Palladium Documentation

Release 1.1.0.1

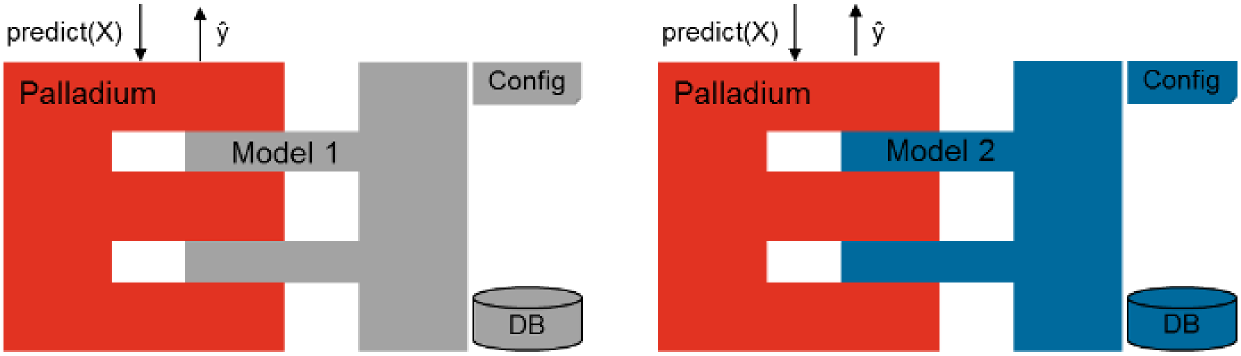
Otto Group BI

April 02, 2017

Contents

Palladium provides means to easily set up predictive analytics services as web services. It is a **pluggable framework** for developing real-world **machine learning solutions**. It provides generic implementations for things commonly needed in machine learning, such as dataset loading, model training with parameter search, a web service, and persistence capabilities, allowing you to concentrate on the core task of developing an accurate machine learning model. Having a well-tested core framework that is used for a number of different services can lead to a reduction of costs during development and maintenance due to harmonization of different services being based on the same code base and identical processes. Palladium has a web service overhead of a few milliseconds only, making it possible to **set up services with low response times**.

A **configuration file** lets you conveniently tie together existing components with components that you developed. As an example, if what you want to do is to develop a model where you load a dataset from a CSV file or an SQL database, and train an SVM classifier to predict one of the rows in the data given the others, and then find out about your model's accuracy, then that's what Palladium allows you to do **without writing a single line of code**. However, it is also possible to independently integrate own solutions.



Much of Palladium's functionality is based on the **scikit-learn** library. Thus, a lot of times you will find yourself looking at the documentation for scikit-learn when developing with Palladium. Although being implemented in Python, Palladium provides support for other languages and is shipped with examples how to **integrate and expose R and Julia models**.

For an efficient deployment of services based on Palladium, a script to **create Docker images automatically** is provided. In order to manage and monitor a number of Palladium service instances in a cluster, **Mesosphere's Mesos framework Marathon can be used for deployment**, also enabling **scalability by having a variable number of service nodes behind a load balancer**. Examples how to create Palladium Docker images and how to use them with Mesos / Marathon are part of the documentation. Other important aspects – especially relevant in enterprise contexts for setting up productive services – like **authentication, logging, or monitoring, can be easily integrated via pluggable decorator lists** in the configuration file of a service, keeping track of service calls and corresponding permissions.

CHAPTER 1

Links

- Source code repository at GitHub: https://github.com/ottogroup/palladium
- Documentation including installation instructions and tutorial: http://palladium.readthedocs.org
- Mailing list: https://groups.google.com/forum/#!forum/pld-list
- Maintainer: Andreas Lattner

CHAPTER 2

User's Guide

This part of the documentation is mostly prose. It starts with installation instructions for setting up Palladium for development, then develops a simple pipeline for predicting classes in the Iris flower dataset dataset. It will explain the concepts behind Palladium as it develops the application.

Installation

Palladium requires Python 3.4 or better to run. If you are currently using an older version of Python, you might want to check the *FAQ entry about virtual environments*. Some of Palladium's dependencies such as `numpy`, `scipy` and `scikit-learn` may require a C compiler to install.

All of Palladium's dependencies are listed in the `requirements.txt` file. You can use either `pip` or `conda` to install the dependencies from this file.

For most installations, it is recommended to install Palladium and its dependencies inside a virtualenv or a conda environment. The following commands assume that you have your environment active.

Install from PyPI

It is a good practice to install dependencies with exactly the same version numbers that the release was made with. You can find the `requirements.txt` that defines those version numbers in the top level directory of Palladium's source tree or can download it here: `requirements.txt`. You can install the dependencies with the following command:

```
pip install -r requirements.txt
```

In order to install Palladium from PyPI, simply run:

```
pip install palladium
```

Install from binstar

For installing Palladium with *conda install*, you have to add the following binstar channel first:

```
conda config --add channels https://conda.binstar.org/ottogroup
conda install palladium
```

Note: Right now, there are only versions for *linux-64* and *osx-64* platforms available at our binstar channel.

Install from source

Download and navigate to your copy of the Palladium source, then run:

```
cd palladium
pip install -r requirements.txt
```

To install the Palladium package itself, run:

```
python setup.py install  # or 'setup.py dev' if you intend to develop Palladium itself
```

If you prefer conda over using pip, run these commands instead to install:

```
cd palladium
conda create -n palladium python=3 --file requirements.txt  #create conda environment
source activate palladium  # activate conda environment
python setup.py install
```

Note: The *virtualenv* or *conda create* and *source activate* commands above generate and activate an environment where specific Python package versions can be installed for a project without interferring with other Python projects. This environment has to be activated in each context you want to call Palladium scripts (e.g., in a shell). So if you run into problems finding the Palladium scripts or get errors regarding missing packages, it might be worth checking if you have activated the corresponding environment. If you want to deactivate an environment, simply run *deactivate* (or *source deactivate* for conda environments).

Note: If you intend to develop Palladium itself or if you want to run the tests, you additionally need to install the `requirements-dev.txt` with `pip install -r requirements-dev.txt` (or `conda install --file requirements-dev.txt` in the Anaconda setting).

Once you have Palladium installed, you should be able to use the `pld-version` command and find out which version of Palladium you're using:

```
pld-version
```

Now that you've successfully installed Palladium, it's time to head over to the *Tutorial* to learn about what it can do for you.

Tutorial

Run the Iris example

In this first part of the tutorial, we will run the simple Iris example that is included in the source distribution of Palladium. The Iris data set consists of a number of entries describing Iris flowers of three different types and is often used as an introductory example for machine learning.

It is assumed that you have already run through the *Installation*. You can either download the files needed for the tutorial here: `config.py` and `iris.data`. Alternatively, you can find the files in the source tree of Palladium. It should include the iris example in the `examples/iris` folder. Navigate to that folder and list its contents:

```
cd examples/iris
ls
```

You will notice that there are two files here. One is `iris.data` which is a CSV file with the dataset we want to train with. For each training example, `iris.data` defines four features and one of the three classes to predict.

The other file, `config.py` is our Palladium configuration file. It has all the configuration necessary to load the dataset CSV file and to train it with a random forest classifier.

All the following commands require you to set an environment variable to point to the `config.py` file. In general, when using any of Palladium's scripts, you will want to have that environment variable set and pointing to your current project's `config.py`. Using Bash, you could set the `PALLADIUM_CONFIG` environment variable so that it is picked up by subsequent calls to Palladium like so:

```
export PALLADIUM_CONFIG=config.py
```

Now we're all set to fit our Iris model:

```
pld-fit
```

This command will print a number of lines and hopefully finish with the message `Wrote model with version 1`. If you list the contents of the directory you are in again, you will notice that there is a new file called `iris-model.db`. This is the *SQLite database* that Palladium created and saved our trained model in. We can now use this trained model and test it on a held-out test set:

```
pld-test
```

This will output an accuracy score, which should be something around 96 percent.

If you run `pld-fit` again, you'll notice that it outputs `Wrote model with version 2`. The next call to `pld-test` will use that newer model to run tests. To test the first model that you trained, run:

```
pld-test --model-version=1
```

Let us try and use the web service that is included with Palladium to use our trained model to generate predictions. Run this command to bring up the web server:

```
pld-devserver
```

And now type this address into your browser's address bar (assuming that you're running the server locally):

> http://localhost:5000/predict?sepal%20length=6.3&sepal%20width=2.5&petal%20length=4.9&petal%20width=1.5

The server should print out something like this:

```
{
    "result": "Iris-virginica",
    "metadata": {
        "service_name": "iris",
        "error_code": 0,
        "status": "OK",
        "service_version": "0.1"
    }
}
```

At this point we've already run through the palladium important scripts that Palladium provides.

Understand Iris' config.py

In this section, we'll take a closer look at the Iris example's `config.py` file and how it wires together the components that we use to train and predict on the Iris dataset.

Open up the `config.py` file inside the `examples/iris` directory in Palladium's source folder and let us now walk step-by-step through the entries of this file.

Note: Despite the `.py` file ending, `config.py` is not conventional Python source code. The file ending exists to help your editor to use Python syntax highlighting. But all that `config.py` consists of is a single Python dictionary.

Dataset loaders

The first configuration entry we'll find inside `config.py` is something called `dataset_loader_train`. This is where we configure our dataset loader that helps us load the training data from the CSV file with the data, and define which rows should be used as data and target values. The first entry inside `dataset_loader_train` defines the type of dataset loader we want to use. That is *`palladium.dataset.Table`*:

```
'dataset_loader_train': {
    '__factory__': 'palladium.dataset.Table',
```

The rest what is inside the `dataset_loader_train` are the keyword arguments that are used to initialize the *`Table`* component. The full definition of `dataset_loader_train` looks like this:

```
'dataset_loader_train': {
    '__factory__': 'palladium.dataset.Table',
    'path': 'iris.data',
    'names': [
        'sepal length',
        'sepal width',
        'petal length',
        'petal width',
        'species',
        ],
    'target_column': 'species',
    'sep': ',',
    'nrows': 100,
    }
```

You can now take a look at *`Table`*'s API to find out what parameters a Table accepts and what they mean. But to summarize: the `path` is the path to the CSV file. In our case, this is the relative path to `iris.data`. Because our CSV file doesn't have the column names in the first line, we have to provide the column names using the `names` parameter. The `target_column` defines which of the columns should be used as the value to be predicted; this is the last column, which we named `species`. The `nrows` parameter tells *`Table`* to return only the first hundred samples from our CSV file.

If you take a look at the next section in the config file, which is `dataset_loader_test`, you will notice that it is very similar to the first one. In fact, the only difference between `dataset_loader_train` and `dataset_loader_test` is that the latter uses a different subset of the samples available in the same CSV file.

So instead of using `nrows`, `dataset_loader_test` uses the `skiprows` parameter and thus skips the first hundred examples (in order to separate training and testing data):

```
'skiprows': 100,
```

Under the hood, `Table` uses `pandas.io.parsers.read_table()` to do the actual loading. Any additional named parameters passed to `Table` are passed on to `read_table()`. That is the case for the `sep` parameter in our example, but there are a lot of other useful options, too, like `usecols`, `skiprows` and so on.

Palladium also includes a dataset loader for loading data from an SQL database: `palladium.dataset.SQL`.

But if you find yourself in need to write your own dataset loader, then that is pretty easy to do: Take a look at Palladium's `DatasetLoader` interface that documents how a `DatasetLoader` like `Table` needs to look like.

Model

The next section in our Iris configuration example is `model`. Here we define which machine learning algorithm we intend to use. In our case we'll be using a logistic regression classifier out of scikit-learn:

```
'model': {
    '__factory__': 'sklearn.linear_model.LogisticRegression',
    'C': 0.3,
    },
```

Notice how we parametrize `LogisticRegression` with the regularization parameter `C` set to `0.3`. To find out which other parameters exist for the `LogisticRegression` classifier, refer to the scikit-learn docs.

If you've written your own scikit-learn estimator before, you'll already know how to write your own `palladium.interfaces.Model` class. You'll want to implement `fit()` for model fitting, and `predict()` for prediction of target values. And possibly `predict_proba()` if you're dealing with class probabilities.

If you need to do pre-processing of your data, say scaling, value imputation, feature selection, or the like, before you pass the data into the ML algorithm (such as the `LogisticRegression` classifier), you'll want to take a look at scikit-learn pipelines. A Palladium `model` is not bound to be a simple estimator class; it can be a composite of several pre-processing steps or transformations, and the algorithm combined.

At this point, feel free to change the configuration file to maybe try out different values for *C*. Can you find a setting for *C* that produces better accuracy?

Grid search

Finding the right set of hyper parameters for your model can be tedious. That is where grid search comes in. Using grid search, we can quickly try out a few parameters and use cross-validation to see which of them work best.

Try running `pld-grid-search` and see what happens:

```
pld-grid-search
```

At the end, you should see something like this output:

```
[mean: 0.95000, std: 0.05138, params: {'C': 1.0},
 mean: 0.91000, std: 0.05022, params: {'C': 0.3},
 mean: 0.84000, std: 0.06408, params: {'C': 0.1}]
```

What happened? We just tried out three different values for *C*, and used a three-fold cross-validation to determine the best setting. The first line is the winner. It tells us that the mean cross-validation accuracy of the model with *C* set to `1.0` is `0.95` and that the standard deviation between accuracies in the cross-validation folds is `0.05138`.

Let us take a look at the configuration of `grid_search`:

```
'grid_search': {
    'param_grid': {
        'C': [0.1, 0.3, 1.0],
        },
    'verbose': 4,
    }
```

What parameters should be checked can be specified in the entry `param_grid`. If more than one parameter with sets of values to check are provided, all possible combinations are explored by grid search. `verbose` allows to set the level for grid search messages. It is possible to set other parameters of grid search, e.g., how many jobs to be run in parallel can be specified in *n_jobs* (if set to -1, all cores are used).

Palladium uses `sklearn.grid_search.GridSearchCV` to do the actual work. Thus, you'll want to take a look at the scikit-learn docs for grid search to understand what these parameters mean and what other parameters exist for `grid_search`.

Model persister

Usually we'll want the `pld-fit` command to save the trained model to disk.

The `model_persister` in the Iris `config.py` file is set up to save those models into a SQLite database. Let us take a look at that part of the configuration:

```
'model_persister': {
    '__factory__': 'palladium.persistence.CachedUpdatePersister',
    'update_cache_rrule': {'freq': 'HOURLY'},
    'impl': {
        '__factory__': 'palladium.persistence.Database',
        'url': 'sqlite:///iris-model.db',
        },
    },
```

The `palladium.persistence.CachedUpdatePersister` wraps the persister actually responsible for reading and writing models. It is possible to provide an update rule which specifies intervals to update the model. In the configuration above, the *update_cache_rrule* is set to an hourly update (in real applications, the frequency will palladium likely be much lower like daily or weekly). For details how to define these rules we refer to the python-dateutil docs. If no *update_cache_rrule* is provided, the model will not be updated automatically. The *impl* entry of this model persister specifies the actual persister to be wrapped.

The `palladium.persistence.Database` persister takes a single argument `url` which is the URL of the database to save the fitted model into. It will automatically create a table called `models` if such a table doesn't exist yet. Please refer to the SQLAlchemy docs for details on which databases are supported, and how to form the database URL.

Palladium ships with another model persister called `palladium.persistence.File` that writes pickles to the file system. If you want to store your model anywhere else, or if you do not use Python's pickle but something else, you might want to take a look at the *`ModelPersister`* interface, which describes the necessary methods. The location for storing the files can be chosen freely. However, the path has to contain a placeholder for adding the model's version:

```
'model_persister': {
    '__factory__': 'palladium.persistence.CachedUpdatePersister',
    'impl': {
        '__factory__': 'palladium.persistence.File',
        'path': 'model-{version}.pickle',
        },
    },
```

Predict service

The next component in the Iris example configuration is called predict_service. The *palladium.interfaces.PredictService* is the workhorse behind what us happening in the /predict HTTP endpoint. Let us take a look at how it is configured:

```
'predict_service': {
    '__factory__': 'palladium.server.PredictService',
    'mapping': [
        ('sepal length', 'float'),
        ('sepal width', 'float'),
        ('petal length', 'float'),
        ('petal width', 'float'),
        ],
    }
```

Again, the specific implementation of the predict_service that we use is specified through the __factory__ setting.

The mapping defines which request parameters are to be expected. In this example, we expect a float number for each of sepal length, sepal width, petal length, petal width. Note that this is exactly the order in which the data was fed into the algorithm for model fitting.

An example request might then look like this (assuming that you're running a server locally on port 5000):

http://localhost:5000/predict?sepal%20length=6.3&sepal%20width=2.5&petal%20length=4.9&petal%20width=1.5

The *palladium.server.PredictService* implementation that we use in this example has some more settings.

Its responsibility is also to create an HTTP response. In our example, if the prediction was successful (i.e., no errors whatsoever occurred), then the *PredictService* will generate a JSON response with an HTTP status code of 200:

```
{
    "result": "Iris-virginica",
    "metadata": {
        "service_name": "iris",
        "error_code": 0,
        "status": "OK",
        "service_version": "0.1"
    }
}
```

In case of a malformed request, you will see a status code of 400 and this response body:

```
{
    "metadata": {
        "service_name": "iris",
        "error_message": "BadRequest: ...",
        "error_code": -1,
        "status": "ERROR",
        "service_version": "0.1"
    }
}
```

If you want the predict service to work differently, then chances are that you get away subclassing from the *PredictService* class and override one of its methods. E.g. to change the way that API responses to the web look like, you would override the *response_from_prediction()* and *response_from_exception()* methods, which are responsible for creating the JSON responses.

Customizing entry points for predict services

Predict service specifications can be customized by setting the entry_point and decorator_list_name in the configuration. It is also possible to specify more than one predict service which can be reached by different entry points, e.g., if a model should be used in two different contexts with different parameters or response formats. This is an example how to specify two predict services with different entry points:

```
'predict_service1': {
    '__factory__': 'mypackage.server.PredictService',
    'mapping': [
        ('sepal length', 'float'),
        ('sepal width', 'float'),
        ('petal length', 'float'),
        ('petal width', 'float'),
        ],
    'entry_point': '/predict',
    'decorator_list_name': 'predict_decorators',
    }
'predict_service2': {
    '__factory__': 'mypackage.server.PredictServiceID',
    'mapping': [
        ('id', 'int'),
        ],
    'entry_point': '/predict-by-id',
    'decorator_list_name': 'predict_decorators_id',
    }
```

It is not necessary that the decorator list as specified by decorator_list_name is defined in the configuration; if it does not exist, no decorators will be used for this predict service.

Note: If entry_point and decorator_list_name are omitted, /predict and predict_decorators will be used as default values, leading to the same behavior as it was the case in earlier Palladium versions.

Implementing the model as a pipeline

As mentioned in the *Model* section, it is entirely possible to implement your own machine learning model and use it. Remember that the only interface our model needed to implement was *palladium.interfaces.Model*. That means we can also use a scikit-learn Pipeline to do the job. Let us extend our Iris example to use a pipeline with two elements: a sklearn.preprocessing.PolynomialFeatures transform and a sklearn.linear_model.LogisticRegression classifier. To do this, let us create a file called iris.py in the same folder as we have our config.py with the following contents:

```
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

def model(**kwargs):
    pipeline = Pipeline([
        ('poly', PolynomialFeatures()),
        ('clf', LogisticRegression()),
        ])
    pipeline.set_params(**kwargs)
    return pipeline
```

The special **kwargs argument allows us to pass configuration options for both the poly and the clf elements of our pipeline in the configuration file. Let us try this: we change the model entry in config.py to look like this:

```
'model': {
    '__factory__': 'iris.model',
    'clf__C': 0.3,
    },
```

Just like in our previous example, we are setting the `C` hyper parameter of our `LogisticRegression` to be `0.3`. However, this time, we have to prefix the parameter name by `clf__` to tell the pipeline that we want to the set a parameter of the `clf` part of the pipeline. If you want to used grid search with this pipeline, keep in mind that you will also need to adapt the parameter's name in the grid search section to *clf_C*.

It is also possible to use nested lists in configurations. With this feature, pipelines can be also defined purely through the configuration file, e.g.:

```
'model': {
    '__factory__': 'sklearn.pipeline.Pipeline',
    'steps': [['clf', {'__factory__': 'sklearn.linear_model.LinearRegression'}],
    ],
},
```

Deployment

In this section, you will find information on how to run Palladium-based application with another web server instead of Flask's built in solution, and how to benchmark your service. Additionally, you will find information on how to use provided scripts to automatically generate Docker images of your service and how to deploy such Docker images using Mesos / Marathon.

Web server installation

Palladium uses HTTP to respond to prediction requests. Through use of the *WSGI protocol* (via Flask), Palladium can be used together with a variety of web servers.

For convenience, a web server is included for development purposes. To start the built-in web server, use the *pld-devserver* command.

For production use, you probably want to use something faster and more robust. Many options are listed in the Flask deployment docs. If you follow any of these instructions, be aware that the Flask app in Palladium is available as `palladium.wsgi:app`. So here's how you would start an Palladium prediction server using gunicorn:

```
export PALLADIUM_CONFIG=/path/to/myconfig.py
gunicorn palladium.wsgi:app
```

An example configuration to use nginx to proxy requests to gunicorn is also available. It can be used without modification for our example and has to be made available in the */etc/nginx/sites-enabled/* folder and is active after a restart of *nginx*. For convenience it is reprinted here:

```
server {
    listen 80;

    server_name _;

    access_log  /var/log/nginx/access.log;
    error_log  /var/log/nginx/error.log;

    location / {
        proxy_pass         http://127.0.0.1:8000/;
        proxy_redirect     off;

        proxy_set_header   Host             $host;
        proxy_set_header   X-Real-IP        $remote_addr;
        proxy_set_header   X-Forwarded-For  $proxy_add_x_forwarded_for;
    }
}
```

Note: In previous versions (Palladium 1.0.1 and older), the Flask app was accessed via `palladium.server:app`. This was changed in order to initialize the configuration during start-up.

Benchmarking the service with Apache Benchmark

In order to benchmark the response time of a service, existing tools like Apache Benchmark (ab) or siege can be used. They can be installed using install packages, e.g., for Ubuntu with *apt-get install apache2-utils* and *sudo apt-get install siege*.

If a web server with the Iris predict service is running (either using the built-in *pld-devserver* or a faster solution as described in *Web server installation*), the *ab* benchmarking can be run as follows:

```
ab -n 1000 -c 10 "http://localhost:5000/predict?
                  sepal%20length=5.2&sepal%20width=3.5&
                  petal%20length=1.5&petal%20width=0.2"
```

In this *ab* call, it is assumed that the web server is available at port 5000 of localhost and 1000 requests with 10 concurrent requests at a time are sent to the web server. The output provides a number of statistics about response times of the calls performed.

Note: If there is an error in the sample request used, response times might be suspiciously low. If very low response times occur, it might be worth manually checking the corresponding response of used request URL.

Note: *ab* does not allow to use different URLs in a benchmark run. If different URLs are important for benchmarking, either siege or a multiple URL patch for ab could be used

Building a Docker image with your Palladium application

Building the Palladium base image

Here's instructions on how to build the Palladium base image. This isn't usually necessary, as you'll probably want to just use the released base images for Palladium and add your application on top, see *Building a Palladium app image*.

A `Dockerfile` is available in the directory `addons/docker/palladium_base_image` for building a base image. You can download the file here: `Dockerfile`.

Run `docker build` in your terminal:

```
sudo docker build -t myname/palladium-base:1.1.0.1 .
```

A Docker image with the name `myname/palladium-base:1.1.0.1` should now be created. You can check this with:

```
sudo docker images
```

Building a Palladium app image

Palladium has support for quickly building a Docker image to run your own application based on the Palladium base image. The Palladium base image can be pulled from Docker Hub as follows:

```
docker pull ottogroup/palladium-base
```

As an example, let's build a Docker image for the Iris example that's included in the source. We'll use the Palladium base image for version 1.0, and we'll name our own image `my-palladium-app`. Thus, we invoke `pld-dockerize` like so:

```
pld-dockerize palladium-src/examples/iris ottogroup/palladium-base:1.1.0.1 myname/my-palladium-app:1
```

This command will in fact create two images: one that's called `my-palladium-app`, another one that's called `my-palladium-app-predict`. The latter extends the former by adding calls to automatically fit your model and start a web server.

By default `pld-dockerize` will create the Dockerfile files *and* create the Docker containers. You may want to create the Dockerfile files only using the `-d` flag, and then modify files `Dockerfile-app` and `Dockerfile-predict` according to your needs.

Your application's folder (`examples/iris` in this case) should look like this:

```
.
|--- config.py
|--- setup.py (optional)
|--- requirements.txt (optional)
'--- python_packages (optional)
     |--- package1.tar.gz
     |--- package2.tar.gz
     '--- ...
```

You may put additional requirements as shown into a `python_packages` subdirectory.

To test your image you can:

1. Create app images using `pld-dockerize` as shown above.
2. Run the "predict" image (e.g., `my-palladium-app-predict` if you used `my-palladium-app` to create the image), and map the Docker container's port 8000 to a local port (e.g., 8001):

```
    sudo docker run -d -p 8001:8000 my-palladium-app-predict
```

3. Your application should be up and running now. You should be able to access this URL: http://localhost:8001/alive

Setup Palladium with Mesos / Marathon and Docker

This section describes how to setup Mesos / Marathon with a containerized Palladium application. If you have not built a docker image with your Palladium application yet, you can follow the instructions that are provided in the *Building a docker image with your Palladium application* section.

For the installation of Mesos and Marathon you can follow the guide on Mesosphere. If you want to try it out locally first, we recommend to set up a single node Mesosphere cluster. Before adding a new application to Marathon you need to make sure that the Mesos slaves and Marathon are configured properly to work with Docker. To do so, follow the steps as described in the Marathon documentation.

An easy way to add a new application to Marathon is to use its REST API. For this task you need a json file which contains the relevant information for Marathon. A basic example of the json file could look like this:

```
{
    "id": "<app_name>",
    "container": {
        "docker": {
            "image": "<owner/palladium-app-name:version>",
            "network": "BRIDGE",
            "parameters": [
                {"key": "link", "value":"<some_container_to_link>"}
            ],
            "portMappings": [
                { "containerPort": 8000, "hostPort": 0, "servicePort": 9000,
                  "protocol": "tcp" }
            ]
        },
        "type": "DOCKER",
        "volumes": [
            {
                "containerPath": "/path/in/your/container",
                "hostPath": "/host/path",
                "mode": "RO"
            }
        ]
    },
    "cpus": 0.2,
    "mem": 256.0,
    "instances": 3,
    "healthChecks": [
        {
            "protocol": "HTTP",
            "portIndex": 0,
            "path": "/alive",
            "gracePeriodSeconds": 5,
            "intervalSeconds": 20,
            "maxConsecutiveFailures": 3
        }
    ],
    "upgradeStrategy": {
        "minimumHealthCapacity": 0.5
    }
}
```

You have to replace the Docker image name, port number (currently set to 8000) and - if there is any dependency - specify links to other containers. If you have a Docker image of the Iris service available (named *user/palladium-iris-predict:0.1*), you can use this file:

```
{
  "id": "palladium-iris",
    "container": {
        "docker": {
            "image": "user/palladium-iris-predict:0.1",
            "network": "BRIDGE",
            "parameters": [
            ],
            "portMappings": [
                { "containerPort": 8000, "hostPort": 0, "servicePort": 9000,
                  "protocol": "tcp" }
            ]
        },
        "type": "DOCKER",
        "volumes": [
         ]
    },
    "cpus": 0.2,
    "mem": 256.0,
    "instances": 3,
    "healthChecks": [
        {
            "protocol": "HTTP",
            "portIndex": 0,
            "path": "/alive",
            "gracePeriodSeconds": 5,
            "intervalSeconds": 20,
            "maxConsecutiveFailures": 3
        }
    ],
    "upgradeStrategy": {
        "minimumHealthCapacity": 0.5
    }
}
```

Now you can send the json application file to Marathon via POST (assuming Marathon is available at *localhost:8080*:

```
curl -X POST -H "Content-Type: application/json" localhost:8080/v2/apps
     -d @<path-to-json-file>
```

You can see the status of your Palladium service instances using the Marathon web user interface (available at *http://localhost:8080* if you run the single node installation mentioned above) and can scale the number of instances as desired. Marathon keeps track of the Palladium instances. If a service instance breaks down, a new one will be started automatically.

Authorization

Sometimes you will want the Palladium web service's entry points */predict* and */alive* to be secured by OAuth2 or similar. Defining `predict_decorators` and `alive_decorators` in the Palladium configuration file allows you to put any decorators in place to check authentication.

Let us first consider an example where you want to use *HTTP Basic Auth* to guard the entry points. Consider this code taken from the Flask snippets repository:

```
# file: mybasicauth.py

from functools import wraps
from flask import request, Response


def check_auth(username, password):
    """This function is called to check if a username /
    password combination is valid.
    """
    return username == 'admin' and password == 'secret'

def authenticate():
    """Sends a 401 response that enables basic auth"""
    return Response(
    'Could not verify your access level for that URL.\n'
    'You have to login with proper credentials', 401,
    {'WWW-Authenticate': 'Basic realm="Login Required"'})

def requires_auth(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        auth = request.authorization
        if not auth or not check_auth(auth.username, auth.password):
            return authenticate()
        return f(*args, **kwargs)
    return decorated
```

The `requires_auth` can now be used to decorate Flask views to guard them with basic authentication. Palladium allows us to add decorators to the */predict* and */alive* views that it defines itself. To do this, we only need to add this bit to the Palladium configuration file:

```
'predict_decorators': [
    'mybasicauth.requires_auth',
    ],

'alive_decorators': [
    'mybasicauth.requires_auth',
    ],
```

Of course, alternatively, you could set up your mod_wsgi server to take care of authentication.

Using Flask-OAuthlib to guard the two views using OAuth2 follows the same pattern. We will configure and use the `flask_oauthlib.provider.OAuth2Provider` for security. In our own package, we might have an instance of `OAuth2Provider` and a `require_oauth` decorator defined thus:

```
# file: myoauth.py

from flask_oauthlib.provider import OAuth2Provider
from palladium.server import app


oauth = OAuth2Provider(app)

# more setup code here... see Flask-OAuthlib

require_oauth = oauth.require_oauth('myrealm')
```

Alternatively, to get more decoupling from Palladium's Flask `app`, you can use the following snippet inside your Palladium configuration and assign the Flask app to `OAuth2Provider` at application startup:

```
'oauth_init_app': {
    '__factory__': 'myoauth.oauth.init_app',
    'app': 'palladium.server.app',
    },
```

Now, to guard, */predict* and */alive* with the previously defined `require_oauth`, add this to your configuration:

```
'predict_decorators': [
    'myoauth.require_oauth'
    ],

'alive_decorators': [
    'myoauth.require_oauth'
    ],
```

Web service

Palladium includes an HTTP service that can be used to make predictions over the web using models that were trained with the framework. There are two endpoints: */predict*, that makes predictions, and */alive* which provides a simple health status.

Predict

The */predict* service uses HTTP query parameters to accept input features, and outputs a JSON response. The number and types of parameters depend on the application. An example is provided as part of the *Tutorial*.

On success, */predict* will always return an HTTP status of 200. An error is indicated by either status 400 or 500, depending on whether the error was caused by malformed user input, or by an error on the server.

The `PredictService` must be configured to define what parameters and types are expected. Here is an example configuration from the *Tutorial*:

```
'predict_service': {
    '__factory__': 'palladium.server.PredictService',
    'mapping': [
        ('sepal length', 'float'),
        ('sepal width', 'float'),
        ('petal length', 'float'),
        ('petal width', 'float'),
        ],
    },
```

An example request might then look like this (assuming that you're running a server locally on port 5000):

> http://localhost:5000/predict?sepal%20length=6.3&sepal%20width=2.5&petal%20length=4.9&petal%20width=1.5

The usual output for a successful prediction has both a `result` and a `metadata` entry. The `metadata` provides the service name and version as well as status information. An example:

```
{
    "result": "Iris-virginica",
    "metadata": {
        "service_name": "iris",
        "error_code": 0,
        "status": "OK",
        "service_version": "0.1"
        }
}
```

An example that failed contains a `status` set to `ERROR`, an `error_code` and an `error_message`. There is generally no `result`. Here is an example:

```
{
    "metadata": {
        "service_name": "iris",
        "error_message": "BadRequest: ...",
        "error_code": -1,
        "status": "ERROR",
        "service_version": "0.1"
        }
}
```

It's also possible to send a POST request instead of GET and predict for a number of samples at the same time. Say you want to predict for the class for two Iris examples, then your POST body might look like this:

```
[
  {"sepal length": 6.3, "sepal width": 2.5, "petal length": 4.9, "petal width": 1.5},
  {"sepal length": 5.3, "sepal width": 1.5, "petal length": 3.9, "petal width": 0.5}
]
```

The response will generally look the same, with the exception that now there's a list of predictions that's returned:

```
{
    "result": ["Iris-virginica", "Iris-versicolor"],
    "metadata": {
        "service_name": "iris",
        "error_code": 0,
        "status": "OK",
        "service_version": "0.1"
        }
}
```

Should a different output format be desired than the one implemented by *`PredictService`*, it is possible to use a different class altogether by setting an appropriate `__factory__` (though that class will likely derive from *`PredictService`* for reasons of convenience).

A list of decorators may be configured such that they will be called every time the */predict* web service is called. To configure such a decorator, that will act exactly as if it were used as a normal Python decorator, use the `predict_decorators` list setting. Here is an example:

```
'predict_decorators': [
    'my_package.my_predict_decorator',
    ],
```

Alive

The */alive* service implements a simple health check. It'll provide information such as the `palladium_version` in use, the current `memory_usage` by the web server process, and all metadata that has been defined in the configuration under the `service_metadata` entry. Here is an example for the Iris service:

```
{
    "palladium_version": "0.6",
    "service_metadata": {
        "service_name": "iris",
        "service_version": "0.1"
    },
    "memory_usage": 78,
    "model": {
        "updated": "2015-02-18T10:13:50.024478",
        "metadata": {
            "version": 2,
            "train_timestamp": "2015-02-18T09:59:34.480063"
        }
    }
}
```

/alive can optionally check for the presence of data loaded into the process' cache (`process_store`). That is because some scenarios require the model and/or additional data to be loaded in memory before they can answer requests efficiently (cf. `palladium.persistence.CachedUpdatePersister` and *`palladium.dataset.ScheduledDatasetLoader`*).

Say you expect the `process_store` to be filled with a `data` entry (because maybe you're using *`ScheduledDatasetLoader`*) before you're able to answer requests. And you want */alive* to return an error status (of *503*) when that data hasn't been loaded yet, then you'd add to your configuration the following entry:

```
'alive': {
    'process_store_required': ['data'],
    },
```

Scripts

Palladium includes a number of command-line scripts, many of which you may have already encountered in the *Tutorial*.

pld-fit: *train models*

See also:

- *Run the Iris example*

pld-test: *test models*

```
Test a model.

Uses 'dataset_loader_test' and 'model_persister' from the
configuration to load a test dataset to test the accuracy of a trained
model with.

Usage:
  pld-test [options]

Options:
  -h --help                  Show this screen.

  --model-version=<version>  The version of the model to be tested. If
                             not specified, the newest model will be used.
```

See also:

- *Run the Iris example*

pld-devserver: *serve the web API*

```
Serve the web API for development.

Usage:
  pld-devserver [options]

Options:
  -h --help               Show this screen.

  --host=<host>           The host to use [default: 0.0.0.0].

  --port=<port>           The port to use [default: 5000].

  --debug=<debug>         Whether or not to use debug mode [default: 0].
```

See also:

- *Run the Iris example*
- *Deployment*

pld-stream: *make predictions through stdin and stdout*

```
Start the streaming server, which listens to stdin, processes line
by line, and returns predictions.

The input should consist of a list of json objects, where each object
will result in a prediction.  Each line is processed in a batch.

Example input (must be on a single line):

  [{"sepal length": 1.0, "sepal width": 1.1, "petal length": 0.7,
    "petal width": 5}, {"sepal length": 1.0, "sepal width": 8.0,
    "petal length": 1.4, "petal width": 5}]

Example output:

  ["Iris-virginica","Iris-setosa"]

An input line with the word 'exit' will quit the streaming server.

Usage:
  pld-stream [options]

Options:
  -h --help                  Show this screen.
```

pld-grid-search: *find optimal hyperparameters*

See also:

- *Grid search*

pld-list: *list available models*

```
List information about available models.

Uses the 'model_persister' from the configuration to display a list of
models and their metadata.

Usage:
  pld-list [options]

Options:
  -h --help                  Show this screen.
```

pld-admin: *administer available models*

pld-version: *display version number*

```
Print the version number of Palladium.

Usage:
  pld-version [options]

Options:
  -h --help                Show this screen.
```

pld-upgrade: *upgrade database*

```
Upgrade the database to the latest version.

Usage:
  pld-ugprade [options]

Options:
  --from=<v>               Upgrade from a specific version, overriding
                           the version stored in the database.

  --to=<v>                 Upgrade to a specific version instead of the
                           latest version.

  -h --help                Show this screen.
```

See also:

- *Upgrading*

Upgrading

Upgrading the database

Changes between Palladium versions may require upgrading the model database or similar. These upgrades are handled automatically by the `pld-upgrade` script. Before running the script, make sure you've set the `PALLADIUM_CONFIG` environment variable, then simply run:

```
pld-upgrade
```

In some rare situations, it may be necessary to run the upgrade steps of specific versions only. `pld-upgrade` supports passing `--from` and `--to` options for that purpose. As an example, if you only want to run the upgrade steps between version `0.9` and `1.0`, this is how you'd invoke `pld-upgrade`:

```
pld-upgrade --from=0.9.1 --to=1.0
```

Upgrading the Database persister from version 0.9.1 to 1.0

Users of `palladium.persistence.Database` that are upgrading from version 0.9.1 to a more recent version (e.g. 1.0) are required to invoke pld-upgrade with an explicit `--from` version like so:

```
pld-upgrade --from=0.9.1
```

Backward incompatibilities in code

The development team makes an effort to try and keep the API backward compatibility, and only gradually deprecate old code where necessary. However, some changes between major Palladium versions still introduce backward incompatibilities and potentially require you to update your Palladium plug-ins.

Breaking changes between 0.9.1. and 1.0

Backward incompatible changes between 0.9.1. and 1.0 are of concern only to users who have implemented their own version of *`palladium.server.PredictService`*.

`palladium.server.PredictService.sample_from_request()` has been replaced by the very similar *`sample_from_data()`*. The new method now accepts a `data` argument instead of `request`. `data` is the equivalent of the former `request.args`. Similarly, `palladium.server.PredictService.params_from_request()` has been replaced by *`params_from_data()`*. The latter now also accepts `data` instead of `request`, which again is the equivalent of the former `request.data`.

R support

Palladium has support for using *`DatasetLoader`* and *`Model`* objects that are programmed in the R programming language.

To use Palladium's R support, you'll have to install R and the Python rpy2 package.

An example is available in the `examples/R` folder in the source tree of Palladium (`config.py`, `iris.R`, `iris.data`). It contains an example of a very simple dataset loader and model implemented in R:

```
packages_needed <- c("randomForest")
packages_missing <-
  packages_needed[!(packages_needed %in% installed.packages()[,"Package"])]
if(length(packages_missing))
```

```
  install.packages(packages_missing, repos='http://cran.uni-muenster.de')

library(randomForest)

dataset <- function() {
    x <- iris[,1:4]
    y <- as.factor(iris[,5])
    list(x, y)
}

train.randomForest <- function(x, y) {
    randomForest(x, as.factor(y))
}
```

When configuring a dataset loader that is programmed in R, use the `palladium.R.DatasetLoader`. An example:

```
'dataset_loader_train': {
    '__factory__': 'palladium.R.DatasetLoader',
    'scriptname': 'iris.R',
    'funcname': 'dataset',
    },
```

The `scriptname` points to the R script that contains the function `dataset`.

R models are configured very similarly, using `palladium.R.ClassificationModel`:

```
'model': {
    '__factory__': 'palladium.R.ClassificationModel',
    'scriptname': 'iris.R',
    'funcname': 'train.randomForest',
    'encode_labels': True,
    },
```

The configuration options are the same as for `DatasetLoader` except for the `encode_labels` option, which when set to `True` says that we would like to use a `sklearn.preprocessing.LabelEncoder` class to be able to deal with string target values. Thus `[’Iris-setosa’, ’Iris-versicolor’, ’Iris-virginica’]` will be visible to the R model as `[0, 1, 2]`.

It is okay to use a `DatasetLoader` that is programmed in Python together with an R model.

Julia support

Palladium has support for using `Model` objects that are implemented in the Julia programming language.

To use Palladium’s Julia support, you’ll have to install Julia 0.3 or better and the julia Python package. You’ll also need to install the PyCall library in Julia:

```
$ julia -e 'Pkg.add("PyCall"); Pkg.update()'
```

The following example also relies on the SVM Julia package. This is how you can install it:

```
$ julia -e 'Pkg.add("StatsBase"); Pkg.add("SVM"); Pkg.update()'
```

Warning: The latest PyCall version from GitHub is known to have significant performance issues. It is recommended that you install revision 120fb03 instead. To do this on Linux, change into your `~/.julia/v0.3/PyCall` directory and issue the necessary `git checkout` command:

```
cd ~/.julia/v0.3/PyCall
git checkout 120fb03
```

Let's now take a look at the example on how to use a model written in Julia in the `examples/julia` folder in the source tree of Palladium (`config.py`, `iris.data`). The configuration in that example defines the model to be of type *`palladium.julia.ClassificationModel`*:

```
'model': {
    '__factory__': 'palladium.julia.ClassificationModel',
    'fit_func': 'SVM.svm',
    'predict_func': 'SVM.predict',
    }
```

There's two required arguments to *`ClassificationModel`* and they're the dotted path to the Julia function used for fitting, and the equivalent for the Julia function that does the prediction. The complete description of available parameters is defined in the API docs:

class `palladium.julia.`**`AbstractModel`**(*fit_func*, *predict_func*, *fit_kwargs=None*, *predict_kwargs=None*, *encode_labels=False*)

`__init__`(*fit_func*, *predict_func*, *fit_kwargs=None*, *predict_kwargs=None*, *encode_labels=False*)

Instantiates a model with the given *fit_func* and *predict_func* written in Julia.

Parameters

- **`fit_func`** (*str*) – The dotted name of the Julia function to use for fitting. The function must take as its first two arguments the *X* and *y* arrays. All elements of the optional *fit_kwargs* dictionary will be passed on to the Julia function as keyword arguments. The return value of *fit_func* will be used as the first argument to *predict_func*.
- **`predict_func`** (*str*) – Similar to *fit_func*, this is the dotted name of the Julia function used for prediction. The first argument of this function is the return value of *fit_func*. The second argument is the *X* data array. All elements of the optional *fit_kwargs* dictionary will be passed on to the Julia function as keyword arguments. The return value of *predict_func* is considered to be the target array *y*.
- **`encode_labels`** (*bool*) – If set to *True*, the *y* target array will be automatically encoded using a `sklearn.preprocessing.LabelEncoder`, which is useful if you have string labels but your Julia function only accepts numeric labels.

Frequently asked questions

How do I contribute to Palladium?

Everyone is welcome to contribute to Palladium. You can help us to improve Palladium when you:

- Use Palladium and give us feedback or submit bug reports to GitHub.
- Improve existing code or documentation and send us a pull request on GitHub.
- Suggest a new feature, and possibly send a pull request for it.

In case you intend to improve or to add code to Palladium, we kindly ask you to:

- Include documentation and tests for new code.
- Ensure that all existing tests still run successfully.
- Ensure backward compatibility in the general case.

How do I configure where output is logged to?

Some commands, such as `pld-fit` use Python's own logging framework to print out useful information. Thus, we can configure where messages with which level are logged to. So maybe you don't want to log to the console but to a file, or you don't want to see debugging messages at all while using Palladium in production.

You can configure logging to suit your taste by adding a `'logging'` entry to the configuration. The contents of this entry are expected to follow the logging configuration dictionary schema. An example for this dictionary-based logging configuration format is available here.

How can I combine Palladium with my logging or monitoring solution?

Similar to adding authentication support, we suggest to use the different pluggable decorator lists in order to send logging or monitoring messages to the corresponding systems. You need to implement decorators which wrap the different functions and then send information as needed to your logging or monitoring solution. Every time, one of the functions is called, the decorators in the decorator lists will also be called and can thus be used to generate logging messages as needed. Let us assume you have implemented the decorators *my_app.log.predict*, *my_app.log.alive*, *my_app.log.fit*, *my_app.log.update_model*, and *my_app.log.load_data*, you can add them to your application by adding the following parts to the configuration:

```
'predict_decorators': [
    'my_app.log.predict',
    ],

'alive_decorators': [
    'my_app.log.alive',
    ],

'update_model_decorators': [
    'my_app.log.update_model',
    ],

'fit_decorators': [
    'my_app.log.fit',
    ],

'load_data_decorators': [
    'my_app.log.load_data',
    ],
```

How can I use Python 3 without messing up with my Python 2 projects?

If you currently use an older version of Python or even need this older version for other projects, you should take a look at virtual environments.

If you use the default Python version, you could use *virtualenv*:

1. Install Python 3 if not yet available
2. pip install virtualenv
3. mkdir <virtual_env_folder>
4. cd <virtual_env_folder>
5. virtualenv -p /usr/local/bin/python3 palladium
6. source <virtual_env_folder>/palladium/bin/activate

If you use Anaconda, you can use the conda environments which can be created and activated as follows:

1. conda create -n palladium python=3 anaconda
2. source activate palladium

Note: Palladium's installation documentation for Anaconda is already using a virtual environment including the requirements.txt.

After having successfully activated the virtual environment, this should be indicated by `(palladium)` in front of your shell command line. You can also check, if `python --version` points to the correct version. Now you can start installing Palladium.

Note: The environment has to be activated in each context you want to call Palladium scripts (e.g., in a shell). So if you run into problems finding the Palladium scripts or get errors regarding missing packages, it might be worth checking if you have activated the corresponding environment.

Where can I find information if there are problems installing numpy, scipy, or scikit-learn?

In the general case, the installation should work without problems if you are using Anaconda or have already installed these packages as provided with your operating system's distribution. In case there are problems during installation, we refer to the installation instructions of these projects:

- numpy / scipy
- scikit-learn

How do I use a custom cross validation iterator in my grid search?

Here’s an example of a grid search configuration that uses a `sklearn.cross_validation.StratifiedKFold` with a parameter `random_state=0`. Note that the required `y` parameter for `StratifiedKFold` is created and passed at runtime.

```
'grid_search': {
    'param_grid': {
        'C': [0.1, 0.3, 1.0],
        },
    'cv': {
        '__factory__': 'palladium.util.Partial',
        'func': 'sklearn.cross_validation.StratifiedKFold',
        'random_state': 0,
        },
    'verbose': 4,
    'n_jobs': -1,
    }
```

Is there any way to use a base configuration for different settings?

Since Palladium 1.0.1 you can use a list of configuration files in the `PALLADIUM_CONFIG` environment variable. During initialization, the list will be processed successively and the configuration dict will be updated using the configuration files from left to right.

If we use a list of two configurations like `PALLADIUM_CONFIG=mypath/mybaseconfig.py,mypath/myconfig.py`, the entries in the configuration files later in the list will override the ones defined before. E.g., if the contents of `mypath/mybaseconfig.py` is `{’a’: 42, ’b’: 6}`” and the contents of `mypath/myconfig.py` is `{’b’: 7, ’c’: 99}`, then the resulting configuration will be `{’a’: 42, ’b’: 7, ’c’: 99}`.

Can I somehow track the location of the configuration file?

You can use the *here* variable in a configuration file in order to refer to the location, e.g.:

```
{
    ...
    'config_path': here,
    ...
}
```

After initialization, this variable will point to the folder where the configuration file is located. In the example, eading the value of *config_path* will give you a string of the path to the folder:

```
from palladium.util import get_config
get_config()['config_path']  % -> config folder path
```

How can I use test Palladium components in a shell?

If you want to interactively check components of your Palladium configuration, you can access Palladium’s components as follows:

```
from palladium.util import initialize_config

config = initialize_config(__mode__='fit')
model = config['model']  # get model
X, y = config['dataset_loader_train']()  # load training data
# ...
```

You can also load the configuration to an interactive shell and access the components directly:

```
from code import InteractiveConsole
from pprint import pformat

from palladium.util import initialize_config

if __name__ == "__main__":
    config = initialize_config(__mode__='fit')
    banner = 'Palladium config:\n{}'.format(pformat(config))
    InteractiveConsole(config).interact(banner=banner)
```

In the interactive console, loading data and fitting a model can be done like this:

```
X, y = dataset_loader_train()
model.fit(X, y)
```

Note: Make sure, the `PALLADIUM_CONFIG` environment variable is pointing to a valid configuration file.

How can I access the active model in my code?

If you want to access the currently used model, you have to retrieve it via the `process_store` or you have to load it using the model persister:

```
from palladium.util import process_store
model = process_store.get('model')

from palladium.util import get_config
model = get_config()['model_persister'].read()
```

Note: `get_config()['model']` might not return the current active model as the entries in the configuration are not updated after initialization.

Related projects

There are a number of other interesting projects out there which have some features in common with Palladium. In the following, we will mention a selection.

- DOMINO

  Infrastructure for data analysis (PaaS). A UI for running and examining experiments is provided. Experiments can be run in parallel and notification mechanisms can be set up. Models can be deployed as web services (referring to DOMINO's documentation, the overhead for the HTTP server is about 150ms) and model updates can be scheduled. Supports Python, R, Julia, Octave, and SAS models. Commercial.

- PredictionIO

ML server based on Hadoop / Spark. Two engine templates for Apache Spark MLlib are provided for setting up a recommendation engine or a classification engine. It also allows for gathering new events. Supports Spark MLlib and Scala models (no support for Python, R, Julia). Open source.

- Scikit-Learn Laboratory

  Tool to support experiments performed with scikit-learn. It allows to run various settings on different test sets and to get a summary of the test's results. It does not aim at exposing models as web services. Supports Python models (no support for R or Julia). Open source.

- yhat ScienceOps

  Platform for managing predictive models in production environments (PaaS). A command line tool and GUI are available for model management. It also provides a Load Balancer and can automatically scale the servers as needed. Supports Python and R models. Commercial.

CHAPTER 3

API Reference

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

palladium package

Submodules

palladium.R module

Support for building models using the R programming language.

class palladium.R.**AbstractModel**(*encode_labels=False*, **args*, ***kwargs*)

Bases: *palladium.interfaces.Model*, *palladium.R.ObjectMixin*

fit(*X*, *y=None*)

class palladium.R.**ClassificationModel**(*encode_labels=False*, **args*, ***kwargs*)

Bases: *palladium.R.AbstractModel*

A *Model* for classification problems that uses an R model for training and prediction.

predict(*X*)

predict_proba(*X*)

score(*X*, *y*)

class palladium.R.**DatasetLoader**(*scriptname*, *funcname*, ***kwargs*)

Bases: *palladium.interfaces.DatasetLoader*, *palladium.R.ObjectMixin*

A *DatasetLoader* that calls an R function to load the data.

class palladium.R.**ObjectMixin**(*scriptname*, *funcname*, ***kwargs*)

Bases: object

palladium.cache module

The cache module provides caching utilities in order to provide faster access to data which is needed repeatedly. The disk cache (*diskcache*) which is primarily used during development when loading data from the local harddisk is faster than querying a remote database.

class `palladium.cache.`**`abstractcache`**(*compute_key=None*, *ignore=False*)

Bases: `object`

An abstract class for providing basic functionality for caching function calls. It contains the handling of keys used for caching objects.

`palladium.cache.`**`compute_key_attrs`**(*attrs*)

class `palladium.cache.`**`diskcache`**(**args*, *filename_tmpl=None*, ***kwargs*)

Bases: *`palladium.cache.abstractcache`*

The disk cache stores results of function calls as pickled files to disk. Usually used during development and evaluation to save costly DB interactions in repeated calls with the same data.

Note: Should changes to the database or to your functions require you to purge existing cached values, then those cache files are found in the location defined in `filename_tmpl`.

class `palladium.cache.`**`picklediskcache`**(**args*, *filename_tmpl=None*, ***kwargs*)

Bases: *`palladium.cache.diskcache`*

Same as diskcache, except that standard pickle is used instead of joblib's pickle functionality.

`dump`(*value*, *filename*)

`load`(*filename*)

palladium.dataset module

`DatasetLoader` implementations.

class `palladium.dataset.`**`EmptyDatasetLoader`**

Bases: *`palladium.interfaces.DatasetLoader`*

This *`DatasetLoader`* can be used if no actual data should be loaded. Returns a `(None, None)` tuple.

class `palladium.dataset.`**`SQL`**(*url*, *sql*, *target_column=None*, *ndarray=True*, ***kwargs*)

Bases: *`palladium.interfaces.DatasetLoader`*

A *`DatasetLoader`* that uses `pandas.io.sql.read_sql()` to load data from an SQL database. Supports all databases that SQLAlchemy has support for.

class `palladium.dataset.`**`ScheduledDatasetLoader`**(*impl*, *update_cache_rrule*)

Bases: *`palladium.interfaces.DatasetLoader`*

A *`DatasetLoader`* that loads periodically data into RAM to make it available to the prediction server inside the `process_store`.

`ScheduledDatasetLoader` wraps another *`DatasetLoader`* class that it uses to do the actual loading of the data.

An *update_cache_rrule* is used to define how often data should be loaded anew.

This class' `read()` read method never calls the underlying dataset loader. It will only ever fetch the data from the in-memory cache.

`cache` = {}

`initialize_component`(*config*)

`update_cache`(**args*, ***kwargs*)

class `palladium.dataset.`**`Table`**(*path*, *target_column=None*, *ndarray=True*, ***kwargs*)

Bases: *`palladium.interfaces.DatasetLoader`*

A *`DatasetLoader`* that uses `pandas.io.parsers.read_table()` to load data from a file or URL.

palladium.eval module

Utilities for testing the performance of a trained model.

`palladium.eval.list` (*model_persister*)

`palladium.eval.list_cmd` (*argv=['-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex']*)

List information about available models.

Uses 'model_persister' from the configuration to display a list of models and their metadata.

Usage:pld-list [options]

Options:-h –help Show this screen.

`palladium.eval.test` (*dataset_loader_test*, *model_persister*, *model_version=None*)

`palladium.eval.test_cmd` (*argv=['-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex']*)

Test a model.

Uses 'dataset_loader_test' and 'model_persister' from the configuration to load a test dataset to test the accuracy of a trained model with.

Usage:pld-test [options]

Options:-h –help Show this screen.

--model-version=<version> The version of the model to be tested. If not specified, the newest model will be used.

palladium.fit module

palladium.interfaces module

Interfaces defining the behaviour of Palladium's components.

class `palladium.interfaces.CrossValidationGenerator`

Bases: `object`

A `CrossValidationGenerator` provides train/test indices to split data in train and validation sets.

`CrossValidationGenerator` corresponds to the *cross validation generator* interface of scikit-learn.

class `palladium.interfaces.DatasetLoader`

Bases: `object`

A `DatasetLoader` is responsible for loading datasets for use in training and evaluation.

class `palladium.interfaces.DatasetLoaderMeta` (*name*, *bases*, *attrs*, ***kwargs*)

Bases: `abc.ABCMeta`

class `palladium.interfaces.Model`

Bases: `Dummy`

A `Model` can be `fit()` to data and can be used to `predict()` data.

`Model` corresponds to the *estimators* interface of scikit-learn.

`fit` (*X*, *y=None*)

Fit to data array *X* and possibly a target array *y*.

Returnsself

predict (*X*, ***kw*)

Predict classes for data array *X* with shape n x m.

Some models may accept additional keyword arguments.

ReturnsA numpy array of length n with the predicted classes (for classification problems) or numeric values (for regression problems).

RaisesMay raise a *PredictError* to indicate that some condition made it impossible to deliver a prediction.

predict_proba (*X*, ***kw*)

Predict probabilities for data array *X* with shape n x m.

ReturnsA numpy array of length n x c with a list class probabilities per sample.

Raises`NotImplementedError` if not applicable.

class `palladium.interfaces.`**ModelPersister**

Bases: `object`

activate (*version*)

Set the model with the given *version* to be the active one.

Implies that any previously active model becomes inactive.

Parameters`version` (*str*) – The *version* of the model that's activated.

RaisesLookupError if no model with given *version* exists.

delete (*version*)

Delete the model with the given *version* from the database.

Parameters`version` (*str*) – The *version* of the model that's activated.

RaisesLookupError if no model with given *version* exists.

list_models ()

List metadata of all available models.

ReturnsA list of dicts, with each dict containing information about one of the available models. Each dict is guaranteed to contain the `version` key, which is the same version number that *ModelPersister.read()* accepts for loading specific models.

list_properties ()

List properties of *ModelPersister* itself.

ReturnsA dictionary of key and value pairs, where both keys and values are of type `str`. Properties will usually include `active-model` and `db-version` entries.

read (*version=None*)

Returns a *Model* instance.

Parameters`version` (*str*) – *version* may be used to read a specific version of a model. If *version* is `None`, returns the active model.

ReturnsThe model object.

RaisesLookupError if no model was available.

upgrade (*from_version=None*, *to_version='n/a'*)

Upgrade the underlying database to the latest version.

Newer versions of Palladium may require changes to the *ModelPersister*'s database. This method provides an opportunity to run the necessary upgrade steps.

It's the `ModelPersister`'s responsibility to keep track of the Palladium version that was used to create and upgrade its database, and thus to determine the upgrade steps necessary.

`write` (*model*)

Persists a `Model` and returns a new version number.

It is the `ModelPersister`'s responsibility to annotate the 'version' information onto the model before it is saved.

The new model will initially be inactive. Use `ModelPersister.activate()` to activate the model.

ReturnsThe new model's version identifier.

exception `palladium.interfaces.`**`PredictError`** (*error_message*, *error_code=-1*)

Bases: `Exception`

Raised by `Model.predict()` to indicate that some condition made it impossible to deliver a prediction.

class `palladium.interfaces.`**`PredictService`**

Bases: `object`

Responsible for producing the output for the '/predict' HTTP endpoint.

`palladium.interfaces.`**`annotate`** (*obj*, *metadata=None*)

palladium.julia module

class `palladium.julia.`**`AbstractModel`** (*fit_func*, *predict_func*, *fit_kwargs=None*, *predict_kwargs=None*, *encode_labels=False*)

Bases: `palladium.interfaces.Model`

`fit` (*X*, *y*)

`predict` (*X*)

class `palladium.julia.`**`ClassificationModel`** (*fit_func*, *predict_func*, *fit_kwargs=None*, *predict_kwargs=None*, *encode_labels=False*)

Bases: `palladium.julia.AbstractModel`

`score` (*X*, *y*)

`palladium.julia.`**`make_bridge`** ()

palladium.persistence module

palladium.server module

HTTP API implementation.

class `palladium.server.`**`PredictService`** (*mapping*, *params=()*, *entry_point='/predict'*, *decorator_list_name='predict_decorators'*, *predict_proba=False*, ***kwargs*)

Bases: `object`

A default `palladium.interfaces.PredictService` implementation.

Aims to work out of the box for the most standard use cases. Allows overriding of specific parts of its logic by using granular methods to compose the work.

`do` (*model*, *request*)

`initialize_component` (*config*)

params_from_data (*model*, *data*)

Retrieve additional parameters (keyword arguments) for `model.predict` from request *data*.

Parameters

- **model** – The `Model` instance to use for making predictions.
- **data** – A dict-like with the parameter data, typically retrieved from `request.args` or similar.

predict (*model*, *sample*, ***kwargs*)

response_from_exception (*exc*)

response_from_prediction (*y_pred*, *single=True*)

Turns a model's prediction in *y_pred* into a JSON response.

sample_from_data (*model*, *data*)

Convert incoming sample *data* into a numpy array.

Parameters

- **model** – The `Model` instance to use for making predictions.
- **data** – A dict-like with the sample's data, typically retrieved from `request.args` or similar.

class `palladium.server.`**PredictStream**

Bases: `object`

A class that helps make predictions through stdin and stdout.

listen (*io_in*, *io_out*, *io_err*)

Listens to provided io stream and writes predictions to output. In case of errors, the error stream will be used.

process_line (*line*)

`palladium.server.`**create_predict_function** (*route*, *predict_service*, *decorator_list_name*)

Creates a predict function and registers it to the Flask app using the route decorator.

Parameters

- **route** (*str*) – Path of the entry point.
- **predict_service** (palladium.interfaces.PredictService) – The predict service to be registered to this entry point.
- **decorator_list_name** (*str*) – The decorator list to be used for this predict service. It is OK if there is no such entry in the active Palladium config.

ReturnsA predict service function that will be used to process predict requests.

`palladium.server.`**devserver_cmd** (*argv=['-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex']*)

Serve the web API for development.

Usage:pld-devserver [options]

Options:-h –help Show this screen.

--host=<host>	The host to use [default: 0.0.0.0].
--port=<port>	The port to use [default: 5000].
--debug=<debug>	Whether or not to use debug mode [default: 0].

`palladium.server.`**`make_ujson_response`**`(`*obj*, *status_code=200*`)`

Encodes the given *obj* to json and wraps it in a response.

ReturnsA Flask response.

`palladium.server.`**`predict`**`(`*model_persister*, *predict_service*`)`

`palladium.server.`**`stream_cmd`**`(`*argv=['-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex']*`)`

Start the streaming server, which listens to stdin, processes line by line, and returns predictions.

The input should consist of a list of json objects, where each object will result in a prediction. Each line is processed in a batch.

Example input (must be on a single line):

[{"sepal length": 1.0, "sepal width": 1.1, "petal length": 0.7,"petal width": 5}, {"sepal length": 1.0, "sepal width": 8.0, "petal length": 1.4, "petal width": 5}]

Example output:

["Iris-virginica","Iris-setosa"]

An input line with the word 'exit' will quit the streaming server.

Usage:pld-stream [options]

Options:-h –help Show this screen.

palladium.util module

Assorted utilties.

class `palladium.util.`**`Config`**

Bases: `dict`

A dictionary that represents the app's configuration.

Tries to send a more user friendly message in case of KeyError.

`palladium.util.`**`Partial`**`(`*func*, ***kwargs*`)`

Allows the use of partially applied functions in the configuration.

class `palladium.util.`**`PluggableDecorator`**`(`*decorator_config_name*`)`

Bases: `object`

class `palladium.util.`**`ProcessStore`**`(`**args*, ***kwargs*`)`

Bases: `collections.UserDict`

class `palladium.util.`**`RruleThread`**`(`*func*, *rrule*, *sleep_between_checks=60*`)`

Bases: `threading.Thread`

Calls a given function in intervals defined by given recurrence rules (from *datetuil.rrule*).

`run``()`

`palladium.util.`**`apply_kwargs`**`(`*func*, ***kwargs*`)`

Call *func* with kwargs, but only those kwargs that it accepts.

`palladium.util.`**`args_from_config`**`(`*func*`)`

Decorator that injects parameters from the configuration.

`palladium.util.`**`create_component`**`(`*specification*`)`

`palladium.util.`**`get_config`**`(`***extra*`)` `palladium.util.`**`get_metadata`**`(`*error_code=0*, *error_message=None*, *status='OK'*`)`

`palladium.util.`**`initialize_config`**`(`***extra*`)`

`palladium.util.`**`memory_usage_psutil`**`()`

Return the current process memory usage in MB.

`palladium.util.`**`resolve_dotted_name`**`(`*dotted_name*`)`

`palladium.util.`**`session_scope`**`(`*session*`)`

Provide a transactional scope around a series of operations.

`palladium.util.`**`timer`**`(`*log=None*, *message=None*`)`

`palladium.util.`**`upgrade`**`(`*model_persister*, *from_version=None*, *to_version=None*`)`

`palladium.util.`**`upgrade_cmd`**`(`*argv=['-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex']*`)`

Upgrade the database to the latest version.

Usage:pld-ugprade [options]

Options:

--from=<v> Upgrade from a specific version, overriding the version stored in the database.

--to=<v> Upgrade to a specific version instead of the latest version.

-h –help Show this screen.

`palladium.util.`**`version_cmd`**`(`*argv=['-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex']*`)`

Print the version number of Palladium.

Usage:pld-version [options]

Options:-h –help Show this screen.

palladium.wsgi module

Module to be used for exposing the web service with other WSGI servers. It initializes Palladium's configuration and provides the Flask object.

Module contents

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index