization-only thumbnails exist (these are not to be used for actual analysis)
feat	Image features
pred	Image predictions stored as a binary double.
hash	Hash codes stored as binary bytes. Separated from feat so that it can be scanned fast.
meta	Image labels, tags, etc.

3.4.2 Videos

There are a variety of ways to store videos, this approach allows for client/server/workers to only have a constant amount of the video in memory at any given time which is essential for long videos. As the video isn't re-encoded, if we want to execute a video in parallel with frame-accurate processing then each client will need to have access to portion of video they need to process along with all video preceeding it. This is a very conservative approach that simplifies the implementation considerably and future optimizations will allow us to only require the preceding chunk. However, any non-trivial analysis of the video will dominate the execution time so this isn't a priority at this point.

Row

Each row corresponds to a "video" along with all associated features, metadata, etc. The video data is broken into chunks, with the chunk size stored in meta:video_chunk_size (the last chunk may be partial) and the number of chunks stored in meta:video_chunks. The current recommended chunk size is 1MB. To ensure that when the video is reconstituted it is identical to the original, the sha1 hash is stored in meta:video_sha1. These must all be set by the user from Picarus's point of view, but practically this will be done by a client side library that developers will interact with.

Permissions

Users can read all columns and write to data:video- and meta: (i.e., anything under meta:).

Column Families

Column Family	Description
data	Video data. data:video-* is where the "source" video goes.
thum	Where visualization-only thumbnails exist (these are not to be used for actual analysis)
feat	Image/video features
pred	Image/video predictions stored as a binary double.
hash	Hash codes stored as binary bytes. Separated from feat so that it can be scanned fast.
meta	Image labels, tags, etc.

3.4.3 Models

Row

Each row corresponds to a "model" which is something derived from data, primarily from the images table. Parameters of the model should be included, along with the source columns used to produce it.

Permissions

Users can read all columns and write to data:tags, data:notes, and user: (i.e., anything under user).

Column Families

Column Family	Description
user	Stored user permissions ("r" or "rw") as user:name@domain.com
data	Used for everything not in user:

3.5 Redis Servers

Picarus uses Redis essentially as a global cluster memory as the servers themselves do not hold any state. Each redis server comes with a script that provides admin-level control through the command line (i.e., run the script by itself, see –help on each) for common tasks that need to be done on them. In addition to these, each annotation task needs it's own Redis server to store state in so there are actually many Redis instances for this capacity and they are registered with the Annotator's server below (using annotators.py).

Name	API Exposed	Script	Description
Users	Partial (prefixes/projects)	users.py	User data including prefixes (row permissions) and projects (prefix/model groups)
Yubikey	No	yubikey.py	Yubikey credentials (AES key, user email, etc.)
Annotators	Partial (user's tasks)	annotators.py	Available annotation redis pool (each task needs a Redis) and current tasks

3.6 Redis Tables

The previous section describes the literal Redis servers used by Picarus; however, not all of these are exposed to the user. This section describes what the user can access through the API.

Table	Description	Row
prefixes	Auth'd user's prefixes	Data table (images/video)
projects	Auth'd user's projects	Data table (images/video)
annotations	Auth'd user's annotation tasks	Annotation Task ID
annotations-results-:task	Annotation results for :task (row=annotation)	Annotation ID
annotations-users-:task	Annotators for :task (row=annotator)	User ID

3.7 Basic Tables

Tables that are a small constant size are stored in memory on the REST servers. For consistency they have the same interface as other Picarus tables.

3.7.1 Parameters

Row

Corresponds to a model/factory that can be instantiated using POST /data/models.

Permissions

Users can read all columns but not write to any.

Columns

Column	Description
type	Either model (i.e., directly create model) or factory (i.e., process input data and create model)
name	Name of the model (also corresponds to the picarus_takeout class) or factory
kind	Category of model such as classifier/feature (solely used for categorization in the web app)
input_type	Input type to the model, defines a specific binary encoding
output_type	Output type from the model, defines a specific binary encoding
params	JSON object, keys are parameter names, values are object with type and constraints for that type
input_types	Used by factories to specify what input types they require (e.g., feature and meta for svmlinear)

3.8 API

There are Python and Javascript libraries available for communicating with Picarus and they are kept in sync with the server. As the API is being changed somewhat frequently at this point, it is best to use these libraries for communication. The documentation below explains the encodings, REST calls, and parameters available.

3.8.1 Data access

You can access data by row (/data/:table/:row) or by slice (/slice/:table/:startRow/:stopRow which is [startRow, stopRow)). Slices exploit the contiguous nature of the rows in HBase and allow for batch execution on Hadoop.

3.8.2 Two-Factor Authentication: Yubikey/Email

Picarus supports two forms of additional authentication Yubikey (yubico.com/yubikey) which is a hardware token that can be programmed and input through a Picarus admin tool (api/yubikey.py) and email where a key is sent to a user's email address. Using a Yubikey has the benefit of a more streamlined login process (i.e., one press vs checking email and pasting key) and is preferred if available.

3.8.3 Authentication

All calls use HTTP Basic Authentication with an email as the user and either the Login Key (only for /auth/) or API Key (everything but /auth/) as the password.

- Email: Used to send API/Login keys, used in all calls as the "user".
- Login Key: Used only for /auth/ calls as they are used to get an API key.
- API Key: Used for all other calls.

Get an API Key (email)

Send user an email with an API key.

Resource URL

POST https://api.picar.us/a1/auth/email

Example Response

```
{}
```

Example: Python

```
r = picarus.PicarusClient(server=server, email=email, login_key=login_key).auth_email_api_key()
assert r == {}
```

Example: Javascript

```
p = new PicarusClient(server=server)
p.setAuth(email, loginKey)
p.authEmailAPIKey({success: testPassed, fail: testFailed})
```

Get an API Key (yubikey)

Return an API Key given a Yubikey One-Time Password (OTP).

RESOURCE URL

POST https://api.picar.us/a1/auth/yubikey

PARAMETERS

- otp (string): Yubikey token

EXAMPLE RESPONSE

```
{"apiKey": "w0tnnb7wcUbpZFp8wH57"}
```

Example: Python

```
r = picarus.PicarusClient(server=server, email=email, login_key=login_key).auth_yubikey(otp)
assert 'apiKey' in r
```

Example: Javascript

```
p = new PicarusClient(server=server)
p.setAuth(email, loginKey)
p.authYubikey({success: function (r) {if (_.has(r, 'apiKey')) testPassed() else testFailed()}, fail:
```

3.8.4 Encodings

JSON has become the standard interchange for REST services; however, it does not support binary data without encoding and when using HBase the row/column/value is, in general, binary as the underlying data is a byte string. Moreover, we often using rows/columns in URLs, making standard url escape (due to %00 primarily) and base64 not appropriate as various browsers and intermediate servers will have issues with URLs containing these characters. Values on the other hand are never used in URLs but they still must be JSON safe. Base64 encoding is often performed natively and as values are often large (much larger than rows/columns) it makes sense to ensure that encoding/decoding them is as efficient as possible. Consequently, rows/columns are always "urlsafe" base64 (+ -> - and / -> _) and values are always base64. Below are implementations of the necessary enc/dec functions for all the encodings necessary in Picarus. The encodings will be referred to by their abbreviated name (e.g., ub64) and from context it will be clear if enc/dec is intended.

Python

```
import base64
import json
b64_enc = base64.b64encode
b64_dec = base64.b64decode
ub64_enc = base64.urlsafe_b64encode
ub64_dec = base64.urlsafe_b64decode
json_ub64_b64_enc = lambda x: json.dumps({ub64_enc(k): b64_enc(v)
                                          for k, v in x.items()})
json_ub64_b64_dec = lambda x: {ub64_dec(k): b64_dec(v)
                               for k, v in json.loads(x).items()}
```

Javascript

```
// Requires underscore.js (http://underscorejs.org/) and base64
// (http://stringencoders.googlecode.com/svn-history/r210/trunk/javascript/base64.js)
// b64
b64_enc = base64.encode
b64_dec = base64.decode
// ub64
function ub64_enc(x) {
    return base64.encode(x).replace(/\+/g , '-').replace(/\//g , '_');
}
function ub64_dec(x) {
    return base64.decode(x.replace(/\-/g , '+').replace(/\_/g , '/'));
}
// json_ub64_b64
function json_ub64_b64_enc(x) {
    return JSON.stringify(_.object(_.map(_.pairs(x), function (i) {
        return [ub64_enc(i[0]), b64_enc(i[1])];
    })));
}
function json_ub64_b64_dec(x) {
    return _.object(_.map(_.pairs(JSON.parse(x)), function (i) {
        return [ub64_dec(i[0]), b64_dec(i[1])];
    }));
}
```

3.8.5 Versioning

All API calls are prefixed with a version (currently /a1/) that is an opaque string.

3.8.6 HTTP Status Codes

Standard status codes used are 400, 401, 403, 404, and 500. In general 4xx is a user error and 5xx is a server error.

3.8.7 Column Semantics

In several API calls a "columns" parameter is available, each column is b64 encoded and separated by commas (,). The parameter itself is optional (i.e., if not specified, all columns are returned). For GET operations, a row will be returned if it contains a single of the specified columns or any columns at all if there are none specified. As these columns are used in HBase, the column family may also be specified and has the same semantics as they do with the Thrift API (i.e., has the effect of returning all columns in the column family); however, this property only holds for tables stored in HBase.

3.8.8 Content-Type: application/json

If the request "Content-Type" is set to "application/json" then JSON parameters may be provided as a JSON object where columns are replaced with lists of b64 encoded values instead of comma delimiting them in a string.

3.8.9 Table Permissions

The table below contains the data commands for Picarus. GET/PATCH/DELETE are idempotent (multiple applications have the same impact as one). Each table defines which columns can be modified directly by a user (see individual table docs for details).

Verb	Path	Table				Encoding	
	images		models	parameters	annotation-*	Input	Output
GET	/data/:table	N	Y	Y	Y	col	row list
POST	/data/:table	Y	Y	N	N	b64/b64	{row: b64}
GET	/data/:table/:row	Y	Y	N	N	col	b64/b64
POST	/data/:table/:row	Y	N	N	N	raw/b64	b64/b64
PATCH	/data/:table/:row	Y	Y	N	N	b64/b64	{}
DELETE	/data/:table/:row	Y	Y	N	N	none	{}
DELETE	/data/:table/:row/:column	Y	Y	N	N	none	{}
GET	/slice/:table/:startRow/:stopRow	Y	N	N	N	col+raw/raw	row list
POST	/slice/:table/:startRow/:stopRow	Y	N	N	N	raw/b64	b64/b64
PATCH	/slice/:table/:startRow/:stopRow	Y	N	N	N	b64/b64	{}
DELETE	/slice/:table/:startRow/:stopRow	N	N	N	N	none	{}

- "col": a key of "columns" with a value that is b64'd columns separated by commas.
- "b64/b64": key/value pairs that are both base64 encoded.
- "raw/b64": keys that are plaintext and values that are base64 encoded.
- "row list": outputs a json list of objects, each with an attribute of "row" that is the base64 encoded row key. All other key/values are base64 encoded.
- In the url "table" is plaintext. "row", "column", "startRow", and "stopRow" are ub64.

3.8.10 Row Operations

Create a row

Upload data without specifying a row.

RESOURCE URL

POST https://api.picar.us/a1/data/:table

PARAMETERS

- *b64 column* (b64): One or more base64 encoded column/value pairs. See table permissions for what values you can set.

EXAMPLE RESPONSE

```
{"row": b64 row}
```

Create/Modify a row

Upload data specifying a row. A row need not be created with POST before this operation can be called. Use this operation when you want the row to be a specific value (normally the case) and the POST method for temporary data.

RESOURCE URL

PATCH https://api.picar.us/a1/data/:table/:row

PARAMETERS

- *b64 column* (b64): One or more base64 encoded column/value pairs. See table permissions for what values you can set.

EXAMPLE RESPONSE

```
{}
```

Get row

Get data from the specified row

RESOURCE URL

GET https://api.picar.us/a1/data/:table/:row

PARAMETERS

- columns (string): Optional list of columns (b64 encoded separated by ‘,’).

EXAMPLE RESPONSE

```
{"meta:class": "horse"}
```

DELETE /data/:table/:row

Delete a specified row

RESOURCE URL

DELETE https://api.picar.us/a1/data/:table/:row

PARAMETERS

None

EXAMPLE RESPONSE

```
{}
```

Example: Python

```
c = picarus.PicarusClient(server=server, email=email, api_key=api_key)
# POST /data/images
r = c.post_table('images', {'meta:class': 'horse', 'data:image': 'not image'})
assert 'row' in r
row = r['row']
# GET /data/images/:row
r = c.get_row('images', row, ['meta:class'])
assert r == {'meta:class': 'horse'}
r = c.get_row('images', row, ['meta:'])
assert r == {'meta:class': 'horse'}
r = c.get_row('images', row, ['data:image'])
assert r == {'data:image': 'not image'}
r = c.get_row('images', row)
assert r == {'meta:class': 'horse', 'data:image': 'not image'}
# PATCH /data/images/:row
r = c.patch_row('images', row, {'meta:class': 'cat', 'data:image': 'image not'})
assert r == {}
# GET /data/images/:row
r = c.get_row('images', row)
assert r == {'meta:class': 'cat', 'data:image': 'image not'}
# DELETE /data/images/:row
r = c.delete_row('images', row)
assert r == {}
```

Creating a Model

Create a model that doesn't require training data.

RESOURCE URL

POST https://api.picar.us/a1/data/models

PARAMETERS

- path (string): Model path (valid values found by GET /data/parameters)
- model-* (string): Model parameter
- module-* (string): Module parameter
- key-* (ub64): Input parameter key

EXAMPLE RESPONSE

```
{"row": b64 row}
```

3.8.11 POST /data/:table/:row

Perform an action on a row

Each action specifies it's own return value and semantics.

PARAMETERS

- action: Execute this on the row

action	parameters	description
i/classify	imageColumn, model	Classify an image using model
i/search	imageColumn, model	Query search index using image

3.8.12 POST /data/:table/:startRow/:stopRow

Get a slice of rows

PARAMETERS

- maxRows: Maximum number of rows (int, max value of 100)
- filter: Valid HBase thrift filter
- excludeStart: If 1 then skip the startRow, **|maxRows|** are still returned if we don't reach stopRow.
- cacheKey: A user provided key (opaque string) that if used on a repeated call with excludeStart=1 and the new startRow (last row of the result), the internal scanner may be reused. This is a significant optimization when enumerating long slices.
- column: This is optional and repeated, represents columns that should be returned (if not specified then all columns are).

Perform an action on a slice

Each action specifies it's own return value and semantics.

PARAMETERS

- action: Execute this on the row

action	parameters	description
io/thumbnail		
io/exif		
io/preprocess	model	
io/classify	model	
io/feature	model	
io/hash	model	
i/dedupe/identical	column	
o/crawl/flickr	className, query, apiKey, apiSecret, hasGeo, minUploadDate, maxUploadDate, page	
io/annotate/image/query	imageColumn, query	
io/annotate/image/entity	imageColumn, entityColum	
io/annotate/image/query_batch	imageColumn, query	
i/train/classifier/svmlinear	key-meta, model-class_positive, key-feature	
i/train/classifier/nbnnlocal	key-meta, key-multi_feature	
i/train/hasher/rrmedian	module-hash_bits, key-feature	
i/train/index/linear	*TODO*	

3.9 Web Application

Picarus has a full REST api from which client applications can be built. Within the project, there is a web app that tracks the REST api and attempts to expose all of Picarus's functionality. It is intended to simplify new functionality by standardizing commonly used widgets, models, authentication, etc. The web application consists of an authorization modal that is displayed when the page is loaded, a navigation bar with project selection (projects filter models/data shown), a variety of tabs that are accessed using the navigation bar, and a series of reusable models/widgets for the tabs. The application is a single-page that is dynamically updated using javscript (specifically using backbone.js) to allow the user to easily move between tabs without communicating with the server unnecessarily.

3.9.1 Libraries

The web app is build using backbone.js, underscore.js, jQuery, and Bootstrap. All Picarus REST API calls are made with the library in /api/js/picarus.js.

3.9.2 Code Layout

Each tab has a .js and .html file in the api/tabs directory named after the major (name of the tab category) and minor (name of the individual tab in the category) names in the form major_minor.{js,html}. This separation allows for easily adding/removing tabs without significantly impacting other portions of code. Inside each .js file is a function with a name of the form "render_major_minor", this is called every time a tab is to be displayed.

```
// Major: data Minor: flickr
function render_data_flickr() {
}
```

Inside each .html file there should be a script tag with an id of the form "tpl_major_minor", this is put into the DOM, replacing what is inside div with id "container".

```
<!-- Major: data Minor: flickr -->
<script type="text/html" id="tpl_data_flickr">
</script>
```

3.9.3 App Structure

The core of the application itself is in /api/app_template.html and /api/js/app.js. A javascript library for picarus is in /api/js/picarus.js and that is what is used for all Picarus REST API communications.

3.9.4 Building

Since the application has many html/css/js files to simplify it's organization, we have to minify them before serving if we want to achieve consistent sub-second load times. The api/build_site.py script is used to compile the site and place it in the /api/static directory for serving. When run, it calls the Google Closure library to minify the javascript and contatenates css/html. To skip running closure (primarily when debugging javscript code), run with –debug to do a simple concatenation instead. This build_site.py script needs to be run anytime the html/js/css is changed for it to be able to be displayed by the rest_server.py

3.9.5 Globals

To simplify the logic across the app, certain global objects are available. If a backbone collection exists you should use it instead of modifying those tables yourself manually as it will make parts of the app out of sync. If you create something (lets say a new model) you have to update the models table with the new info. For small tables (anything except models) it's ok to resync the whole collection.

Name	Description	Mutating Tabs
EMAIL_AUTH	User's login as {auth, email}	
PICARUS	Picarus API instance	
PROJECTS	Backbone collection of projects	data/projects
PREFIXES	Backbone collection of prefixes	data/prefixes
ANNOTATIONS	Backbone collection of annotations	annotate/*
PARAMETERS	Backbone collection of parameters	
MODELS	Backbone collection of models (only meta:, nothing from data:)	models/{list,create}

3.10 Takeout

An issue with any complex system/library is that as more features are added, it steepens the learning curve for new users. The cost of understanding the relevant parts of the system and installing it can become prohibitive. Moreover, while web applications are a plausible way to bridge language/platform gaps (e.g., perform complex server side processing instead of custom android/iphone implementations) it is an advantage to have the ability to run a system on a variety of platforms. This was considered during Picarus's development and we decided to write all of the execution-time algorithms (e.g., classifier prediction) in C/C++ so that the models (e.g., classifiers, search indexes, features) used by the system would be as cross-platform as possible. The reason for specifiying execution-time algorithms is that they are often simple (e.g., for linear classifiers it is a dot product) compared to the training portion (e.g., numerical optimization). What that means is that the training phase may be tied into the server or depend on Python, but post-training, everything is done in pure C/C++ with BLAS.

To simplify installation of just the execution-time algorithms, they are in the Picarus Takeout repo. All modules in Picarus can be exported using the Takeout link in the web app and then executed on a system with Picarus Takeout installed. The goal is to make this library have as few required dependencies as possible (e.g., use ifdef's to ignore modules that need dependencies we don't have); however, this is an ongoing process and the most significant dependency is on OpenCV, primarily for IO, color conversion, and resizing which could be replicated in the takeout tree without depending on the entire library.

https://github.com/bwhite/picarus_takeout

3.10.1 Linux (ubuntu)

- Build OpenCV (latest version) from source
- Get msgpack (apt-get install libmsgpack-dev)
- Get blas (apt-get install libatlas3gf-base)
- Get fftw3 (apt-get install libfftw3-dev)
- Get cmake (apt-get install cmake)
- cmake picarus_takeout
- make

3.10.2 Windows

Below is a guide to installing Picarus Takeout on Windows.

CMake

http://www.cmake.org/cmake/resources/software.html

Visual Studio

Visual Studio (tested on C++ 2010 Express). https://www.microsoft.com/visualstudio/eng/downloads

OpenCV

MAKE SURE TO USE RELEASE MODE IN VISUAL STUDIO! OpenCV won't work without it. http://sourceforge.net/projects/opencvlibrary/files/opencv-win/ http://docs.opencv.org/doc/tutorials/introduction/windows_install/windows_install.html#windowssetpathandenviromentvariable

PThreads

ftp://sourceware.org/pub/pthreads-win32/prebuilt-dll-2-9-1-release

Msgpack

http://www.7-zip.org/ (for opening up the .tar.gz) http://sourceforge.net/projects/msgpack/files/msgpack/cpp/

Convert the project, build it.

A few problems need to be fixed https://github.com/qehgt/myrpc/issues/3

- Move type.hpp from includemsgpacktype into includemsgpack
- Replace this file https://raw.github.com/msgpack/msgpack-c/master/sysdep.h

FFTW

http://www.fftw.org/install/windows.html http://www.fftw.org/fftw-3.3.3.tar.gz

Include/Library Paths

An example of all of the paths, libraries, and command line arguments can be found here https://gist.github.com/bwhite/5493885

Building

- Use cmake to create a directory (e.g., /build) by pointed its source input at the inner picarus_takeout folder (has the CMakeLists.txt file) and clicking "generate".
- Go into build, open picarus.vcxproj (note, this example is for the picarus project, but the same applies to any other projects generated by cmake)
- Change to release mode. In the solution explorer, right click on picarus, go to properties.
- In Configuration Properties/C++/General/Additional Include Directories add all of the include dirs. (see gist above for example of all these paths)
- For opencv make sure to include every single directory in modules/*/include (lots of copy and pasting)
- Under linker/general, add the library paths under "Additional Library Directories".
- Close that window, right click on picarus, go to build.
- You will need a 1.) a picarus model (if you need one ask me) and an input is an image
- Copy the model and input image into the /build/Release directory.
- Either 1.) copy all the dll's into the /build/Release directory or 2.) make sure windows knows where to find them.
- Go into the /build/Release directory in a terminal, and type "picarus <model> <input> <output>"
- Model is the picarus model, input is an image, and output is where to store the output file
- If everything worked, you'll have a new output file at the path you specified, it is in msgpack format.

3.11 Annotation

Every row in the Redis annotation db's is prefixed with the annotation task as a namespace (e.g., <task>:<row>). Below is a table of the Redis db's used for annotation, all run on a single Redis server.

Redis DB	Description
users	
response	
state	
key_to_path	Maps from a random key to a b64(row) + ' ' + b64(column)
path_to_key	Maps from b64(row) + ' ' + b64(column) to a random key

The state is the global memory of the task as it is being executed. Below is a table of the state rows.

State Row	Description
<task>:rows	PQ of rows, user view reduces priority. Determines which row to show next.
<task>:seen:<user>	Set of rows user has seen.
<task>:data_lock	Lock needed to mutate state/key_to_path/path_to_key

3.12 Examples

3.12.1 Workflow: Crawl/Annotate/Train/Evaluate

3.13 Installation

3.13.1 Requirements

Our projects

- hadoopy (doc): Cython based Hadoop library for Python. Efficient, simple, and powerful.
- picarus_takeout: C/C++ module that contains the core picarus algorithms, separate so that it can be built as a standalone executable.

Third party

- Scipy
- OpenCV
- Hadoop (CDH recommended)

3.13.2 Useful Tools

These are all optional, but you may find them useful. Our projects (ordered by relevance)

- hadoopy_flow: Hadoopy monkey patch library to perform automatic job-level parallelism.
- vision_data: Library of computer vision dataset interfaces with standardized output formats.
- image_server: Server that displays all images in the current directory as a website (very convenient on headless servers).
- static_server: Server that allows static file access to the current directory.
- pycassa_server: Pycassa viewer.
- vision_results: Library HTML and Javascript tools to display computer vision results.
- hadoop_log: Tool to scrape Hadoop jobtracker logs and provide stderr output (simplifies debugging).
- pyram: Tiny parameter optimization library (useful when tuning up algorithms).
- mturk_vision: Mechanical turk scripts.

3.14 Administration

3.14.1 User Management

All user admin tasks are performed using the "users.py" script in the picarus/api directory. Use python users.py –help for details on how to use it. Some of the basic functionality is described below.

To add a user to the database

```
python users.py add you@host.com
```

If you have AWS email enabled (by filling out your api details in email_auth.example.js and moving it to email_auth.js) then they will be sent an email. If not then the information will be printed to the console and must be provided to them manually.

To give a user read/write access to a row prefix

```
python users.py add_image_prefix you@host.com "yourprefix:" rw
```

3.15 Development

3.15.1 Web App

Each tab has a corresponding .js and .html file in api/tabs. The name is "group"_"name" and this is used to generate the nested tab structure. New tabs must be added to build_site.py so that the order is consistent. The build_site.py script concatenates the javascript, html, and css. The javascript is minified using the closure compiler unless –debug is used as a command line argument (this is useful when debugging javascript errors as the minified source is highly transformed). The composite site is in api/static and none of the files are modified at runtime with templating, so they may be cached or placed in a CDN. The server loads these files from disk at runtime and maintains them in memory to avoid disk reads (note that if you change the underlying files you have to stop/start the server process).

3.15.2 REST Server

3.16 Testing

3.16.1 Takeout

Picarus takeout is the core runtime algorithm library used throughout Picarus and ensuring it is operating as it should and is consistent is essential. If there were to be a bug in a takeout algorithm, the bad output would be stored in the database and unless there is some way to know that it would persist and propagate to any other algorithm that uses it. The primary goal is to identify bugs, but even determining when the output is different than it used to be is important so that we can identify if the change is correct and if user's with old (different/wrong) results should be notified. To do this both the python runtime and the standalone picarus executable (both in picarus_takeout) are run through a wide variety of model and input pairs. Moreover, the standalone executable is run using valgrind which is set to fail if any memory bug is detected.

This test suite ensures the following

- Algorithm outputs don't change (equality at a binary level)
- Memory errors are identified immediately (valgrind)
- Python runtime and standalone executable agree

This test suite doesn't check/do the following (we should work towards these)

- Performance regressions (could timestamp runs, notify by delta)
- Behavior of various parameters outside those selected in the models (which are likely conservative)
- May have False Positives on other machines due to differing image libraries (can have alternate set all ppm)

3.16.2 REST Server

Using the Python library, there is a test suite in picarus/tests that exercises post of the functionality.

3.16.3 Web App

Using casper.js (which uses phantom.js, a headless webkit) we script several user interactions with the server. These can be replayed to ensure that they are repeatable and to identify any functional bugs.

3.16.4 TODOs

See https://github.com/bwhite/picarus/issues?labels=Testing&page=1&sort=created&state=open

3.17 History

Concept and Draft (2008)

- "Computer vision web service"

First usable implementation (2011)

- "The OpenCV for Big Data and Hadoop"
- "Mahout for visual data"

Focus expanded (2012)

- "Full-lifecycle CV web application"
- Crawl, annotate, execute, train, analyze, visualize